# KUBERNETES

## WHY KUBERNETES:

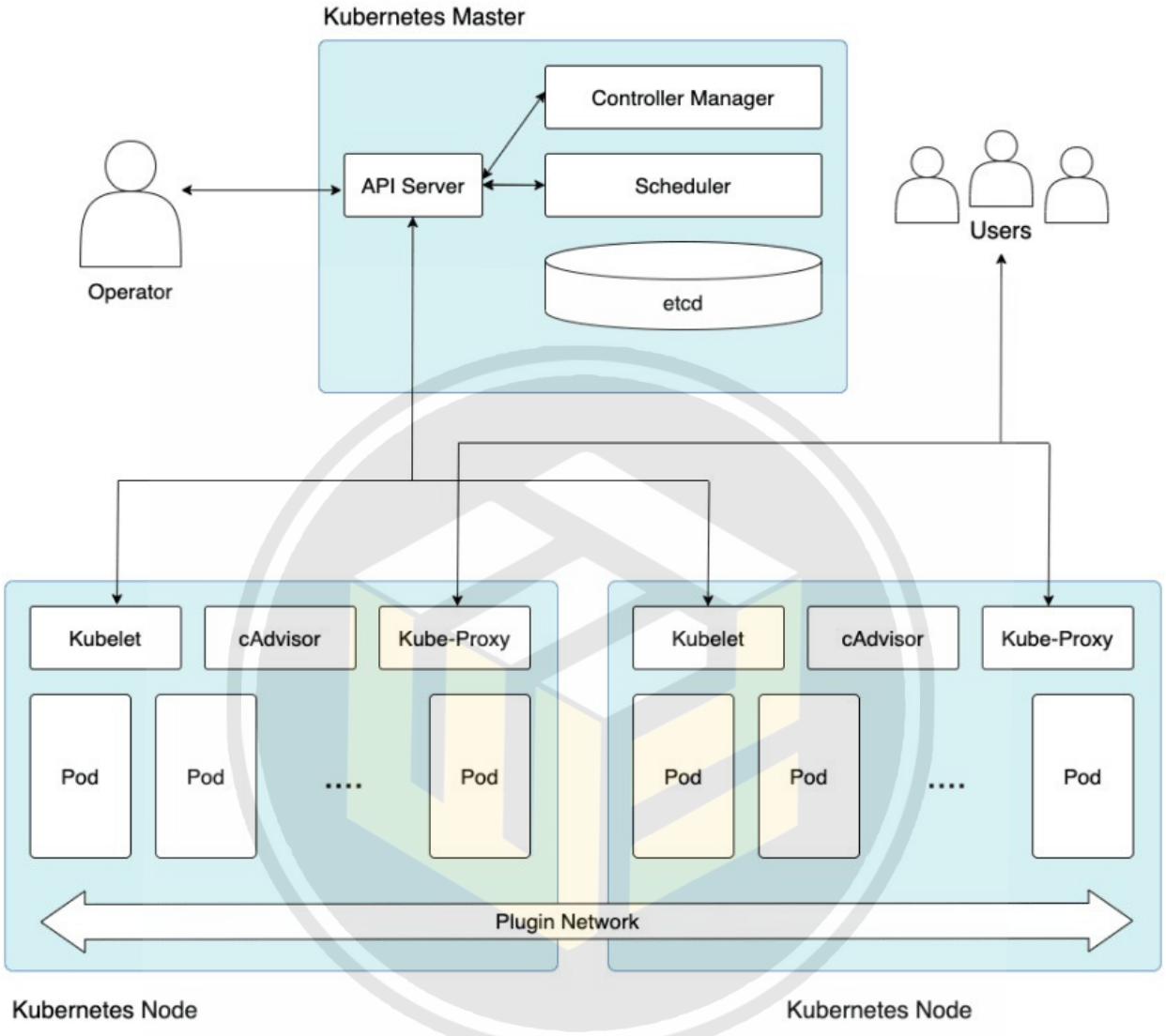
Earlier We used Docker Swarm as a container orchestration tool that we used to manage multiple containerized applications on our environments.

| FEATURES | DOCKER SWARM | KUBERNETES |
|---|---|---|
| Setup | Easy | Complex |
| Auto Scaling | No Auto Scaling | Auto Scaling |
| Community | Good Community | Greater community for users like documentation, support and resources |
| GUI | No GUI | GUI |

## KUBERNETES:

- IT is an open-source container orchestration platform.
- It is used to automates many of the manual processes like deploying, managing, and scaling containerized applications.
- Kubernetes was developed by GOOGLE using GO Language.
- Google donated K8's to CNCF in 2014.

- 1st version was released in 2015.




## CLUSTER:

- It is a group of serversIt will have both manager and worker nodes.
- Master Node is used to assign tasks to worker nodes.
- Worker node will perform the task.
- we have 4 components in Master Node
  a. API Server
  b. ETCD
  c. Controllers-manager
  d. Schedulers
- we have 4 components in Worker Node.
  a. Kubelet
  b. Kube-Proxy
  c. Pod
  d. Container

1. **API Server:** It is used to accept the request from the user and store the request in ETCD.
2. **ETCD:** It is like a database to our k8's. it is used to store the requests from API Server in the KEY-VALUE format.
3. **Scheduler**: It is used to search pending tasks which are present in ETCD. If any pending task found in ETCD, it will schedule in worker node. It will decide in which worker node our task should gets executed. It will decide by communication with the kubelet in worker node.
4. **Controllers:** It is used to perform the operations which is scheduled by the scheduler. it wil control the containers creation in worker nodes.
5. **kubelet:** Agent ensures that pod is running or not.
6. **kube-proxy:** It is used to maintain a network connection between worker and manager nodes.
7. **pod:** A group of one or more containers.
8. **container:** It is a virtual machine which does not have any OS. it is used to run the applications in worker nodes.

## KUBERNETES CLUSTER SETUP:

A Kubernetes cluster is a set of nodes (physical or virtual machines) that run containerized applications.

There are multiple ways to setup kubernetes cluster.

1.SELF MANAGER K8'S CLUSTER

a.mini kube (single node cluster)

b.kubeadm(multi node cluster)

c. KOPS

2. CLOUD MANAGED K8'S CLUSTER

a. AWS EKS

b.AZURE AKS

c.GCP GKS

d.IBM IKE

In a Kubernetes cluster, the following key components work together:

**Node:** A node is a physical or virtual machine that runs containerized applications. Each node in the cluster typically runs a container runtime (such as Docker) to manage and execute containers.

**Master:** The master is responsible for managing the overall state of the cluster. It controls and coordinates activities on the nodes, schedules containers, and ensures that the desired state of the applications matches the actual state.

**Pod:** The smallest deployable unit in Kubernetes is a pod. A pod represents a single instance of a running process in a cluster and can contain one or more containers that share the same network namespace and storage. Containers within the same pod can communicate with each other using localhost.

**Controller:** Controllers are responsible for ensuring that the desired state of the system matches the actual state. Examples include ReplicaSets, Deployments, and StatefulSets, which manage the deployment and scaling of pods.

**Service:** A Kubernetes service provides a stable endpoint (IP address and DNS name) for accessing a set of pods. It allows applications to discover and communicate with each other, both within and outside the cluster.

## MINIKUBE:

- It is a tool used to setup single node cluster on K8's.
- It contains API Servers, ETDC database and container runtime
- It helps you to containerized applications.
- It is used for development, testing, and experimentation purposes on local. Here Master and worker runs on same machine
- It is a platform Independent.
- By default it will create one node only.
- Installing Minikube is simple compared to other tools.

NOTE: But we dont implement this in real-time

## MINIKUBE SETUP:

### REQUIREMENTS:

a. 2 CPUs or more
b. 2GB of free memory
c. 20GB of free disk space
d. Internet connection
e. Container or virtual machine manager, such as: Docker.

## COMMANDS TO SETUP MINIKUBE ON UBUNTU:

- apt update -y
- sudo apt install curl wget apt-transport-https -y
- sudo curl -fsSL https://get.docker.com -o get-docker.sh
- sudo sh get-docker.sh
- sudo curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
- sudo mv minikube-linux-amd64 /usr/local/bin/minikube
- sudo chmod +x /usr/local/bin/minikube
- sudo minikube version
- sudo curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
- sudo curl -LO "https://dl.k8s.io/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl.sha256"
- sudo echo "$(cat kubectl.sha256) kubectl" | sha256sum --check
- sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl
- sudo kubectl version --client
- sudo kubectl version --client --output=yaml
- sudo minikube start --driver=docker --force

## COMMANDS TO SETUP MINIKUBE ON AMAZON LINUX:

- yum install docker -y
- systemctl start docker
- systemctl status docker
- curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
- sudo mv kubectl /usr/local/bin/kubectl
- sudo chmod +x /usr/local/bin/kubectl
- curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
- sudo install minikube-linux-amd64 /usr/local/bin/minikube
- sudo yum install iptables -y
- yum install conntrack -y
- minikube start --driver=docker --force
- minikube status

## KOPS SETUP ON AMAZON LINUX:

STEP-1: LAUNCH INSTANCE WITH T2.MICRO AND 30 GB SSD

STEP-2: INSTALL AWS CLI

- curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
- unzip awscliv2.zip
- sudo ./aws/install

TO  CHECK VERSION: /usr/local/bin/aws --version

TO SET PATH:

- vim .bashrc
- export PATH=$PATH:/usr/local/bin/
- source .bashrc
- aws --version

STEP-3: INSTALL KOPS & KUBECTL

- curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
- wget https://github.com/kubernetes/kops/releases/download/v1.24.1/kops-linux-amd64

PERMISSIONS: chmod +x kops-linux-amd64 kubectl

MOVE FILES:  mv kubectl /usr/local/bin/kubectl

MOVE FILES:  mv kops-linux-amd64  /usr/local/bin/kops

TO SEE VERSION: kubectl version && kops version

STEP-4: CREATE IAM USER WITH ADMIN PERMISSIONS AND CONFIGURE IT IN ANY REGION WITH TABLE FORMAT

STEP-5: CREATE INFRA SETUP

TO CREATE BUCKET: aws s3api create-bucket --bucket mustafask.k8s.local --region us-east-1

TO ENABLE VERSION: aws s3api put-bucket-versioning --bucket mustafask.k8s.local --region us-east-1 --versioning-configuration Status=Enabled

EXPORT CLUSTER DATA INTO BUCKET: export KOPS_STATE_STORE=s3://mustafask.k8s.local

GENERATE-KEY: ssh-keygen

TO CREATE CLUSTER: kops create cluster --name musta.k8s.local --zones us-east-1a --master-size t2.medium --node-size t2.micro

TO SEE THE CLUSTER: kops get cluster

IF YOU WANT TO EDIT THE CLUSTER: kops edit cluster cluster_name

TO RUN THE CLUSTER: kops update cluster --name musta.k8s.local --yes --admin

TO DELETE THE CLUSTER: Kops delete cluster --name musta.k8s.local --yes

* list clusters with: kops get cluster

* edit this cluster with: kops edit cluster musta.k8s.local

* edit your node instance group: kops edit ig --name=musta.k8s.local nodes-us-east-1a

* edit your master instance group: kops edit ig --name=musta.k8s.local master-us-east-1a

Finally configure your cluster with: kops update cluster --name musta.k8s.local --yes --admin

## KUBECTL:

- kubectl is the CLI which is used to interact with a Kubernetes cluster.
- We can create, manage pods, services, deployments, and other resources We can also monitoring, troubleshooting, scaling and updating the pods.
- To perform these tasks it communicates with the Kubernetes API server.
- It has many options and commands, to work on.
- The configuration of kubectl is in the $HOME/.kube directory.
- The latest version is 1.28

### SYNTAX:

kubectl [command] [TYPE] [NAME] [flags]

kubectl api-resources : to list all api resources

## POD:

- It is a smallest unit of deployment in K8's.
- It is a group of containers.
- Pods are ephemeral (short living objects)
- Mostly we can use single container inside a pod but if we required, we can create multiple containers inside a same pod.
- when we create a pod, containers inside pods can share the same network namespace, and can share the same storage volumes .
- While creating pod, we must specify the image, along with any necessary configuration and resource limits.
- K8's cannot communicate with containers, they can communicate with only pods.
- We can create this pod in two ways,
  - 1. Imperative(command)
  - 2. Declarative (Manifest file)

# POD CREATION:

## IMPERATIVE:

- The imperative way uses kubectl command to create pod.
- This method is useful for quickly creating and modifying the pods.

**SYNTAX**: kubectl run pod_name --image=image_name

**COMMAND**: ***kubectl run pod-1 --image=nginx***

**kubectl** : command line tool run : action

**pod-1** : name of pod

**nginx** : name of image

## DECLARATIVE:

- The Declarativ tive way we need to create a Manifest file in YAML Extension. This file contains the desired state of a Pod.
- It takes care of creating, updating, or deleting the resources.
- This manifest file need to follow the yaml indentation.
- YAML file consist of KEY-VALUE Pair.
- Here we use create or apply command to execute the Manifest file.

