kubernetes

Content List

Kubernetes (K8s)

What is kubernetes?

- Kubernetes is an orchestration engine and open-source platform for managing containerized applications.
- Responsibilities include container deployment, scaling and descaling of containers and container load balancing.
- Actually, kubernetes is not a replacement for Docker, but kubernetes can be considered as a replacement for Docker swarm, kubernetes is significantly more complex than docker swarm and requires more work to deploy.
- Born in Google, written in Go/Golang. Donated to CNCF (Cloud Native Computing Foundation) in 2014.

- Kubernetes v1.0 was released on July 21, 2015.
- Current stable release v1.30.0.

Features of Kubernetes

1. **Automated Deployment and Scaling**: Kubernetes allows you to define desired states for your applications and then automates the process of deploying, updating, and scaling them. It can automatically roll out and roll back changes based on defined criteria.

2. **Service Discovery and Load Balancing**: Kubernetes provides built-in service discovery and load balancing, allowing you to expose your applications as network services and balance the load across different instances.

3. **Self-Healing**: Kubernetes continuously monitors the health of your applications and can automatically restart containers that fail, replace containers, or reschedule them on other nodes if necessary.

4. **Horizontal Scaling**: Kubernetes supports horizontal scaling of applications, allowing you to add or remove instances of your applications dynamically based on demand.

5. **Storage Orchestration**: Kubernetes manages persistent storage using various storage systems and providers, enabling you to dynamically provision, manage, and attach storage to your applications.

6. **Configuration Management**: Kubernetes supports dynamic configuration and secrets management through ConfigMaps and Secrets, allowing you to separate application configuration from the code and manage it more efficiently.

7. **Declarative API**: Kubernetes uses a declarative API, allowing you to specify the desired state of your applications and letting Kubernetes handle the complexities of achieving that state.

8. **Namespace Isolation**: Kubernetes supports multi-tenancy through namespaces, which allow you to create isolated environments for different teams or projects within a single cluster.

9. **Resource Management**: Kubernetes provides mechanisms for managing and optimizing the allocation of resources such as CPU and memory across different applications, ensuring efficient utilization.

10. **Built-in Monitoring and Logging**: Kubernetes offers integration with monitoring and logging tools to track the health and performance of your applications.
11. **Role-Based Access Control (RBAC)**: Kubernetes provides fine-grained access control to your cluster resources through RBAC, allowing you to manage user permissions and access levels.
12. **Helm**: While not a core feature of Kubernetes, Helm is a package manager for Kubernetes that simplifies the management of applications and services within the cluster.

Kubernetes Architecture

Kubernetes architecture is designed to manage and orchestrate containerized applications in a cluster of nodes.

It follows a client-server model and consists of several components that work together to achieve this.

Here is an overview of the main components of Kubernetes architecture:

1. **Cluster**:
    - A Kubernetes cluster consists of a set of worker nodes and at least one control plane node (master node).
2. **Control Plane (Master Node)**:
    - The control plane is responsible for managing the cluster and orchestrating the deployment of applications. It includes the following components:
        - **kube-apiserver**: The API server is the entry point to the Kubernetes cluster and exposes the Kubernetes API. It handles all API requests and is the primary interface for interacting with the cluster.
        - **kube-scheduler**: The scheduler is responsible for determining which node a new pod should be scheduled on based on resource availability and other constraints.
        - **kube-controller-manager**: The controller manager runs various controllers that watch for changes in the cluster state and take corrective actions as needed, such as replicating pods or handling node failures.