**SYNTAX**: kubectl create/apply -f file_name

**CREATE**: if you are creating the object for first time we use create only.

**APPLY**: if we change any thing on files and changes need to apply the resources.

# MANIFEST FILE:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

- **apiVersion:** For communicating with master node
- **Kind:** it is a type of resource
- **Metadata:** data about pod
- **Spec:** it is a specifications of a container
- **Name :** Name of the container
- **Image :** image name
- **Ports :** To Expose the Application

## PODS COMMANDS:

- To get all the pods: kubetctl get pods (or) kubectl get pod (or) kubectl get po
- To delete a pod: kubectl delete pod pod_name
- To get IP of a pod: kubectl get po pod_name -o wide
- To get IP of all pods: kubectl get po -o wide
- To get all details of a pod: kubectl describe pod podname
- To get all details of all pods: kubectl describe po
- To get the pod details in YAML format: kubectl get pod pod-1 -o yaml
- To get the pod details in JSON format: kubectl get pod pod-1 -o json
- To enter into a pod: kubectl exec -it pod_name -c cont_name bash
- To get the logging info of our pod: kubectl logs pod_name
- To get the logs of containers inside the pod: kubectl logs pod_name -c cont-name

## LABELS, SELECTORS, AND NODE SELECTORS:

## LABELS:

- Labels are used to organize Kubernetes Objects such as Pods, nodes, etc.

- You can add multiple labels over the Kubernetes Objects.
- Labels are defined in key-value pairs.
- Labels are similar to tags in AWS or Azure where you give a name to filter the resources quickly.
- You can add labels like environment, department or anything else according to you.

## LABEL-SELECTORS:

- Once the labels are attached to the Kubernetes objects those objects will be filtered out with the help of labels-Selectors known as Selectors.
- The API currently supports two types of label-selectors equality-based and set-based. Label selectors can be made of multiple selectors that are comma-separated.

## NODE SELECTORS:

- Node selector means selecting the nodes. Node selector is used to choose the node to apply a particular command.
- This is done by Labels where in the manifest file, we mentioned the node label name. While running the manifest file, master nodes find the node that has the same label and create the pod on that container.
- Make sure that the node must have the label. If the node doesn't have any label then, the manifest file will jump to the next node.

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-1
  labels:
    env: testing
    department: DevOps
spec:
  containers:
    - name: containers1
      image: ubuntu
      command: ["/bin/bash", "-c", "while true; do echo This is our Pod; sleep 5 ; done"]
```

To see the list of pods with labels: kubectl get pods --show-labels

Now, I want to list those pods that have the label env=testing.

kubectl get pods -l env=testing

kubectl get pods -l depar tment!=DevOps

As we have discussed earlier, there are two types of label-selectors equality and set-based.

This is the example of equality based where we used equalsTo(=).

Now, Suppose I forgot to add the label through the declarative(via manifest) method. So, I can add labels after creating the pods as well which is known as the imperative(via command) method.

*kubectl label pods Pod_name Location=India*

As we discussed the type of label-selector, let's see the example of a set-based label-selector where we use in, notin, and exists.

Lets list all those pods that have an env label with value for either testing or development.

kubectl get pods -l 'env in (testing, development)'

we are trying to list all those pods that don't have the India or US value for the Location key in the Label.

*kubectl get pods -l 'Location not in (India, US)'*

We can also delete the pods as per the label.

Let's deleted all those pods that don't have the China value for the location key in the Label.

*kubectl delete pod -l Location!=China*

## KUBERNETES SERVICES

- Service is a method for exposing Pods in your cluster.
- Each Pod gets its own IP address But we need to access from IP of the Node.. If you want to access pod from inside we use Cluster-IP.
- If the service is of type NodePort or LoadBalancer, it can also be accessed. from outside the cluster.
- It enables the pods to be decoupled from the network topology, which makes it easier to manage and scale applications

## TYPES OF SERVICES:

1. CLUSTER-IP
2. NODE PORT
3. LOAD BALANCER

4. EXTERNALNAME

**ClusterIP:** A ClusterIP service provides a stable IP address and DNS name for pods within a cluster. This type of service is only accessible within the cluster and is not exposed externally.

**NodePort:** A NodePort service provides a way to expose a service on a static port on each node in the cluster. This type of service is accessible both within the cluster and externally, using the node's IP address and the NodePort.

**LoadBalancer**: A LoadBalancer service provides a way to expose a service externally, using a cloud provider's load balancer. This type of service is typically used when an application needs to handle high traffic loads and requires automatic scaling and load balancing capabilities.

**ExternalName:** An ExternalName service provides a way to give a service a DNS name that maps to an external service or endpoint. This type of service is typically used when an application needs to access an external service, such as a database or API, using a stable DNS name.

## COMPONENTS OF SERVICES :

A service is defined using a Kubernetes manifest file that describes its properties and specifications. Some of the key properties of a service include:

- **Selector**: A label selector that defines the set of pods that the service will route traffic to.
- **Port**: The port number on which the service will listen for incoming traffic.
- **TargetPort**: The port number on which the pods are listening for traffic.
- **Type**: The type of the service, such as ClusterIP, NodePort, LoadBalancer, or ExternalName.

## CLUSTER-IP:

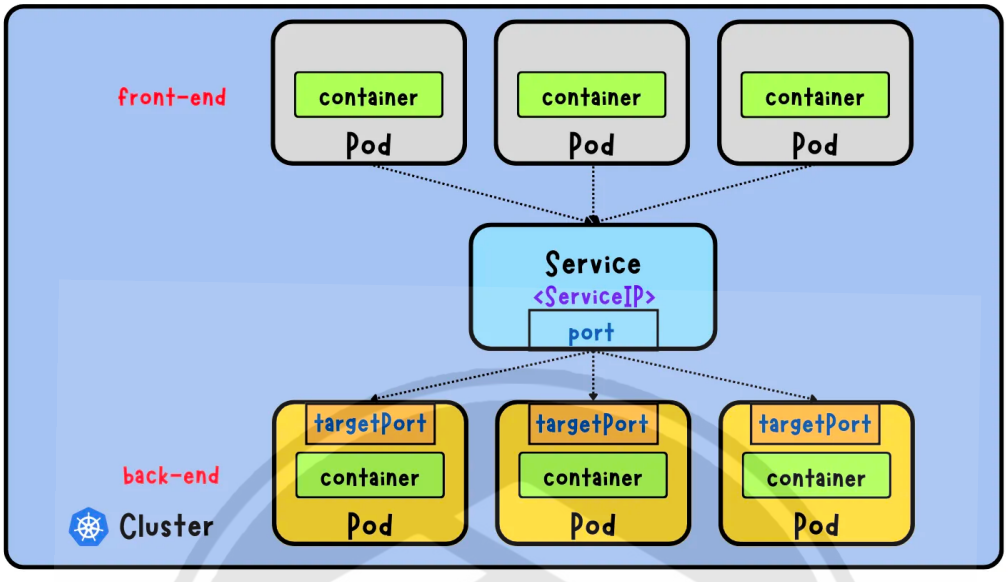
```yaml
apiVersion: v1
kind: Pod
metadata:
  name: frontend
  labels:
    app: swiggy
spec:
  containers:
    - name: cont1
      image: rahamshaik/tic-tac-toe-1:latest
      ports:
      - containerPort: 80
```

To deploy the application we create a container, which stores inside the pod. After container is created we will be not able to access the application. Because we cannot access the pods and ports from the cluster.

To Avoid this we are creating the Services.


- In this code we are exposing the applcation. here we use clusterip service
- By using this we can access application inside the cluster only.
- But if we want to access the application from outside we need to use nodeport.
- ClusterIP will assign one ip for service to access the pod.
- Just Replace ***ClusterIP=NodePort***
- kubectl apply -f filename.yml

# Three Types of Services: ClusterIP




port: port on Service

targetPort: port on Pod (destination)

## NODEPORT:

```yaml
---
apiVersion: v1
kind: Service
metadata:
  name: frontend
spec:
  type: NodePort
  selector:
    app: swiggy
  ports:
    - port: 80
      targetPort: 80
      nodePort: 30001
```

In this code we are exposed the application from anywhere (inside & outside)

We need to Give public ip of node where pod is running. Node Port Range= 30000 - 32767

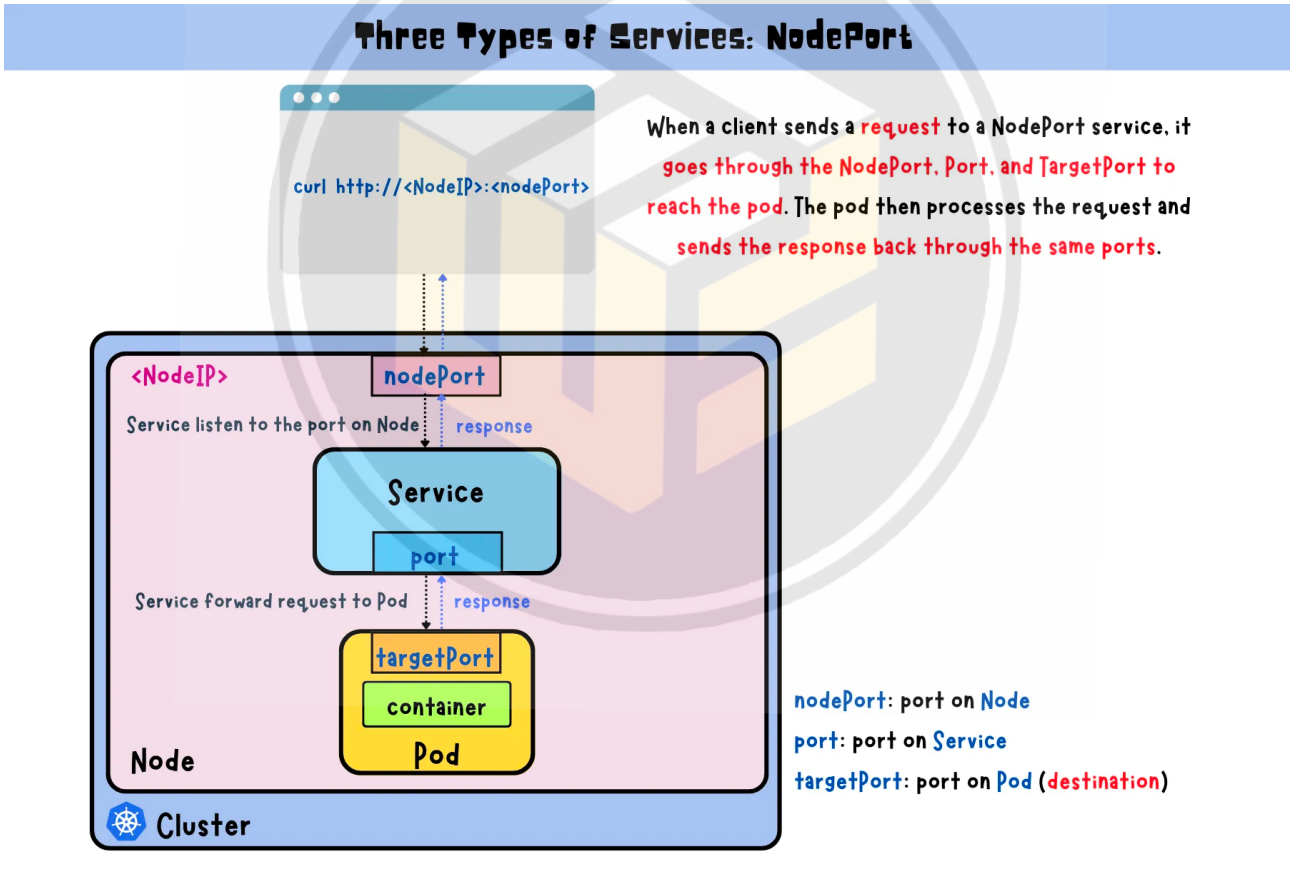
here i hae defined port number as 30001

if we dont specify the port it will assign automatically. kubectl apply -f filename.yml.

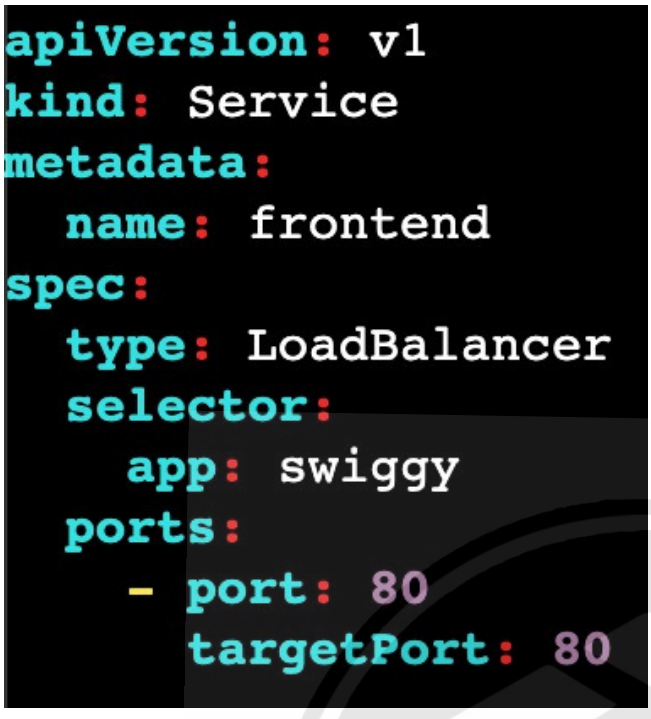
NodePort expose service on a static port on each node. NodePort services are typically used for smaller applications with a lower traffic volume.

To avoid this we are using the LoadBalancer service.

Just Replace ***NodePort=LoadBalancer***

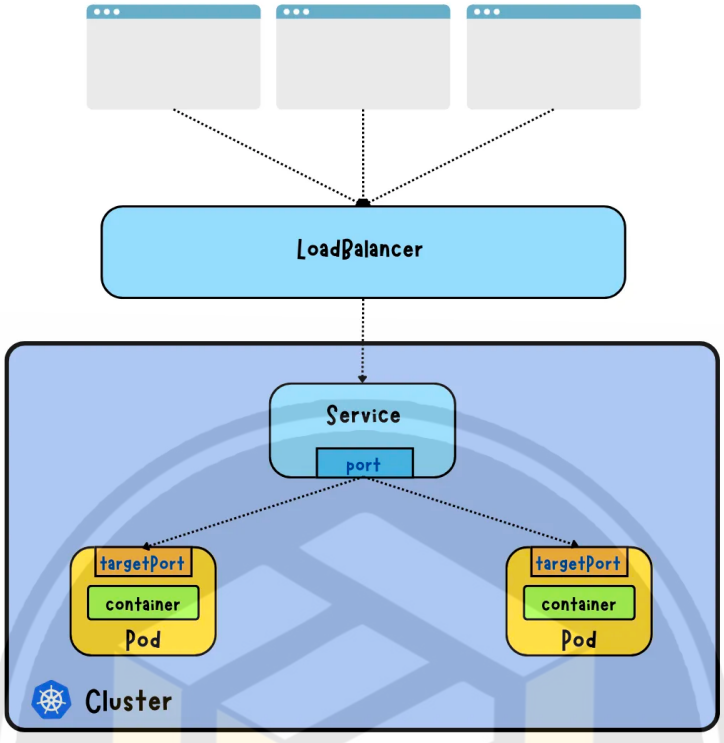



## LOADBALANCER:




- In LoadBalaner we can expose application externally with the help of Cloud Provider LoadBalancer.
- it is used when an application needs to handle high traffic loads and requires automatic scaling and load balancing capabilities.
- After the LoadBalancer service is created, the cloud provider will created the Load Balancer.
- This IP address can be used by clients outside the cluster to access the service.
- The LoadBalancer service also automatically distributes incoming traffic across the pods that match the selector defined in the YAML manifest.
- access : publicip:port and LB url
- http://a0dce056441c04035918de4bfb5bff97-40528368.us- east-1.elb.amazonaws.com/

# Three Types of Services: LoadBalancer




## COMMANDS FOR SERVICES:

**1. Creates a new service based on the YAML file**

kubectl create -f filename

**2. Create a ClusterIP service**

kubectl create service clusterip <service-name> --tcp=<port>:<targetPort>

**3. Create a NodePort service**

kubectl create service nodeport <service-name> --tcp=<port>:<targetPort>

--node-port=<nodePort>

**4. Create a LoadBalancer service**

kubectl create service **loadbalancer** <service-name> --tcp=<port>:<targetPort>

**5. Create a service for a pod**

kubectl expose pod <pod-name> --name=<service-name> \

--port=<port> --target-port=<targetPort> --type=<service-type>

**6. Create a service for a deployment**

kubectl expose deployment <deployment-name> --name=<service-name> \

--port=<port> --target-port=<targetPort> --type=<service-type>

**7. Retrieve a list of services that have been created**

kubectl get services

**8. Retrieve detailed information about a specific service**

kubectl describe services <service-name>

## REPLICAS IN KUBERNETES:

- Before Kubernetes, other tools did not provide important and customized features like scaling and replication.
- When Kubernetes was introduced, replication and scaling were the premium features that increased the popularity of this container orchestration tool.
- Replication means that if the pod's desired state is set to 3 and whenever any pod fails, then with the help of replication, the new pod will be created as soon as possible. This will lead to a reduction in the downtime of the application.
- Scaling means if the load becomes increases on the application, then Kubernetes increases the number of pods according to the load on the application.

ReplicationController is an object in Kubernetes that was introduced in v1 of Kubernetes which helps to meet the desired state of the Kubernetes cluster from the current state. ReplicationController works on equality-based controllers only.

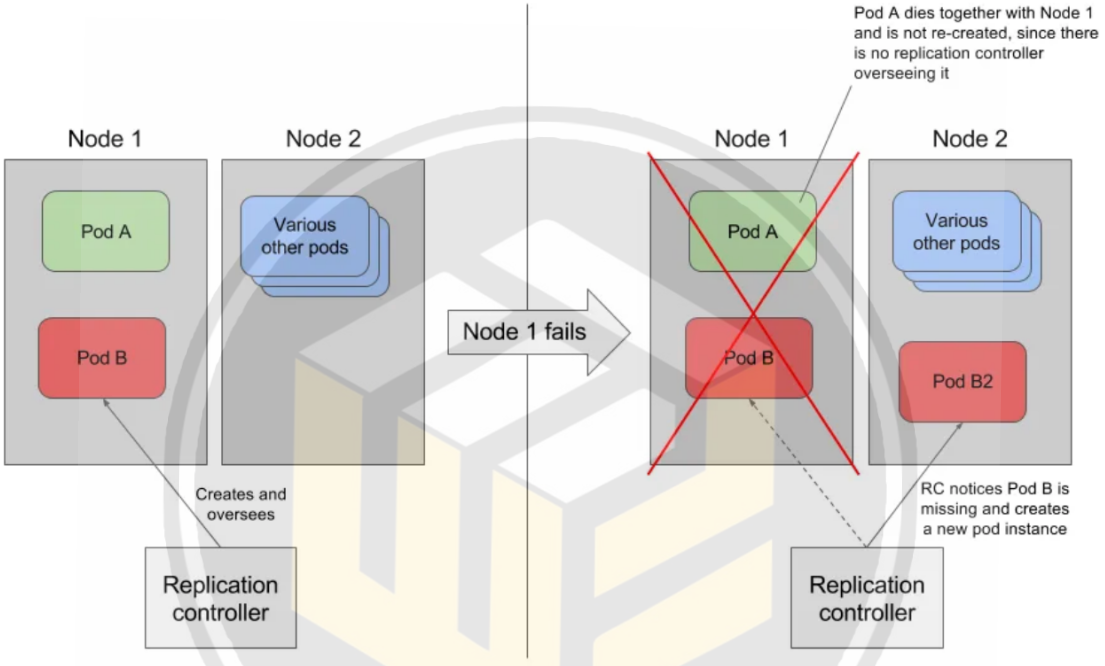
ReplicaSet is an object in Kubernetes and it is an advanced version of ReplicationController. ReplicaSet works on both equality-based controllers and set-based controllers.

## REPLICATION CONTROLLER:

- Replication controller can run specific number of pods as per our requirement.
- It is the responsible for managing the pod lifecycle
- It will make sure that always pods are up and running.
- If there are too many pods, RC will terminates the extra pods.
- If there are too less RC will create new pods.
- This Replication controller will have slef-healing capability, that means automatically it will creates.

- If a pod is failed, terminated or deleted then new pod will get created automatically. Replication Controllers use labels to identify the pods that they are managing.
- We need to specifies the desired number of replicas in YAML file.

## Replication Controller




```yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-replicationcontroller
spec:
  replicas: 3
  selector:
    app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-container
          image: my-image:latest
          ports:
          - containerPort: 80
```

**SELECTOR:** It select the resources based on labels. Labels are key value pairs.

This will helps to pass a command to the pods with lable app=nginx

## COMMANDS FOR REPLICATION CONTROLLER:

- TO EXECUTE : kubectl create -f file_name.yml
- TO GET : kubectl get rc
- TO DESCRIBE : kubectl describe rc/nginx
- TO SCALE UP : kubectl scale rc rc_name --replicas 5
- TO SCALE DOWN : kubectl scale rc rc_name --replicas 2 (drops from 5 to 2)
- TO DELETE REPLICA CONTROLLER : kubectl delete rc rc_name

IF WE DELETE RC, PODS ARE ALSO GETS DELETED, BUT IF WE DON'T WANT TO DELETE PODS, WE WANT TO DELETE ONLY REPLICA SETS THEN

- kubectl delete rc rc_name --cascade=orphan
- kubectl get rc rc_name
- kubectl get pod

Now we deleted the RC but still pods are present, if we want to assign this pods to another RC use the same selector which we used on the RC last file.

## REPLICASET:

- it is nothing but group of identical pods. If one pod crashes automatically replica sets gives one more pod immediately.
- it uses labels to identify the pods
- Difference between RC and RS is selector and it offers more advanced functionality.
- The key difference is that the replication controller only supports ***based selectors*** whereas the replica set supports
- it monitoring the number of replicas of a pod running in a cluster and creating or deleting new pods.
- It also provides better control over the scaling of the pod.
- A ReplicaSet identifies new Pods to acquire by using its selector.
- we can provide multiple values to same key.

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: my-replicaset
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-container
          image: my-image:latest
          ports:
          - containerPort: 80
```

**Replicas**: Number of pod copies we need to create Matchelabel: label we want to match with pods Template: This is the template of pod.

**Note**: give apiVersion: apps/v1 on top

**Labels** : mandatory to create RS (if u have 100

pods in a cluster we can inform which pods we need to take care by using labels) if u labeled some pods a my-app then all the pods with label my-app will be managed.

## COMMANDS FOR REPLICASET:

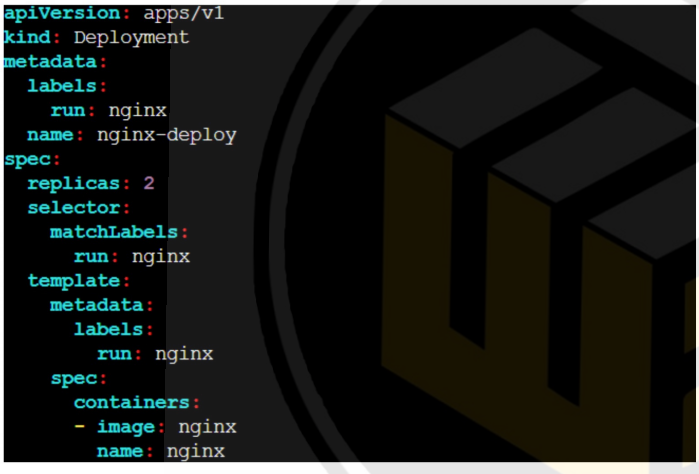
- TO EXECUTE : kubectl create -f replicaset-nginx.yaml
- TO LIST : kubectl get replicaset/rs
- TO GET INFO : kubectl get rs -o wide
- TO GET IN YAML : kubectl get rs -o yaml
- TO GET ALL RESOURCES : kubectl get all

***Now delete a pod and do list now it will be created automatically***

- TO DELETE A POD : Kubectl delete po pod_name
- TO DELETE RS : kubectl delete rs
- TO SHOW LABLES OF RS : Kubectl get po --show-labels
- TO SHOW POD IN YAML : Kubectl get pod -o yaml

## KUBERNETES DEPLOYMENT:

- It has features of Replicaset and some other extra features like updating and rollbacking to a particular version.
- The best part of Deployment is we can do it without downtime.
- you can update the container image or configuration of the application. Deployments also provide features such as versioning, which allows you to track the history of changes to the application.
- It has a pause feature, which allows you to temporarily suspend updates to the application
- Scaling can be done manually or automatically based on metrics such as CPU utilization or requests per second.
- Deployment will create ReplicaSet, ReplicaSet will created Pods.
- If you delete Deployment, it will delete ReplicaSet and then ReplicaSet will delete Pods.



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    run: nginx
  name: nginx-deploy
spec:
  replicas: 2
  selector:
    matchLabels:
      run: nginx
  template:
    metadata:
      labels:
        run: nginx
    spec:
      containers:
      - image: nginx
        name: nginx
```

- Replicas: Number of pod copies we need to create
- MatchLabels: label we want to match with pods
- Template: This is the template of pod.
- Labels : mandatory to create RS (if u have 100
- pods in a cluster we can inform which pods we need to take care by using labels) if u labeled some pods a raham then all the pods with label raham will be managed.

## COMMANDS FOR DEPLOYMENT USING MANIFEST FILE:

- To create a replica set : kubectl create -f replicaset-nginx.yaml
- To see a deployment : kubectl get deployment/deploy
- To see full details of deployment : kubectl get deploy -o wide
- To view in YAML format : kubectl get deploy -o yaml
- To get all info : kubectl describe deploy

## COMMAND FOR DEPLOYMENT USING IMPERATIVE WAY:

*kubectl create deployment deployment-name --image=image-name --replicas=4*

**NOTE**: Now delete a pod and do list now it will be created automatically.

- To delete a pod: Kubectl delete po pod_name
- To check logs : kubectl logs pod_name
- To delete a deployment : kubectl delete deploy deploy_name

When you inspect the Deployments in your cluster, the following fields are displayed:

- **NAME** lists the names of the Deployments in the namespace.
- **READY** displays how many replicas of the application are available to your users. It follows the pattern ready/desired.
- **UP**-TO-DATE displays the number of replicas that have been updated to achieve the desired state.
- **AVAILABLE** displays how many replicas of the application are available to your users.
- **AGE** displays the amount of time that the application has been running.

## COMMANDS FOR DEPLOYMENT:

- update the container image:
- kubectl set image deployment/deployment-name my-container=image-name (or) kubectl edit deployment/deployment-name
- To roll back: kubectl rollout status deployment/deployment-name
- To check the status: kubectl rollout status deployment/deployment-name
- To get the history: kubectl rollout history deployment/deployment-name
- To rollback specific version: kubectl rollout undo deployment/deployment-name --to-revision=number
- To pause a pod in deployment: kubectl rollout pause deployment/deployment-name
- To un-pause a pod in deployment: kubectl rollout resume deployment/deployment-name

## DEPLOYMENT SCALING:

Lets assume we have 3 pods, if i want to scale it to 10 pods (**manual scaling**)

kubectl scale deployment/nginx-deployment --replicas=10

Assume we have HPA(**Horizontal POD Auto-scaling**) enabled in cluster.

So if we want to give the min and max count of replicas

kubectl autoscale deployment/nginx-deployment --min=10 --max=15

Now if we want to assign cpu limit for it

kubectl autoscale deployment/nginx-deployment --min=10 --max=15 --cpu- percent=80

Manual scaling means we can define the replicas (--repicas=5)

HPA automatically scale the replicas based on cpu metrics

Now run kubectl get rs : you will get the new deployment.

kubectl get pod

You see that the number of old replicas is 3, and new replicas is 0.

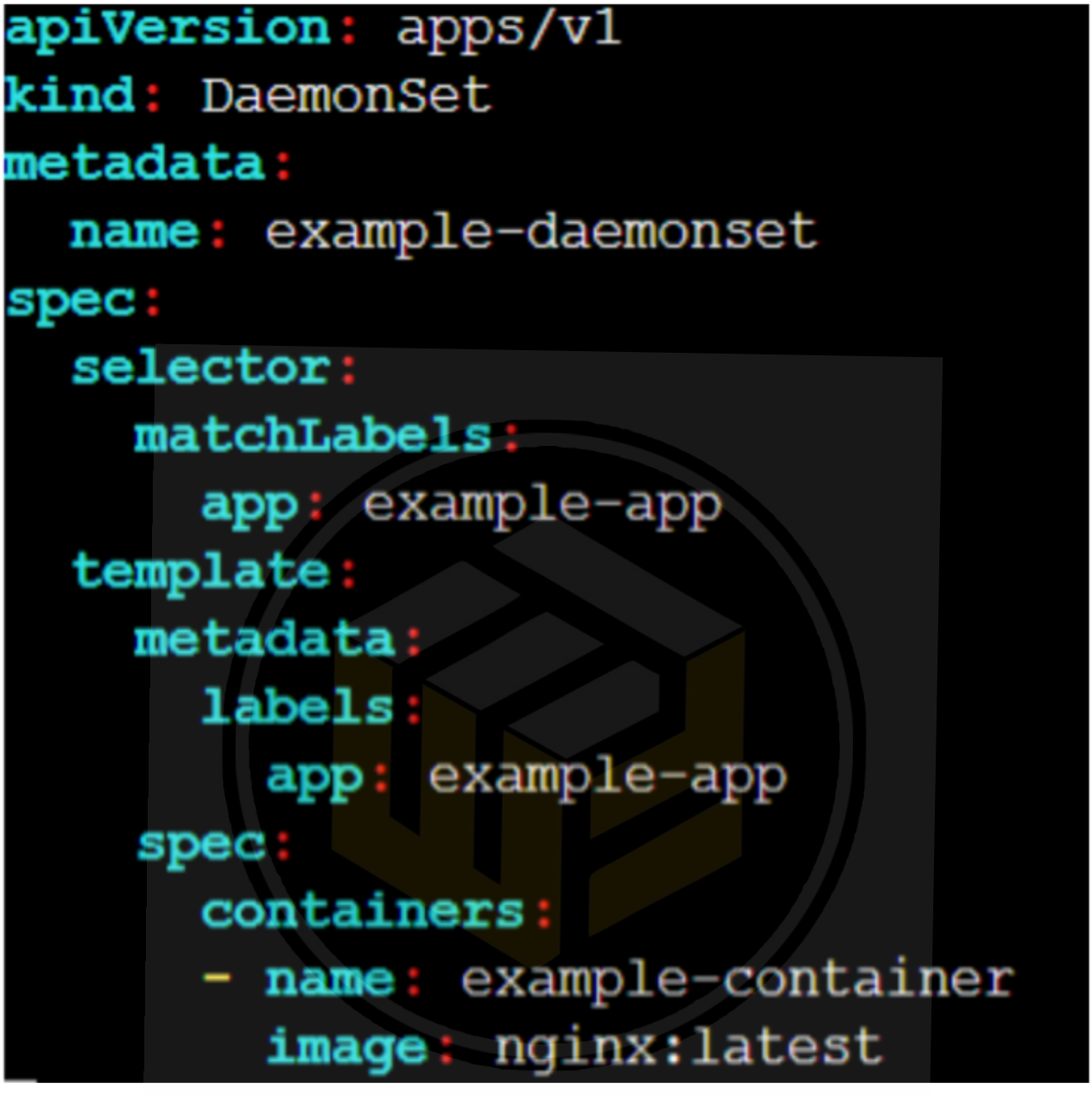
Next time you want to update these Pods, you only need to update the Deployment's Pod template again.

While updating pods, 75% will be available and 25% will be unavailable during updates

Lets assume, if we have 4 pods, while trying to update them atleast 3 of them are available and 1 should be updated in the mean time.

## DAEMON-SET:

- It is used to run a copy a pod to each worker node.
- It is used to perform some tasks like monitoring, Log collection etc.. that need to run on every node of the cluster.
- A DaemonSet ensures that all eligible nodes run a copy of a Pod
- When you create daemon set k8s schedules one copy of pod in each node.
- If we added new node it will copy the pod automatically to new node.
- If I remove on node from cluster the pod in that node also removed.
- Deleting a DaemonSet will clean up the Pods it created.
- Daemon-set does not contains replicas, because bydefault it will create only 1 pod in each node.
- Daemon sets will not used for deployments on real-time, it is only to schedule pods.




## NAMESPACE:

- Namespaces are used to group the components like pods, services, and deployments.
- This can be useful for separating environments, such as development, staging, and production, or for separating different teams or applications.
- In real-time all the frontend pods are mapped to one namespace and backend pods are mapped to another namespace.
- It represents the cluster inside the cluster.
- You can have multiple namespaces within one Kubernetes cluster, and they are all logically isolated from one another.
- Namespaces provide a logical separation of cluster resources between multiple users, teams, projects, and even customers.
- Within the same Namespace, Pod to Pod communication.

- Namespaces provide a logical separation between the environments (Dev, QA, Test, and Prod) with many users, or projects.
- Namespaces are only hidden from each other but are not fully isolated from each other.
- One service in a Namespace can talk to another service in another Namespace if the target and sources are used with the full name which includes service/object name followed by Namespace.
- The name of resources within one namespace must be unique.
- When you delete a namespace all the resources will gets deleted.

kubectl get namespaces : To get the name space

- **Default** : when we create resources like pod, service, deployements all will gets stored in default namespace
- **kube-public** : The namespace for resources that are publicly available by all.
- **kube-system**: The namespace for objects created by the Kubernetes system.
- **kube-node-lease**: It is used for the lease objects associated with each node that improves the performance of the node heartbeats as the cluster scales

## COMMANDS FOR NAMESPACE:

- kubectl get ns : *used to get namespaces*
- kubectl create ns mustafa : *used to create new namespace*
- kubectl config set-context --current --namespace=mustafa : *to check to namespace*
- kubectl config view --minify | grep namespace : *to verify the name space.*

## CREATE A POD IN NS (IMPERATIVE):

1.Create a namespace - kubectl create namespace mustafa

2.create a pod in ns - kubectl run nginx --image=nginx --namespace mustafa

(or)

kubectl run nginx --image=nginx -n development

3. To check the pods: kubectl get pod -n mustafa

(Or)

kubectl get pod --namespace mustafa

## CREATE A POD IN NS (DECLARATIVE):

```yaml
apiVersion: v1
kind: Pod
metadata:
   name: nginx
   namespace: mustafa
~
```

When you delete a namespace all the resources will gets deleted.

- kubectl get pods -n mustafa : used to get pods from namespace
- kubectl describe pod nginx -n development : describe a pod in namespace
- kubectl delete pod nginx -n development : delete a pod in namespace

*POD FILE*

```yaml
---
apiVersion: v1
kind: Pod
metadata:
  name: frontendnew
  labels:
    app: swiggy
  namespace: mustafa
spec:
  containers:
    - name: cont1
      image: rahamshaik/cycle:latest
      ports:
        - containerPort: 80
~
```

*NODEPORT FILE*

```yaml
---
apiVersion: v1
kind: Service
metadata:
  name: frontendnew
  namespace: mustafa
spec:
  type: NodePort
  selector:
    app: swiggy
  ports:
    - port: 80
      targetPort: 80
      nodePort: 30003
```

Before going to deploy this file we need to create mustafa namespace.
Here i just mentioned namespace in normal manifest file

## CONFIG MAPS:

- ConfigMap is used to store the configuration data in key-value pairs within Kubernetes.
- But the data should be non confidential data.
- This is one of the ways to decouple the configuration from the application to get rid of hardcoded values.
- Also, if you observe some important values keep changing according to the environments such as development, testing, production, etc ConfigMap helps to fix this issue to decouple the configurations
- So we can set the configuration of data of application separately
- But it does not provider security and encryption. If we want to provide encryption use secrets in Kubernetes.
- Limit of config map data in only 1 MB (we cannot store more than that)
- But if we want to store a large amount of data in config maps we have to mount a volume or use a separate database or file service.

## USE CASES IN CONFIG MAPS:

- **Configure application settings**: By using this config maps, we can store the configuration data that helps to run our application like database connections and environment variables
- **Configuring a pod or container**: It is used to send a configuration data to a pod or container at runtime like CLI or files.
- **Sharing configuration data across multiple resources:** By using this configuration data multiple resources can access, such as a common configuration file for different pods or services.
- **We can store the data:** By using this config maps, we can store the data like IP address, URL's and DNS etc...

## SOME POINTS ABOUT CONFIG MAPS:

- Creating the configMap is the first process which can be done by commands only or a YAML file.
- After creating the configMap, we use the data in the pod by injecting the pods.
- After injecting the pods, if there is any update in the configuration we can modify the configMap, and the changes will be reflected in the injected pod.

## CREATING CONFIGMAPS FROM LITERAL:

we have created the configMap through — from-literal which means you just need to provide the key value instead of providing the file with key-value pair data.

kubectl create cm my-first-cm --from-literal=Course=DevOps --from-literal=Cloud=AWS --from-literal=Trainer=Mustafa

- To get Config Maps: kubectl get cm
- To descibe: kubectl describe configmap/configmap-name

- To delete: kubectl delete configmap/configmap

## CREATE A CONFIG-MAPS FROM FILE:

We have created one file first.conf which has some data, and created the configMap with the help of that file.

cat config-map-data.txt

Name=Mustafa

Course=Python

Duration=60 days

- command to create config map: kubectl create cm cm-name --from-file=filename
- To descibe: kubectl describe configmap/configmap-name
- To delete: kubectl delete configmap/configmap

## CREATE A CONFIG-MAPS FROM ENV FILE:

We have created one environment file first.env which has some data in key-value pairs, and created the configMap with the help of the environment file.

cat one.env

Tool=Kubernetes

Topic=Config Maps

Course=DevOps

- command to create config map: kubectl create cm cm-name --from-env-file=one.env
- To descibe: kubectl describe configmap/one.env
- To delete: kubectl delete configmap/one.env

## CREATE A FOLDER FOR CONFIG-MAPS:

We have created multiple files in a directory with different extensions that have different types of data and created the configMap for the entire directory.

mkdir folder1

cat folder1/file1.txt

key1=value1

key2=23

cat folder2/file2.txt

key1=value1

key2=23

command to create config map: kubectl create cm mummy --from-file=folder1/

## CREATE A YAML FOR CONFIG-MAPS:

The imperative way is not very good if you have to repeat the same tasks again and again. Now, we will look at how to create configMap through the YAML file.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-config
data:
  DATABASE_URL: "mysql://db.example.com:3306/mydb"
  API_KEY: "your-api-key"
```

command to create config map: kubectl create -f one.yml

## INJECTING CM INTO THE POD WITH SPECIFIC KEY PAIRS:

we have created four types of configMaps but here, we will learn how to use those configMaps by injecting configMaps into the pods.