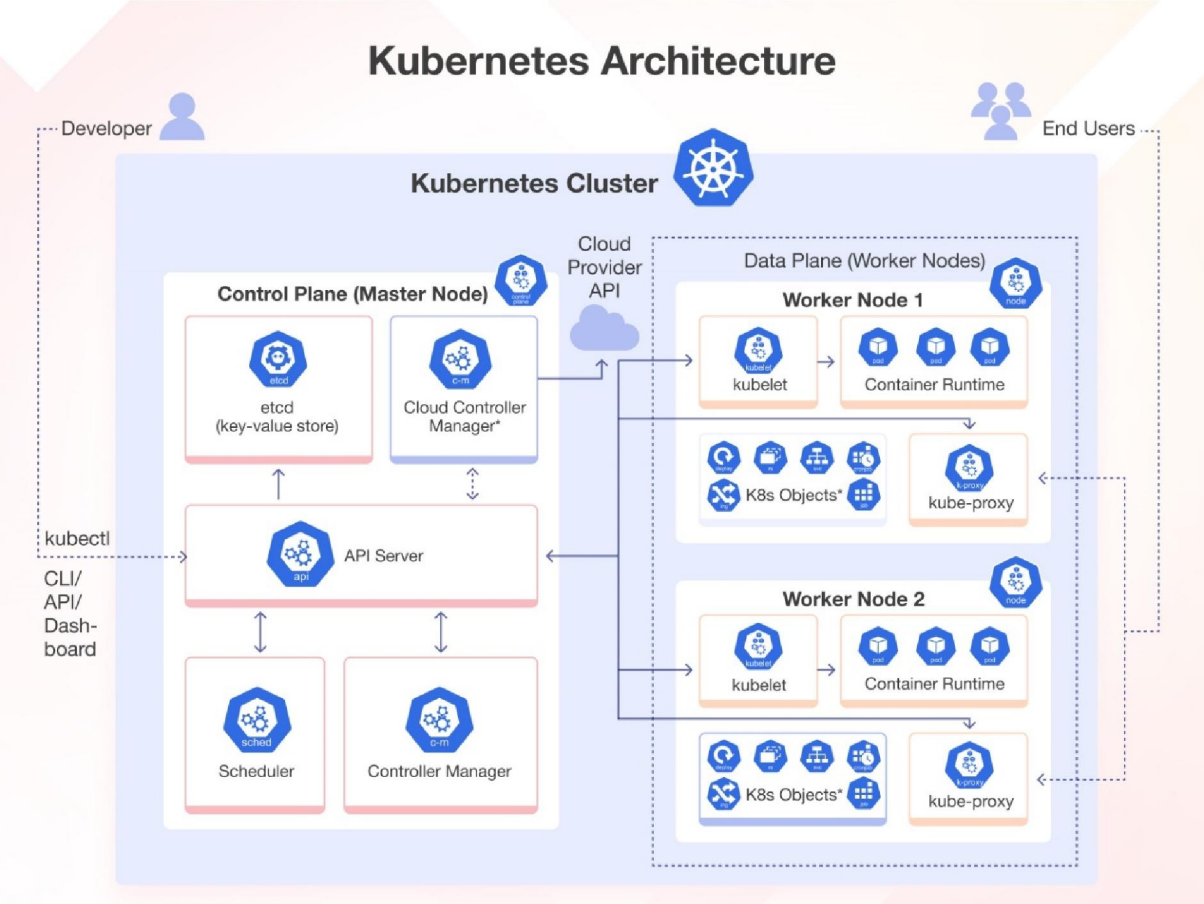
- **cloud-controller-manager**: This component manages interactions with the underlying cloud provider's infrastructure (if applicable) for tasks such as load balancing and persistent storage.
- **etcd**: etcd is a distributed key-value store that stores the cluster's state, including configuration and metadata. It serves as the source of truth for the cluster.

3. **Worker Nodes**:
    - Worker nodes run the applications (pods) and include the following components:
        - **kubelet**: The kubelet is an agent that runs on each node. It ensures that the containers described in pod specs are running and manages their lifecycle.
        - **kube-proxy**: The kube-proxy manages network rules on the node, allowing network communication to and from pods.
        - **Container Runtime**: The container runtime is responsible for running and managing containers on the node. Examples include Docker and containerd.
4. **Pods**:
    - Pods are the smallest deployable units in Kubernetes and represent a group of one or more containers that share network and storage resources.
5. **Services**:
    - Services are a way to expose pods and provide stable network endpoints for them. They can also handle load balancing.
6. **Networking**:
    - Kubernetes provides a flat network model that allows all pods to communicate with each other without Network Address Translation (NAT).
7. **Ingress**:

- Ingress is a resource that manages external access to services, providing load balancing, SSL termination, and path-based routing.