- Create a file for pod and execute it
- Inside the pod.yml we have to declare any valuefrom our configmap
- After creating the pod, enter into the pod using kubectl exec -it pod_name /bin/bash
- Once you entered into pod give a command env
- That will print the list of environments.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-1
spec:
  containers:
    - image: nginx
      name: cont-1
      env:
       - name: FirstValue
         valueFrom:
           configMapKeyRef:
             key: Name
             name: babu
```

## INJECTING MULTIPLE CMs WITH SPECIFIC AND MULTIPLE VALUES:

- Lets add multiple key pairs from different files.
- Create a file for pod and execute it
- Inside the pod.yml we have to declare any value from any configmaps
- After creating the pod, enter into the pod using kubectl exec -it pod_name /bin/bash
- Once you entered into pod give a command env
- That will print the list of environments.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-1
spec:
  containers:
    - image: nginx
      name: cont-1
      env:
       - name: FirstValue
         valueFrom:
           configMapKeyRef:
             key: Name
             name: Config-Map-1
       - name: SecondValue
         valueFrom:
           configMapKeyRef:
             key: Course
             name: Config-Map-2

       - name: ThirdValue
         valueFrom:
           configMapKeyRef:
             key: Duration
             name: Config-Map-3
```

## INJECTING THE ENVIRONMENT FILE CM's TO POD:

- Create a file for pod and execute it
- Inside the pod.yml we have to declare a configmap name which was created with env file
- After creating the pod, enter into the pod using kubectl exec -it pod_name /bin/bash
- Once you entered into pod give a command env
- That will print the list of environments.


## INJECTING CM IN THE POD WITH THE ENTIRE PROPER FILE:

- Create a file for pod and execute it
- Inside the pod.yml we have to declare a configmap name which was created with normal file
- After creating the pod, enter into the pod using kubectl exec -it pod_name /bin/bash
- Once you entered into pod give a command ls
- You will find a folder called env-values.
- In side the folder you will get file names as keys
- open the file using cat then we will get labels

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-1
spec:
  containers:
    - image: nginx
      name: cont-1
      volumeMounts:
      - name: my-volume
        mountPath: "/env-values"
        readOnly: true
  volumes:
    - name: my-volume
      configMap:
            name: my-cm-1
```

## INJECTING CM AND CREATING A FILE IN THE POD WITH THE SELECTED KEY PAIRS:

- This concept also will works as same as previous topic. But Inside the pod we can declare the filenames by our own inside the env-values folder.
- After creating the pod, enter into the pod using kubectl exec -it pod_name /bin/bash
- Once you entered into pod give a command ls
- You will find a folder called env-values.
- In side the folder you will get file names as file1 and file2.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-1
spec:
  containers:
    - image: nginx
      name: cont-1
      volumeMounts:
      - name: test
        mountPath: "/env-values"
        readOnly: true
  volumes:
    - name: test
      configMap:
            name: babu
            items:
             - key: Name
               path: "file1"
             - key: Occupation
               path: "file2"
```

## SECRETS:

- There are lot of confidential information that needs to be stored on the server such as database usernames, passwords, or API Keys.
- To keep all the important data secure, Kubernetes has a Secrets feature that encrypts the data.
- Secrets can store data up to 1MB which would be enough.
- Secrets can be created via imperative or declarative ways.
- Secrets are stored in the /tmps directory and can be accessible to pods only.
- After creating the Secrets, applications need to use the credentials or database credentials which will be done by injecting with the pods.

## CREATING SECRET FROM LITERAL:

we have created the Secrets through --from-literal which means you just need to provide the key value instead of providing the file with key-value pair data.

you can see the key and encrypted value because Kubernetes encrypts the secrets.

kubectl create secret generic my-secret --from-literal=username=sm7243

- To get Config Maps: kubectl get secret
- To descibe: kubectl describe secret/my-secret
- To Get in yaml : kubectl get secret my-secret -o yaml

- To delete: kubectl delete secret/my-secret

## SECRETS FROM FILE:

We have created one file first.conf which has some data, and created the Secrets with the help of that file.

cat first.conf

username=sm7234

password=admin@123

kubectl create secret generic secret-from-file --from-file=first.conf

- To get Config Maps: kubectl get secret
- To descibe: kubectl describe secret/secret-from-file
- To Get in yaml : kubec tl get secret secret-from-file -o yaml
- To delete: kubectl delete secret/secret-from-file

## SECRETS FROM ENV-FILE:

We have created one environment file first.env which has some data in key-value pairs, and created the Secrets with the help of the environment file.

cat mustafa.env

Name=mustafa

Place=Hyderabad

Compamy=TCS

kubectl create secret generic secret-from-env --from-env-file=mustafa.env

- To get secrets: kubectl get secret
- To descibe: kubectl describe secret/secret-fro m-env
- To Get in yaml : kubec tl get secret secret-fro m-env -o yaml
- To delete: kubectl delete secret/secret-fro m-env

## CREATE A FOLDER FOR SECRETS:

We have created multiple files in a directory with a different extension that has different types of data and created the Secrets for the entire directory.

mkdir folder1

cat folder1/file1.txt

Name=Mustafa

Place=Hyderabad=23

cat folder2/file2.txt

database=mysq1

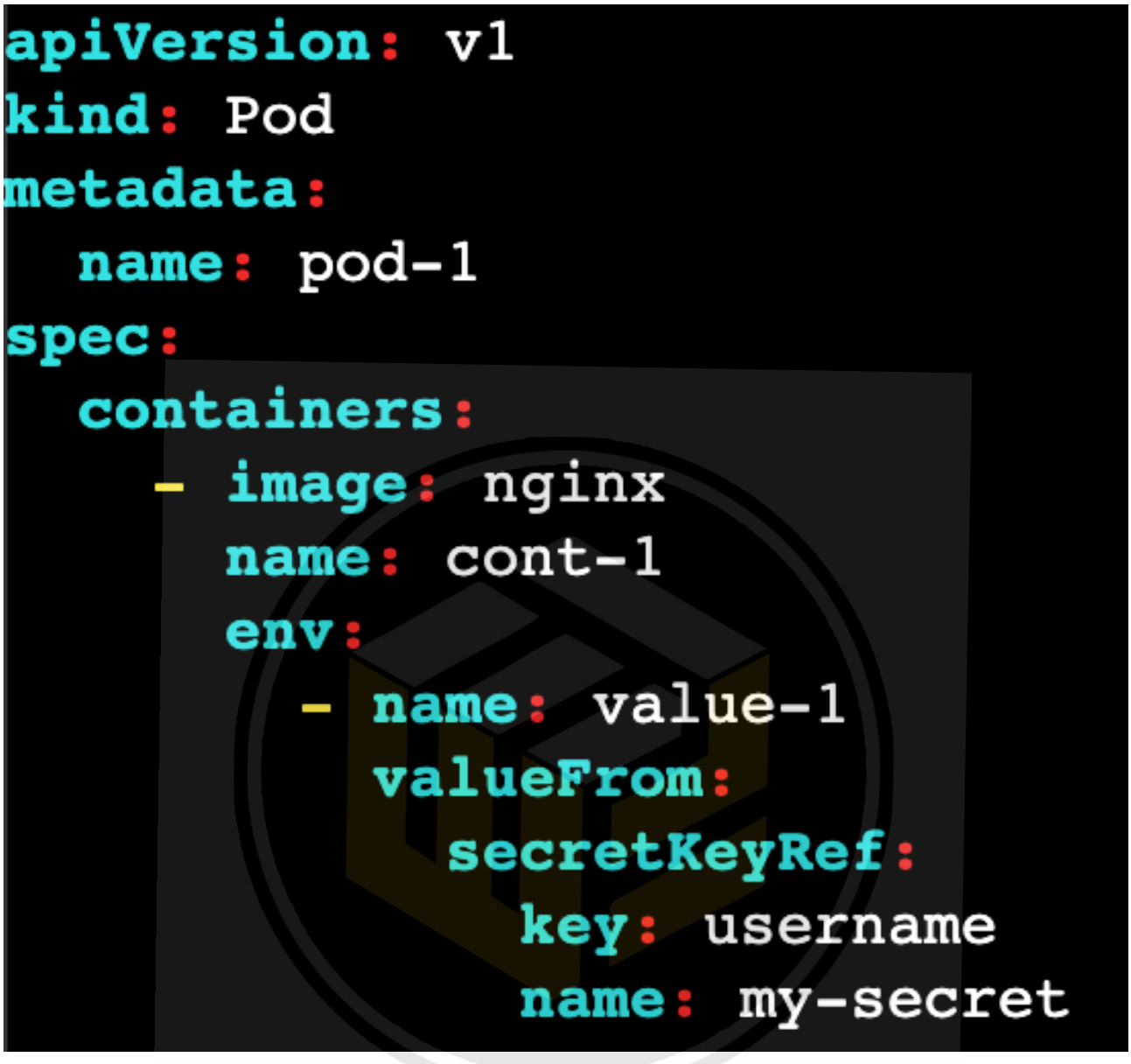
password=mypassword

command to create secret: kubectl create secret generic secret-from-folder --from-file=folder1/

To Get in yaml : kubectl get secret secret-from-env -o yaml

## INJECTING SECRET WITH A POD FOR A PARTICULAR KEY PAIRS:

- we have created four types of Secrets but here, we will learn how to use those Secrets by injecting Secrets to the pods.
- Create a file for pod and execute it
- Inside the pod.yml we have to declare any value from our secret
- After creating the pod, enter into the pod using kubectl exec -it pod_name /bin/bash
- Once you entered into pod give a command env
- That will print the list of environments.




## INJECTING THE ENVIRONMENT FILE SECRET TO POD:

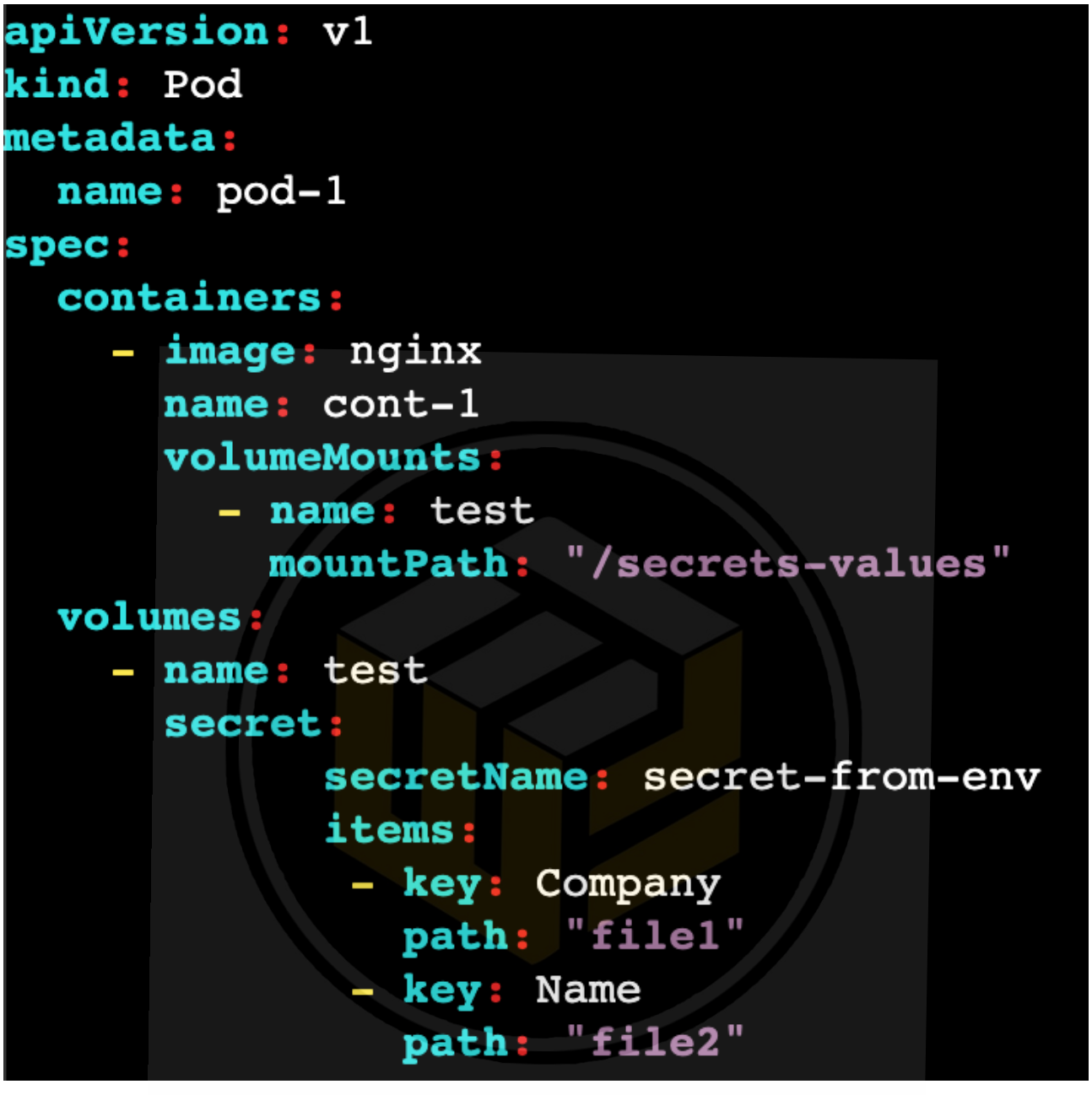
- Create a file for pod and execute it
- Inside the pod.yml we have to declare a secret name which was created with env file
- After creating the pod, enter into the pod using **kubectl exec -it pod_name /bin/bash**
- Once you entered into pod give a command **ls**
- You will find a folder called secret-values.
- In side the folder you will get file names as keys
- open the file using **cat** then we will get labels