8. **Namespaces**:

   - Namespaces provide a way to divide cluster resources among different teams or projects, allowing for resource isolation and management.



Install Kubernetes

Installing Kubernetes on a single system is mainly for development, testing, and learning purposes. There are several tools available that allow you to set up a single-node Kubernetes cluster quickly and easily. Here is an example of the most popular options:

Minikube

- **Description**: Minikube is a tool that enables you to run a single-node Kubernetes cluster locally on your machine. It's suitable for development and testing.
- **Steps**:

1. **Install Minikube**: Download and install Minikube on your system from the official website.
2. **Install kubectl**: Ensure **kubectl** is installed to interact with the Kubernetes cluster.
3. **Start Minikube**: Run **minikube start** to start the cluster.
4. **Configure kubectl**: Minikube automatically configures **kubectl** to connect to the local cluster.
5. **Verify the installation**: Use commands like **kubectl get nodes** to verify the cluster is running.

Multi-Node Kubernetes clusters

1. Self-managed kubernetes cluster

   kubeadm ➔ We can setup multi-node k8's cluster using kubeadm.

   kubespray ➔ We can setup multi-node k8s cluster using kubespray (Ansible playbooks used internally by kubespray).

   It needs manually management of the control plane and worker nodes.
2. Managed k8s cluster (Cloud services)

   The control plane and its components are managed by the cloud provider. Examples:

   - ✓ EKS ➔ Elastic Kubernetes Service (AWS)
   - ✓ AKS ➔ Azure Kubernetes Service (Azure)
   - ✓ GKE ➔ Google Kubernetes Engine (GCP)
   - ✓ IKE ➔ IBM Kubernetes Engine (IBM Cloud)
3. **KOPs (Kubernetes Operations)**

   It is a software used to create a production ready kubernetes in AWS.

   It is highly available k8s services in cloud like AWS.

   It will leverage cloud services like:

   - ➢ Autoscaling groups
   - ➢ Load balancer
   - ➢ Launch template/configuration
   - ➢ Nodes (WorkerNodes and MasterNodes)
4. **Rancher**

   With Rancher, we can deploy both managed and self-managed k8s

   It serves as a glass to access and manage multiple k8s from the UI- Rancher Dashboard.

Rancher needs authentication and authorization access the different cloud platforms.

Kubernetes Setup Using Kubeadm in AWS EC2 Ubuntu Servers Container-D As Runtime

Prerequisite:

3 - Ubuntu Servers

- ✓ 1 - Manager (4GB RAM , 2 Core) t2.medium
- ✓ 2 - Workers (1 GB, 1 Core) t2.micro
- ✓ **Note**: Open Required Ports in AWS Security Groups. For now, we will open All traffic.

======IMPERATIVE APPROACH TO SETUP KUBERNETES IN AWS ==========

1. Assign hostname & login as 'root' user because the following set of commands need to be executed with 'sudo' permissions

```
sudo hostnamectl set-hostname master
sudo -i
```

```
# run the following below as a script
# This will Install Required packages and apt keys.
#!/bin/bash
sudo apt update -y
sudo apt install -y apt-transport-https
sudo                          curl                          -s
https://packages.cloud.google.com/apt/doc/apt-key.gpg     |
apt-key add -
cat <<EOF >/etc/apt/sources.list.d/kubernetes.list
deb https://apt.kubernetes.io/ kubernetes-xenial main
EOF
apt update -y
#Turn Off Swap Space
swapoff -a
sed -i '/ swap / s/^\(.*\)$/#\1/g' /etc/fstab
```

```
#Install kubeadm, Kubelet And Kubectl containerd
sudo apt-get install -y kubelet containerd kubeadm kubectl
kubernetes-cni
# apt-mark hold will prvent the package from being
authomatically upgraded or removed
sudo apt-mark hold kubelet containerd kubeadm kubectl
kubernetes-cni
#
cat <<EOF | sudo tee /etc/modules-load.d/containerd.conf
overlay
br_netfilter
EOF

# Setup required sysctl params, these persist across
reboots.
cat <<EOF | sudo tee /etc/sysctl.d/99-kubernetes-cri.conf
net.bridge.bridge-nf-call-iptables  = 1
net.ipv4.ip_forward                 = 1
net.bridge.bridge-nf-call-ip6tables = 1
EOF

# Apply sysctl params without reboot
sudo sysctl --system
# Configure containerd:
sudo mkdir -p /etc/containerd
containerd config default | sudo tee
/etc/containerd/config.toml
# Restart containerd:
sudo systemctl restart containerd
# If you get error releated to kubernetes-cni if alreay
exists install with out kubernetes-cni
apt-get install -y kubelet kubeadm kubectl
# Enable and start kubelet service
sudo systemctl daemon-reload
```

```
sudo systemctl start kubelet
sudo systemctl enable kubelet.service
```

2. Exit as root user & execute the below commands as normal ubuntu user

```
sudo su - ubuntu
```

3. Initialised the control plane

```
# Initialize Kubernates master by executing below commond.
sudo kubeadm init

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

4. To verify, if kubectl is working or not, run the following command

```
kubectl get pods -A
```

5. Deploy the network plugin - weave network

```
#deploy the network plugin - weave network
kubectl apply -f https://github.com/weaveworks/weave/releases/download/v2.8.1/weave-daemonset-k8s.yaml
kubectl get pods -A
kubectl get node
```

6. Copy kubeadm join token from the master and execute in Worker Nodes to join to cluster

```
kubeadm join 172.31.10.12:6443 --token cdm6fo.dhbrxyleqe5suy6e \
        --discovery-token-ca-cert-hash sha256:1fc51686afd16c46102c018acb71ef9537c1226e331840e7d401630b96298e7d
```

======SCRIPT TO SETUP KUBERNETES, COMMON FOR MASTER & SLAVES ==========START ==========

```
#!/bin/bash
#i1) Switch to root user [ sudo -i]

sudo hostnamectl set-hostname  node2
sudo -i

#2) Disable swap & add kernel settings

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


#3) Add  kernel settings & Enable IP tables(CNI Prerequisites)

cat <<EOF | sudo tee /etc/modules-load.d/k8s.conf
overlay
br_netfilter
EOF

modprobe overlay
modprobe br_netfilter

cat <<EOF | sudo tee /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-iptables  = 1
net.bridge.bridge-nf-call-ip6tables = 1
net.ipv4.ip_forward                 = 1
EOF

sysctl --system

#4) Install containerd run time

#To install containerd, first install its dependencies.
```

```
apt-get update -y
apt-get install ca-certificates curl gnupg lsb-release -y

#Note: We are not installing Docker Here.Since containerd.io package is
part of docker apt repositories hence we added docker repository & it's
key to download and install containerd.
# Add Docker's official GPG key:
sudo mkdir -p /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --
dearmor -o /etc/apt/keyrings/docker.gpg

#Use follwing command to set up the repository:

echo \
  "deb [arch=$(dpkg --print-architecture) signed-
by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu
\
  $(lsb_release -cs) stable" | sudo tee
/etc/apt/sources.list.d/docker.list > /dev/null

# Install containerd

apt-get update -y
apt-get install containerd.io -y

# Generate default configuration file for containerd

#Note: Containerd uses a configuration file located in
/etc/containerd/config.toml for specifying daemon level options.
#The default configuration can be generated via below command.