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-1
spec:
  containers:
    - image: nginx
      name: cont-1
      volumeMounts:
        - name: my-volume
          mountPath: "/secrets-values"
  volumes:
    - name: my-volume
      secret:
            secretName: secret-from-env
```

## INJECTING SECRET AND CREATING A FILE IN THE POD WITH THE SELECTED KEY PAIRS:

This concept also will works as same as previous topic. But Inside the pod we can declare the filenames by our own inside the env-values folder.

- After creating the pod, enter into the pod using kubectl exec -it pod_name /bin/bash
- Once you entered into pod give a command ls
- You will find a folder called secret-values.
- In side the folder you will get file names as file1 and file2.


apiVersion: v1
kind: Pod
metadata:
name: pod-1
spec:
containers:
- image: nginx
name: cont-1
volumeMounts:
- name: test
mountPath: "/secrets-values"
volumes:
- name: test
secret:
secretName: secret-from-env
items:
- key: Company
path: "file1"
- key: Name
path: "file2"

There are 2 possible ways to store the data in config maps

| DATA | BINARY DATA |
| --- | --- |
| • Stores data in plain text<br>• used to store configuration, script and other text based data<br>• Encoded in UFT-8 format<br>• Decoding in not necessary | • Stores data in binary data<br>• stores non-text data like images, videos or binary files<br>• Encoded in BASE-64 format<br>• Here you need to decode data from BASE-64 format before using |

## K8'S VOLUME:

- Basically these K8's will works on short living data. So lets unveil the power of volumes like EmptyDir, HostPath, PV & PVC.
- The data is a very important thing for an application. In K8's, data is kept for a short time in the applications in the pods/containers. By default the data will no longer available. To overcome this we will use Kubernetes Volumes.
- But before going into the types of Volumes. Let's understand some facts about pods and containers' short live data.
- The volumes reside inside the Pod which stores the data of all containers in that pod.
- If the container gets deleted, then the data will persist and it will be available for the new container which was created recently.
- Multiple containers within a pod can share one volume because the volume is attached to the pod.
- If the Pod gets deleted, then the volume will also get deleted which leads to a loss of data for all containers permanently.
- After deleting the pod, the new pod will be created with volume but this time volumes don't have any previous data or any data.

## TYPES OF VOLUMES

1. EmptyDir
2. HostPath
3. Persistent Volume
4. Persistent Volume Claim(PVC)

## EMPTY DIR:

- This volume is used to share the volumes between multiple containers within a pod instead of the host machine or any Master/Worker Node.
- EmptyDir volume is created when the pod is created and it exists as long as a pod.
- There is no data available in the EmptyDir volume type when it is created for the first.
- Containers within the pod can access the other containers' data. However, the mount path can be different for each container.
- If the Containers get crashed then, the data will still persist and can be accessible by other or newly created containers.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-1
spec:
  containers:
  - name: container1
    image: ubuntu
    command: ["/bin/bash", "-c", "while true; do echo Welcome to DevOps classes; sleep 5 ; done"]
    volumeMounts:
      - name: myvolume
        mountPath: "/tmp/container1"
  - name: container2
    image: centos
    command: ["/bin/bash", "-c", "while true; do echo This is Kubernetes classes; sleep 5 ; done"]
    volumeMounts:
      - name: myvolume
        mountPath: "/tmp/container2"
  volumes:
  - name: myvolume
    emptyDir: {}
```

In the above file, it will creates 2 containers inside a pod. if you create a file in any container, then the file will be available on both the containers.

docker exec -it pod-1 -c container1 /bin/bash

By using the above command create a file in cont-1 and it will be available on cont-2 as well

## 2. HOSTPATH:

- This volume type is the advanced version of the previous volume type EmptyDir.
- In EmptyDir, the data is stored in the volumes that reside inside the Pods only where the host machine doesn't have the data of the pods and containers.
- hostpath volume type helps to access the data of the pods or container volumes from the host machine.
- hostpath replicates the data of the volumes on the host machine and if you make the changes from the host machine then the changes will be reflected to the pods volumes(if attached).

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: hostpath
spec:
  containers:
  - name: container1
    image: ubuntu
    command: ["/bin/bash", "-c", "while true; do echo This is Kubernets class; sleep 5 ; done"]
    volumeMounts:
    - mountPath: "/tmp/cont"
      name: hp-vm
  volumes:
  - name: hp-vm
    hostPath:
      path: /tmp/data
```

In the above file, it will creates a pod. if you create a file in container, by using the command

docker exec -it pod-1 -c container1 /bin/bash

file will gets created. Now even if you delete the pod then another pods will gets created with same data.

Because we are using HostPath which creates local volume in our host machine.

## PERSISTENT VOLUME:

- Persistent means always available.
- Persistent Volume is an advanced version of EmptyDir and hostPath volume types.
- Persistent Volume does not store the data over the local server. It stores the data on the cloud or some other place where the data is highly available.
- In previous volume types, if pods get deleted then the data will be deleted as well. But with the help of Persistent Volume, the data can be shared with other pods or other worker node's pods as well after the deletion of pods.
- PVs are independent of the pod lifecycle, which means they can exist even if no pod is using them.
- With the help of Persistent Volume, the data will be stored on a central location such as EBS, Azure Disks, etc.
- One Persistent Volume is distributed across the entire Kubernetes Cluster. So that, any node or any node's pod can access the data from the volume accordingly.
- In K8S, a PV is a piece of storage in the cluster that has been provisioned by an administrator.
- If you want to use Persistent Volume, then you have to claim that volume with the help of the manifest YAML file.
- When a pod requests storage via a PVC, K8S will search for a suitable PV to satisfy the request.
- If a PV is found that matches the request, the PV is bound to the PVC and the pod can use the storage.
- If no suitable PV is found, K8S then PVC will remain unbound (pending).
- To get the Persistent Volume, you have to claim the volume with the help of PVC.
- When you create a PVC, Kubernetes finds the suitable PV to bind them together.

- After a successful bound to the pod, you can mount it as a volume.
- Once a user finishes its work, then the attached volume gets released and will be used for recycling such as new pod creation for future usage.
- If the pod is terminating due to some issue, the PV will be released but as you know the new pod will be created quickly then the same PV will be attached to the newly created Pod.
- After bounding is done to pod you can mount it as a volume. The pod specifies the amount and type of storage it needs, and the cluster provisions a persistent volume that matches the request. If its not matches then it will be in pending state.

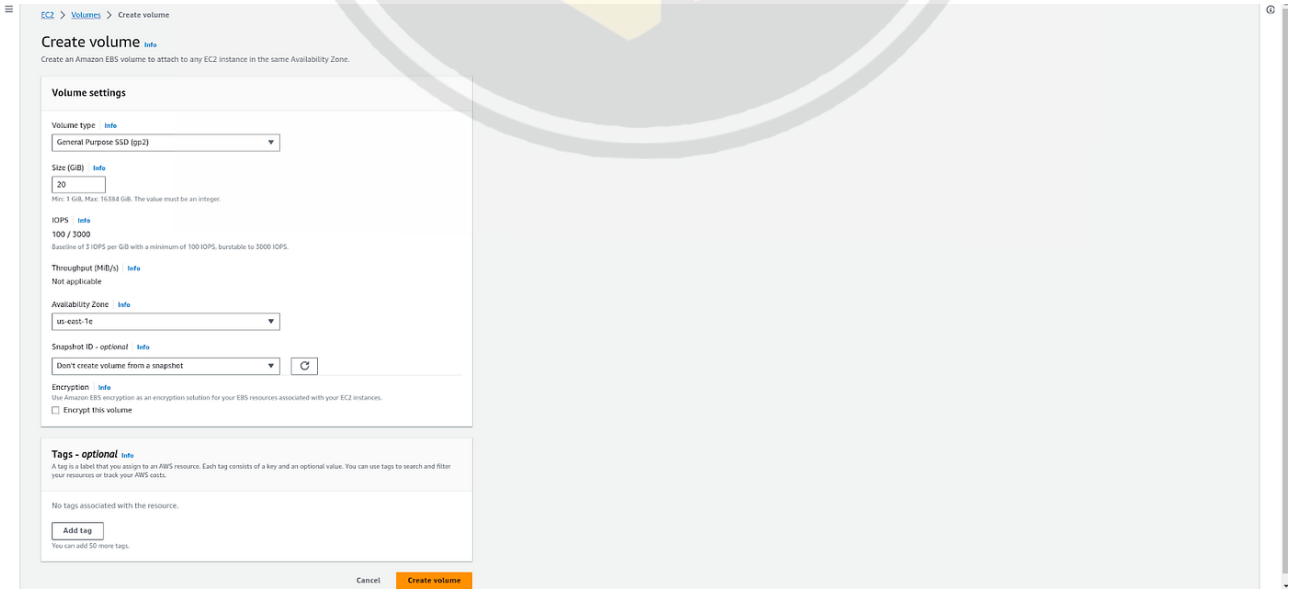
## FACTS ABOUT EBS:

Now, As you know the Persistent Volume will be on Cloud. So, there are some facts and terms and conditions are there for EBS because we are using AWS cloud for our K8 learning. So, let's discuss it as well:

- EBS Volumes keeps the data forever where the emptydir volume did not. If the pods get deleted then, the data will still exist in the EBS volume.
- The nodes on which running pods must be on AWS Cloud only(EC2 Instances).
- Both(EBS Volume & EC2 Instances) must be in the same region and availability zone.
- EBS only supports a single EC2 instance mounting a volume

Create an EBS volume by clicking on 'Create volume'.

Pass the Size for the EBS according to you, and select the Availability zone where your EC2 instance is created, and click on Create volume.




Now, copy the volume ID and paste it into the PV YML file

## PV FILE:

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: myebsvol
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  awsElasticBlockStore:
    volumeID: vol-0a0232b56c59cc682
    fsType: ext4
```

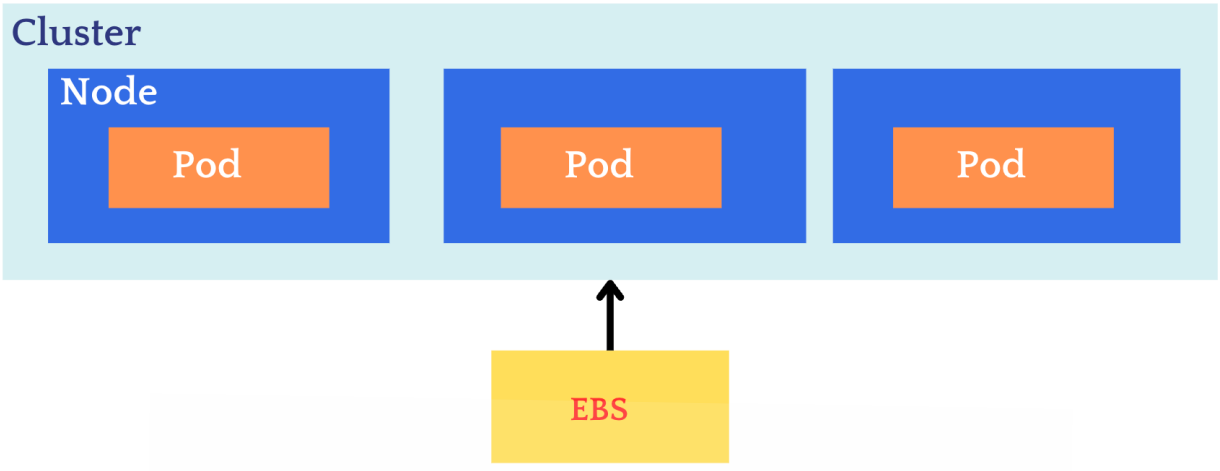
PVC FILE:

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myebsvolclaim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

## DEPLOYMENT FILE:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pvdeploy
spec:
  replicas: 1
  selector:
    matchLabels:
     app: mypv
  template:
    metadata:
      labels:
        app: mypv
    spec:
      containers:
      - name: shell
        image: centos
        command: ["bin/bash", "-c", "sleep 10000"]
        volumeMounts:
        - name: mypd
          mountPath: "/tmp/persistent"
      volumes:
        - name: mypd
          persistentVolumeClaim:
            claimName: myebsvolclaim
```

we have logged in to the created container (kubectl exec -it pod_name -c shell /bin/bash)and created one file with some text.




we have deleted the pod, and then because of replicas the new pod was created quickly. Now, we have logged in to the newly created pod and checked for the file that we created in the previous step, and as you can see the file is present which is expected.

## vim pv.yml

```
apiVersion: v1
kind: PersistentVolume
metadata:
 name: my-pv
spec:
 capacity:
 storage: 10Gi
 accessModes:
 - ReadWriteOnce
 awsElasticBlockStore:
 volumeID: vol-0e5bdf9400dbb7b57
 fsType: ext4
```

## vim pvc.yml

```
apiVersion: v1
```

```
kind: PersistentVolumeClaim
metadata:
 name: my-pvc
spec:
 accessModes:
 - ReadWriteOnce
 resources:
 requests:
 storage: 10Gi
```

COMMANDS TO GET PV, PVC:

```
kubectl apply -f pv.yml
kubectl apply -f pvc.yml
kubectl get pv,pvc
```

## ACCESS MODES IN VOLUMES:

**Access Modes**

access modes determine how many pods can access a Persistent Volume (PV) or a Persistent Volume Claim (PVC) simultaneously. There are several access modes that can be set on a PV or PVC, including:

- **ReadWriteOnce:** This access mode allows a single pod to read and write to the PV or PVC. This is the most common access mode, and it's appropriate for use cases where a single pod needs exclusive access to the storage.
- **ReadOnlyMany:** This access mode allows multiple pods to read from the PV or PVC, but does not allow any of them to write to it. This access mode is useful for cases where many pods need to read the same data, such as when serving a read-only database.
- **ReadWriteMany:** This access mode allows multiple pods to read and write to the PV or PVC simultaneously. This mode is appropriate for use cases where many pods need to read and write to the same data, such as a distributed file system.
- **Execute:** This access mode allows the pod to execute the data on the PV or PVC but not read or write to it. This mode is useful for use cases where the data is meant to be

executed by the pods only, such as application code.

## STATELESS APPLICATION:

- It can't store the data permanently.
- The word STATELESS means no past data.
- It depends on non-persistent data means data is removed when Pod, Node or Cluster is stopped.
- Non-persistent mainly used for logging info (ex: system log, container log etc..)
- In order to avoid this problem, we are using stateful application.
- A stateless application can be deployed as a set of identical replicas, and each replica can handle incoming requests independently without the need to coordinate with other replicas.

## STATEFUL APPLICATION:

Stateful applications are applications that store data and keep tracking it.

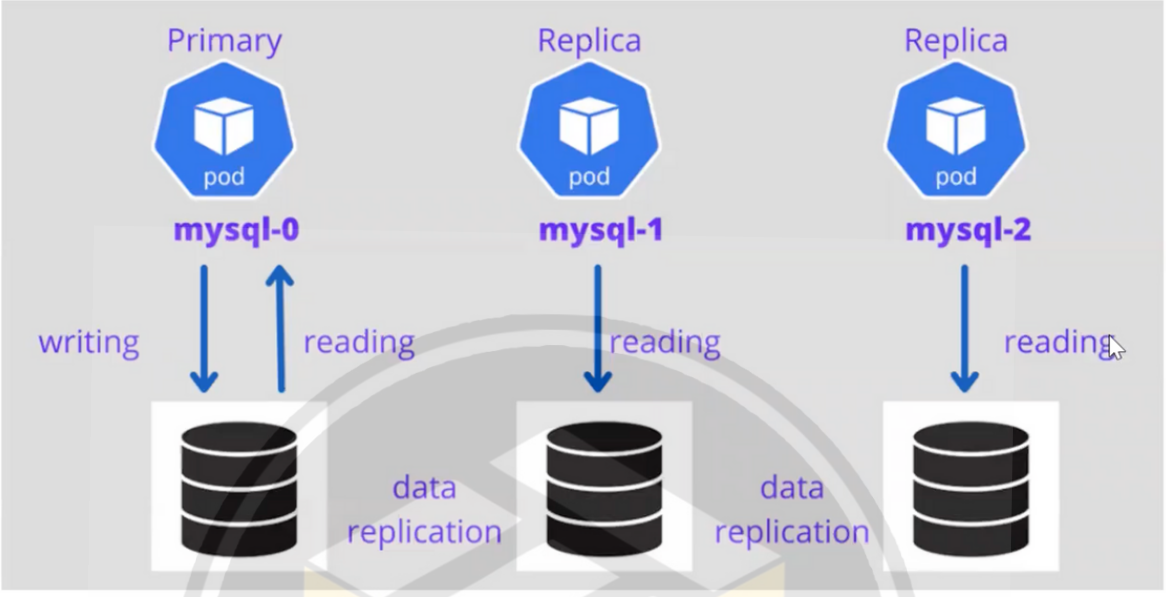
Example of stateful applications:

- All RDS databases (MySQL, SQL)
- Elastic search, Kafka , Mongo DB, Redis etc...
- Any applicaiton that stores data

## STATEFUL SET:

- A StatefulSet is the Kubernetes controller used to run the stateful application as containers (Pods) in the Kubernetes cluster.
- StatefulSets assign a sticky identity-an orginal number starting from zero-to each Pod instead of assigning random IDs for each replica Pod.
- A new Pod is created by cloning the previous Pod's data. If the previous Pod is in the pending state, then the new Pod will not be created.
- If you delete a Pod, it will delete the Pod in reverse order, not in random order.

# StatefulSet




- Lets assume we deployed 3 replicas in a node,
- 1st pod is primary pod, remaining pods are secondary pods.
- Primary pod is having both read and write access
- But secondary pod is having only read access
- If we insert some data in primary pod, we can access the data from any pod i.e.. pod will share the data each other

## DEPLOYMENT VS STATEFUL SET:

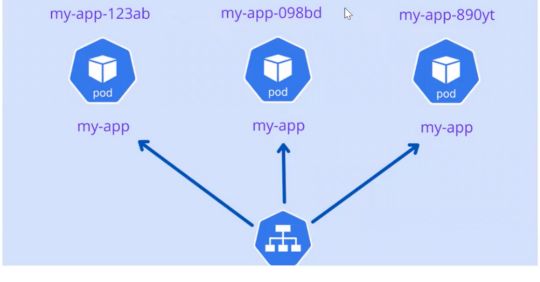
deployment cannot give the numbers or index for resources

but the stateful set will give the sticky identities called index numbers

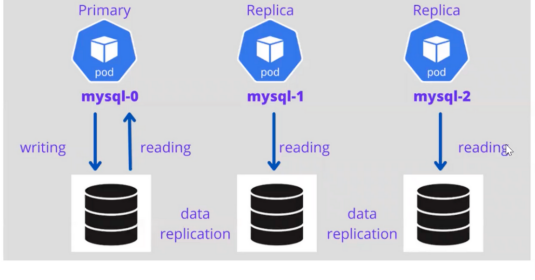
when we delete pods in stateful set last pod will be deleted first

and primary pod deletes last

Deployment




StatefulSet



| DEPLOYMENT | STATEFUL SET |
| --- | --- |
| • It will create POD's with random ID's<br>• Scale down the POD's in random ID's<br>• POD's are stateless POD's<br>• We use this for application deployment | • It will create POD's with sticky ID's<br>• Scale down the POD's in reverse order<br>• POD's are stateful POD's<br>• We use this for database deployment |

## K8'S VOLUMES:

- Basically these K8's will works on short living data. So lets unveil the power of volumes like EmptyDir, HostPath, PV & PVC.
- The data is a very important thing for an application. In K8's, data is kept for a short time in the applications in the pods/containers. By default the data will no longer available. To overcome this we will use Kubernetes Volumes.
- But before going into the types of Volumes. Let's understand some facts about pods and containers' short live data.
- The volumes reside inside the Pod which stores the data of all containers in that pod.
- If the container gets deleted, then the data will persist and it will be available for the new container which was created recently.
- Multiple containers within a pod can share one volume because the volume is attached to the pod.
- If the Pod gets deleted, then the volume will also get deleted which leads to a loss of data for all containers permanently.
- After deleting the pod, the new pod will be created with volume but this time volumes don't have any previous data or any data.

## TYPES OF VOLUMES

1. EmptyDir
2. HostPath
3. Persistent Volume
4. Persistent Volume Claim(PVC)

## EMPTY DIR:

- This volume is used to share the volumes between multiple containers within a pod instead of the host machine or any Master/Worker Node.

- EmptyDir volume is created when the pod is created and it exists as long as a pod.
- There is no data available in the EmptyDir volume type when it is created for the first.
- Containers within the pod can access the other containers' data. However, the mount path can be different for each container.
- If the Containers get crashed then, the data will still persist and can be accessible by other or newly created containers.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-1
spec:
  containers:
  - name: container1
    image: ubuntu
    command: ["/bin/bash", "-c", "while true; do echo Welcome to DevOps classes; sleep 5 ; done"]
    volumeMounts:
      - name: myvolume
        mountPath: "/tmp/container1"
  - name: container2
    image: centos
    command: ["/bin/bash", "-c", "while true; do echo This is Kubernetes classes; sleep 5 ; done"]
    volumeMounts:
      - name: myvolume
        mountPath: "/tmp/container2"
  volumes:
  - name: myvolume
    emptyDir: {}
```

In the above file, it will creates 2 containers inside a pod. if you create a file in any container, then the file will be available on both the containers.

docker exec -it pod-1 -c container1 /bin/bash

By using the above command create a file in cont-1 and it will be available on cont-2 as well

## 2. HOSTPATH:

- This volume type is the advanced version of the previous volume type EmptyDir.
- In EmptyDir, the data is stored in the volumes that reside inside the Pods only where the host machine doesn't have the data of the pods and containers.
- hostpath volume type helps to access the data of the pods or container volumes from the host machine.
- hostpath replicates the data of the volumes on the host machine and if you make the changes from the host machine then the changes will be reflected to the pods volumes(if attached).