containerd config default > /etc/containerd/config.toml

# Run following command to update configure cgroup as systemd for
contianerd.

sed -i 's/SystemdCgroup \= false/SystemdCgroup \= true/g'
/etc/containerd/config.toml

# Restart and enable containerd service

systemctl restart containerd
systemctl enable containerd

#5) Installing kubeadm, kubelet and kubectl

# Update the apt package index and install packages needed to use the
Kubernetes apt repository:

apt-get update
apt-get install -y apt-transport-https ca-certificates curl

# Download the Google Cloud public signing key:

curl -fsSLo /etc/apt/keyrings/kubernetes-archive-keyring.gpg
https://packages.cloud.google.com/apt/doc/apt-key.gpg
```

```
# Add the Kubernetes apt repository:

echo "deb [signed-by=/etc/apt/keyrings/kubernetes-archive-keyring.gpg]
https://apt.kubernetes.io/ kubernetes-xenial main" | sudo tee
/etc/apt/sources.list.d/kubernetes.list

# Update apt package index, install kubelet, kubeadm and kubectl, and pin
their version:

# Add Kubernetes APT keyring and repository
sudo curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.28/deb/Release.key |
sudo gpg --dearmor -o /etc/apt/keyrings/kubernetes-apt-keyring.gpg
echo "deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg]
https://pkgs.k8s.io/core:/stable:/v1.28/deb/ /" | sudo tee
/etc/apt/sources.list.d/kubernetes.list > /dev/null

# Update package index
sudo apt update

# Install Kubernetes packages
sudo apt install -y kubelet kubeadm kubectl

# apt-mark hold will prevent the package from being automatically
upgraded or removed.

apt-mark hold kubelet kubeadm kubectl

# Enable and start kubelet service

systemctl daemon-reload
systemctl start kubelet
systemctl enable kubelet.service

#Copy kubeadm join token from the master and execute in Worker Nodes to
join to cluster
# replace this token with yours

kubeadm join 172.31.63.97:6443 --token hapn6z.ufwr4b82282z3aix \
        --discovery-token-ca-cert-hash
sha256:d45cee841b1694ff440baabf11067b2b7c237bc60b1df5eab4983f4dbae6011c
```

========== **END** ==========

==========In Master Node Only==========

```
#kubeadm command in the master node will generate the config
for the setup #of the entire cluster.

kubeadm init
```

==========Start of Master Node Script==========

```
#!/bin/bash
#i1) Switch to root user [ sudo -i]

sudo hostnamectl set-hostname  node1

#2) Disable swap & add kernel settings

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



#3) Add  kernel settings & Enable IP tables(CNI Prerequisites)

cat <<EOF | sudo tee /etc/modules-load.d/k8s.conf
overlay
br_netfilter
EOF

modprobe overlay
modprobe br_netfilter

cat <<EOF | sudo tee /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-iptables  = 1
net.bridge.bridge-nf-call-ip6tables = 1
net.ipv4.ip_forward                 = 1
EOF

sysctl --system

#4) Install containerd run time
```

```
#To install containerd, first install its dependencies.

apt-get update -y
apt-get install ca-certificates curl gnupg lsb-release -y

#Note: We are not installing Docker Here.Since containerd.io package is part of docker apt repositories hence we added docker repository & it's key to download and install containerd.
# Add Docker’s official GPG key:
sudo mkdir -p /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg

#Use follwing command to set up the repository:

echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

# Install containerd

apt-get update -y
apt-get install containerd.io -y

# Generate default configuration file for containerd

#Note: Containerd uses a configuration file located in /etc/containerd/config.toml for #specifying daemon level options.
#The default configuration can be generated via below command.
```

```
containerd config default > /etc/containerd/config.toml

# Run following command to update configure cgroup as systemd
for contianerd.

sed -i 's/SystemdCgroup \= false/SystemdCgroup \= true/g'
/etc/containerd/config.toml