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: hostpath
spec:
  containers:
  - name: container1
    image: ubuntu
    command: ["/bin/bash", "-c", "while true; do echo This is Kubernets class; sleep 5 ; done"]
    volumeMounts:
    - mountPath: "/tmp/cont"
      name: hp-vm
  volumes:
  - name: hp-vm
    hostPath:
      path: /tmp/data
```

In the above file, it will creates a pod. if you create a file in container, by using the command

docker exec -it pod-1 -c container1 /bin/bash

file will gets created. Now even if you delete the pod then another pods will gets created with same data.

Because we are using HostPath which creates local volume in our host machine.

## PERSISTENT VOLUME:

- Persistent means always available.
- Persistent Volume is an advanced version of EmptyDir and hostPath volume types.
- Persistent Volume does not store the data over the local server. It stores the data on the cloud or some other place where the data is highly available.
- In previous volume types, if pods get deleted then the data will be deleted as well. But with the help of Persistent Volume, the data can be shared with other pods or other worker node's pods as well after the deletion of pods.
- PVs are independent of the pod lifecycle, which means they can exist even if no pod is using them.
- With the help of Persistent Volume, the data will be stored on a central location such as EBS, Azure Disks, etc.
- One Persistent Volume is distributed across the entire Kubernetes Cluster. So that, any node or any node's pod can access the data from the volume accordingly.
- In K8S, a PV is a piece of storage in the cluster that has been provisioned by an administrator.
- If you want to use Persistent Volume, then you have to claim that volume with the help of the manifest YAML file.
- When a pod requests storage via a PVC, K8S will search for a suitable PV to satisfy the request.
- If a PV is found that matches the request, the PV is bound to the PVC and the pod can use the storage.
- If no suitable PV is found, K8S then PVC will remain unbound (pending).
- To get the Persistent Volume, you have to claim the volume with the help of PVC.
- When you create a PVC, Kubernetes finds the suitable PV to bind them together.

- After a successful bound to the pod, you can mount it as a volume.
- Once a user finishes its work, then the attached volume gets released and will be used for recycling such as new pod creation for future usage.
- If the pod is terminating due to some issue, the PV will be released but as you know the new pod will be created quickly then the same PV will be attached to the newly created Pod.
- After bounding is done to pod you can mount it as a volume. The pod specifies the amount and type of storage it needs, and the cluster provisions a persistent volume that matches the request. If its not matches then it will be in pending state.

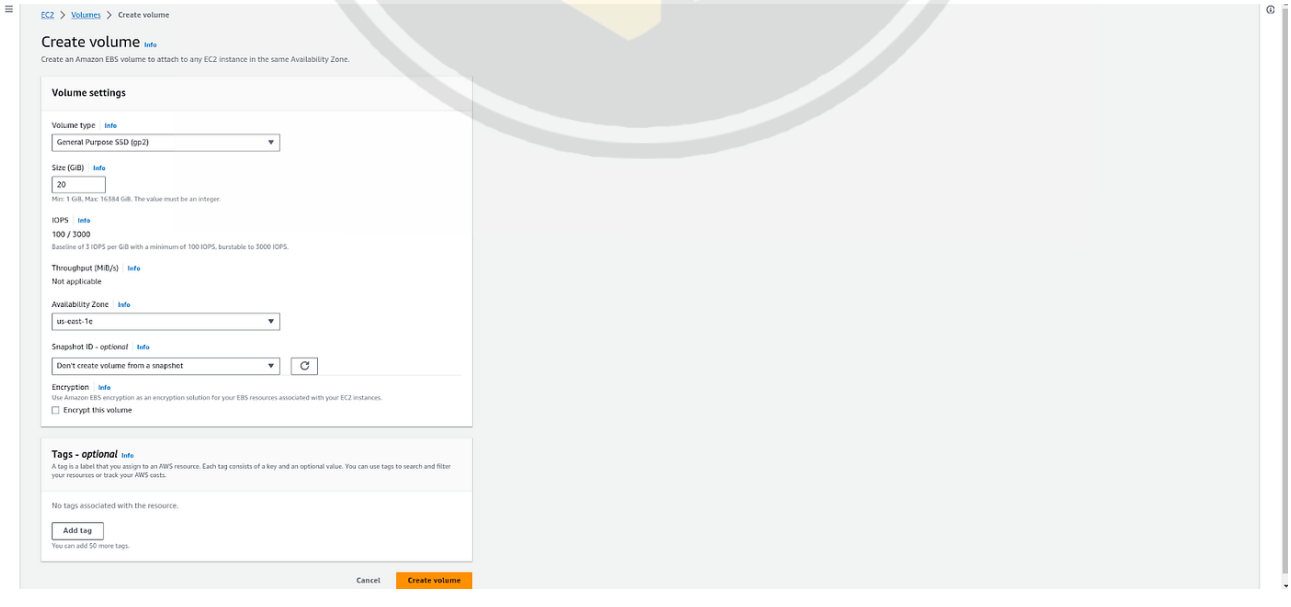
## FACTS ABOUT EBS:

Now, As you know the Persistent Volume will be on Cloud. So, there are some facts and terms and conditions are there for EBS because we are using AWS cloud for our K8 learning. So, let's discuss it as well:

- EBS Volumes keeps the data forever where the emptydir volume did not. If the pods get deleted then, the data will still exist in the EBS volume.
- The nodes on which running pods must be on AWS Cloud only(EC2 Instances).
- Both(EBS Volume & EC2 Instances) must be in the same region and availability zone.
- EBS only supports a single EC2 instance mounting a volume

Create an EBS volume by clicking on 'Create volume'.

Pass the Size for the EBS according to you, and select the Availability zone where your EC2 instance is created, and click on Create volume.




Now, copy the volume ID and paste it into the PV YML file

## PV FILE:

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: myebsvol
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  awsElasticBlockStore:
    volumeID: vol-0a0232b56c59cc682
    fsType: ext4
```

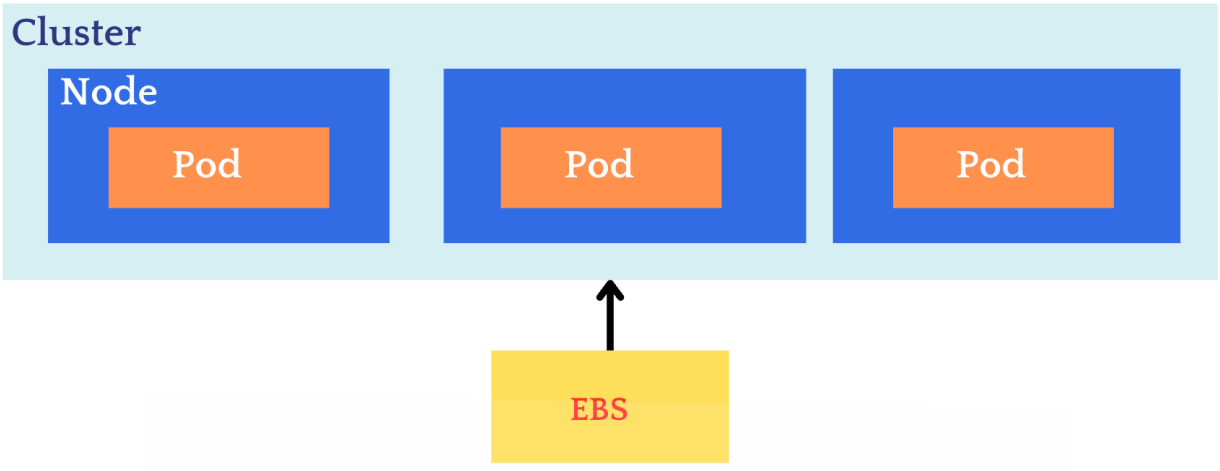
PVC FILE:

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myebsvolclaim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

## DEPLOYMENT FILE:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pvdeploy
spec:
  replicas: 1
  selector:
    matchLabels:
     app: mypv
  template:
    metadata:
      labels:
        app: mypv
    spec:
      containers:
      - name: shell
        image: centos
        command: ["bin/bash", "-c", "sleep 10000"]
        volumeMounts:
        - name: mypd
          mountPath: "/tmp/persistent"
      volumes:
        - name: mypd
          persistentVolumeClaim:
            claimName: myebsvolclaim
```

we have logged in to the created container (kubectl exec -it pod_name -c shell /bin/bash)and created one file with some text.




we have deleted the pod, and then because of replicas the new pod was created quickly. Now, we have logged in to the newly created pod and checked for the file that we created in the previous step, and as you can see the file is present which is expected.

## vim pv.yml

```
apiVersion: v1
kind: PersistentVolume
metadata:
 name: my-pv
spec:
 capacity:
 storage: 10Gi
 accessModes:
 - ReadWriteOnce
 awsElasticBlockStore:
 volumeID: vol-0e5bdf9400dbb7b57
 fsType: ext4
```

## vim pvc.yml

```
apiVersion: v1
```

```
kind: PersistentVolumeClaim
metadata:
 name: my-pvc
spec:
 accessModes:
 - ReadWriteOnce
 resources:
 requests:
 storage: 10Gi
```

COMMANDS TO GET PV, PVC:

```
kubectl apply -f pv.yml
kubectl apply -f pvc.yml
kubectl get pv,pvc
```

## ACCESS MODES IN VOLUMES:

**Access Modes**

access modes determine how many pods can access a Persistent Volume (PV) or a Persistent Volume Claim (PVC) simultaneously. There are several access modes that can be set on a PV or PVC, including:

- **ReadWriteOnce:** This access mode allows a single pod to read and write to the PV or PVC. This is the most common access mode, and it's appropriate for use cases where a single pod needs exclusive access to the storage.
- **ReadOnlyMany:** This access mode allows multiple pods to read from the PV or PVC, but does not allow any of them to write to it. This access mode is useful for cases where many pods need to read the same data, such as when serving a read-only database.
- **ReadWriteMany:** This access mode allows multiple pods to read and write to the PV or PVC simultaneously. This mode is appropriate for use cases where many pods need to read and write to the same data, such as a distributed file system.
- **Execute:** This access mode allows the pod to execute the data on the PV or PVC but not read or write to it. This mode is useful for use cases where the data is meant to be

executed by the pods only, such as application code.

## STATELESS APPLICATION:

- It can't store the data permanently.
- The word STATELESS means no past data.
- It depends on non-persistent data means data is removed when Pod, Node or Cluster is stopped.
- Non-persistent mainly used for logging info (ex: system log, container log etc..)
- In order to avoid this problem, we are using stateful application.
- A stateless application can be deployed as a set of identical replicas, and each replica can handle incoming requests independently without the need to coordinate with other replicas.

## STATEFUL APPLICATION:

Stateful applications are applications that store data and keep tracking it.

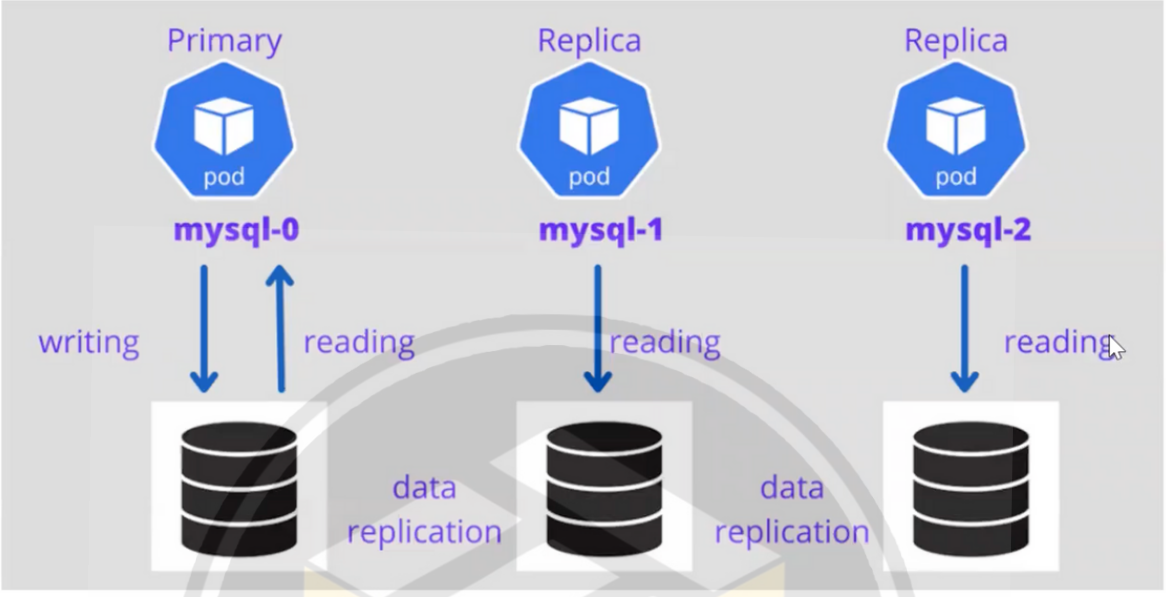
Example of stateful applications:

- All RDS databases (MySQL, SQL)
- Elastic search, Kafka , Mongo DB, Redis etc...
- Any applicaiton that stores data

## STATEFUL SET:

- A StatefulSet is the Kubernetes controller used to run the stateful application as containers (Pods) in the Kubernetes cluster.
- StatefulSets assign a sticky identity-an orginal number starting from zero-to each Pod instead of assigning random IDs for each replica Pod.
- A new Pod is created by cloning the previous Pod's data. If the previous Pod is in the pending state, then the new Pod will not be created.
- If you delete a Pod, it will delete the Pod in reverse order, not in random order.

# StatefulSet



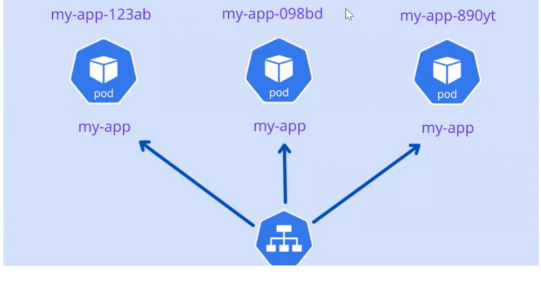

- Lets assume we deployed 3 replicas in a node,
- 1st pod is primary pod, remaining pods are secondary pods.
- Primary pod is having both read and write access
- But secondary pod is having only read access
- If we insert some data in primary pod, we can access the data from any pod i.e.. pod will share the data each other

## DEPLOYMENT VS STATEFUL SET:

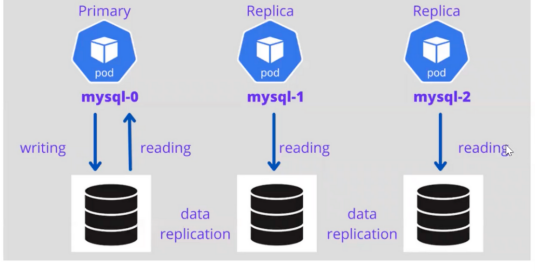
- deployment cannot give the numbers or index for resources but the stateful set will give the sticky identities called index numbers

when we delete pods in stateful set last pod will be deleted first and primary pod deletes last

Deployment




StatefulSet



| DEPLOYMENT | STATEFUL SET |
| --- | --- |
| • It will create POD's with random ID's<br>• Scale down the POD's in random ID's<br>• POD's are stateless POD's<br>• We use this for application deployment | • It will create POD's with sticky ID's<br>• Scale down the POD's in reverse order<br>• POD's are stateful POD's<br>• We use this for database deployment |