# Restart and enable containerd service

systemctl restart containerd
systemctl enable containerd

#5) Installing kubeadm, kubelet and kubectl

# Update the apt package index and install packages needed to
use the Kubernetes apt repository:

apt-get update
apt-get install -y apt-transport-https ca-certificates curl

# Download the Google Cloud public signing key:

curl -fsSLo /etc/apt/keyrings/kubernetes-archive-keyring.gpg
https://packages.cloud.google.com/apt/doc/apt-key.gpg

# Add the Kubernetes apt repository:

echo "deb [signed-by=/etc/apt/keyrings/kubernetes-archive-
keyring.gpg] https://apt.kubernetes.io/ kubernetes-xenial
main" | sudo tee /etc/apt/sources.list.d/kubernetes.list
```

```
# Update apt package index, install kubelet, kubeadm and kubectl, and pin their version:

apt-get update
apt-get install -y kubelet kubeadm kubectl

# apt-mark hold will prevent the package from being automatically upgraded or removed.

apt-mark hold kubelet kubeadm kubectl

# Enable and start kubelet service

systemctl daemon-reload
systemctl start kubelet
systemctl enable kubelet.service

# initialise the control plane
kubeadm init

# Configure kubectl as a normal ubuntu user in a different node to manage #to manage the cluster from it and not from master node. This is for #security reasons
sudo su - ubuntu

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

# To verify, if kubectl is working or not, run the following command.
kubectl get pods -o wide -n kube-system
#You will notice from the previous command, that all the pods are running except one: 'core-#dns'. For resolving this we
```

```
will install a # pod network addon like Calico or Weavenet
..etc.
#Note: Install any one network addon don't install both.
Install either weave net or calico.
#To install Weave network plugin/addon run the following
command.

kubectl                          apply                          -f
https://github.com/weaveworks/weave/releases/download/v2.8.1/
weave-daemonset-k8s.yaml
```

==========End of Master Node Script==========

Add Worker Machines to Kubernates Master

Run this script below:

==========Start of Worker Node Script==========

```
#!/bin/bash
#i1) Switch to root user [ sudo -i]

sudo hostnamectl set-hostname  node1

#2) Disable swap & add kernel settings

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


#3) Add  kernel settings & Enable IP tables(CNI Prerequisites)
```

```
cat <<EOF | sudo tee /etc/modules-load.d/k8s.conf
overlay
br_netfilter
EOF

modprobe overlay
modprobe br_netfilter

cat <<EOF | sudo tee /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-iptables  = 1
net.bridge.bridge-nf-call-ip6tables = 1
net.ipv4.ip_forward                 = 1
EOF

sysctl --system

#4) Install containerd run time

#To install containerd, first install its dependencies.

apt-get update -y
apt-get install ca-certificates curl gnupg lsb-release -y

#Note: We are not installing Docker Here.Since containerd.io package is part of docker apt repositories hence we added docker repository & it's key to download and install containerd.
# Add Docker’s official GPG key:
sudo mkdir -p /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg

#Use follwing command to set up the repository:
```

```
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

# Install containerd

apt-get update -y
apt-get install containerd.io -y

# Generate default configuration file for containerd

#Note: Containerd uses a configuration file located in /etc/containerd/config.toml for specifying daemon level options.
#The default configuration can be generated via below command.

containerd config default > /etc/containerd/config.toml

# Run following command to update configure cgroup as systemd for contianerd.

sed -i 's/SystemdCgroup \= false/SystemdCgroup \= true/g' /etc/containerd/config.toml

# Restart and enable containerd service

systemctl restart containerd
systemctl enable containerd

#5) Installing kubeadm, kubelet and kubectl
```

```
# Update the apt package index and install packages needed to use the Kubernetes apt repository:

apt-get update
apt-get install -y apt-transport-https ca-certificates curl

# Download the Google Cloud public signing key:

curl -fsSLo /etc/apt/keyrings/kubernetes-archive-keyring.gpg https://packages.cloud.google.com/apt/doc/apt-key.gpg

# Add the Kubernetes apt repository:

echo "deb [signed-by=/etc/apt/keyrings/kubernetes-archive-keyring.gpg] https://apt.kubernetes.io/ kubernetes-xenial main" | sudo tee /etc/apt/sources.list.d/kubernetes.list

# Update apt package index, install kubelet, kubeadm and kubectl, and pin their version:

apt-get update
apt-get install -y kubelet kubeadm kubectl

# apt-mark hold will prevent the package from being automatically upgraded or removed.

apt-mark hold kubelet kubeadm kubectl

# Enable and start kubelet service

systemctl daemon-reload
systemctl start kubelet
systemctl enable kubelet.service
```

```
#Copy kubeadm join token from the master and execute in Worker
Nodes to join to cluster
# replace this token with yours

kubeadm join 10.0.0.11:6443 --token 03em5o.agjpy9wbj98izbpn \
        --discovery-token-ca-cert-hash
sha256:92a1cf1e452961a550f05eff32f25500fc9bcfd5d5f771689e8ee2
1b6a0da243
#Copy kubeadm join token from master output of kubeadm init
command and execute in #Worker Nodes to join to cluster.

#kubectl commonds has to be executed in master machine.

#Check Nodes
#=============
kubectl get nodes
```

==========End of Worker Node Script==========

Deploy Sample Application

==========

```
kubectl run nginx-demo --image=nginx --port=80

kubectl expose pod nginx-demo --port=80 --target-port=80 --
type=NodePort



#Get Node Port details
#====================
kubectl get services
```

```
#=========================================
install-kubectl.sh
curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
curl -LO https://dl.k8s.io/release/v1.26.0/bin/linux/amd64/kubectl
sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl
chmod +x kubectl
mkdir -p ~/.local/bin
mv ./kubectl ~/.local/bin/kubectl
# and then append (or prepend) ~/.local/bin to $PATH
kubectl version -client
```

How to deploy/run/execute tasks/workloads in kubernetes

In Kubernetes, deploying components (such as pods, services, or other resources) can be done using either an imperative or declarative approach. Each approach offers different ways to manage resources within a Kubernetes cluster.

Imperative Approach

The imperative approach involves issuing direct commands to the Kubernetes API server to create, modify, or delete resources. This approach is often used for interactive or one-time tasks and is beneficial for immediate changes.

Characteristics:

- **Direct Actions**: Commands specify the action to be taken (e.g., create, modify, delete).
- **Immediate Changes**: Changes take effect immediately.
- **Interactive**: Suitable for one-off or immediate tasks.

Examples:

- Create a Pod:

```
apiVersion:v1
kind: Pod
metadata:
    name: <podName>
    namespace: <namespaceName>
    labels:
       key: <value>
       key: <value>
spec:
   containers:
    - name: containerName
     image: imageName
     ports:
         - containerPort:podNumber
```

- Scale a Deployment:

```
kubectl scale deployment my-deployment --replicas=3
```

- Delete a Service:

```
kubectl delete service my-service
```

Declarative Approach

The declarative approach involves specifying the desired state of resources using configuration files (usually in YAML format) and applying them to the Kubernetes cluster. This approach is ideal for managing resources in a consistent, reproducible, and automated way.

Characteristics:

- **Desired State**: You define the desired state of resources in a configuration file.

- **Consistent and Reproducible**: Changes are applied consistently and can be versioned and shared.
- **Automation**: Files can be managed with version control systems and automation tools. Examples:
- Create or Update a Deployment:
    - Define the desired state in a YAML file (**deployment.yaml**):

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-container
        image: nginx
```

    - Apply the configuration:

```
kubectl apply -f deployment.yaml
```

- Delete Resources:
    - Define the resources you want to delete in a YAML file (**delete.yaml**):

```
apiVersion: v1
```

```
kind: Service
metadata:
  name: my-service
```

- Apply the deletion:

```
kubectl delete -f delete.yaml
```

Choosing an Approach

- **Imperative**: Use the imperative approach for immediate, interactive tasks such as debugging or quick fixes.
- **Declarative**: Use the declarative approach for consistent and reproducible deployments, especially in production environments. This approach aligns well with DevOps practices and automation tools.

More about Pods

Again, pods are the smallest and most basic deployable units in kubernetes. A pod represents a single instance of an application and can contain one or more containers that share the same network and storage resources. Pods are versatile and can be managed in various ways depending on the use case and application requirements. Here are the general types of pods in Kubernetes:

1. **Regular Pods:**

- Regular pods are the most common type of pods in Kubernetes and are managed by higher-level controllers such as Deployments, StatefulSets, or DaemonSets.
- These pods typically contain one or more containers and share the same network namespace, allowing them to communicate directly with each other using **localhost**.

2. **Static Pods:**

- Static pods are managed directly by the kubelet on each node and are defined using local YAML configuration files stored in the **/etc/kubernetes/manifests/** directory on each node.
- They are used for critical system components, such as control plane services, and are node-specific, meaning they run only on the node where their configuration resides.

3. **Multi-Container Pods:**

- Pods can contain multiple containers that work together as a single unit. These containers share the same network namespace and can communicate directly with each other using **localhost**.
- Multi-container pods are useful when you want tightly coupled containers that collaborate on a task, such as a main application container and a sidecar container for logging or monitoring.

4. **Debug Pods:**

- Debug pods are temporary pods created for troubleshooting and debugging purposes.
- These pods may contain tools for investigating issues in other running pods, such as network connectivity or application errors.

Pod Management Using Controllers:

- Although not strictly types of pods, the way pods are managed often depends on the controller that oversees them. The main types of controllers include:
    - The **ReplicaSet:** It manages the lifecycle of ReplicaSets, which ensure a specified number of identical pods are running at all times. It automatically creates, deletes, and scales pods to maintain the desired state defined in the ReplicaSet.

```
# Define a ReplicaSet
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: my-replicaset  # Name of the ReplicaSet
spec:
  replicas: 3  # Desired number of replicas (pods)
  # Selector specifies how to identify pods managed
by this ReplicaSet
  selector:
    matchLabels:
      app: my-app
  # Template specifies the pod template for creating
pods
  template:
```

```
    metadata:
      # Labels are essential for matching pods with
the selector
      labels:
        app: my-app
    spec:
      # Define the container(s) for the pods
      containers:
        - name: my-container
          image: nginx   # Docker image for the
container
          ports:
            - containerPort: 80   # Expose port 80
within the container
```

- **Deployments**: Manage replicas of identical pods, enabling rolling updates, rollbacks, and scaling.

```
# Define a Deployment
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-deployment  # Name of the deployment
spec:
  replicas: 3  # Desired number of replicas
  selector:
    # Selector that matches pods based on their
labels
    matchLabels:
      app: my-app
  template:
    # Pod template that defines how the pods should
be created
    metadata:
```

```
      labels:
        app: my-app    # Pod labels matching the
selector
    spec:
      containers:
        - name: my-container
          image: nginx  # Docker image used for the
pod
          ports:
            - containerPort: 80   # Expose port 80
within the container
```

- **StatefulSets**: Manage stateful pods, providing ordered deployment and stable network identities for each pod.

```
# Define a StatefulSet
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: my-statefulset  # Name of the StatefulSet
spec:
  serviceName: "my-service"  # Name of the headless
service used for network identity
  replicas: 3  # Number of pods to create
  selector:
    matchLabels:
      app: my-app  # Selector that matches pods based
on their labels
  template:
    metadata:
      labels:
        app: my-app    # Pod labels matching the
selector
    spec:
```

```
      containers:
        - name: my-container
          image: nginx  # Docker image used for the
pods
          ports:
            - containerPort: 80   # Expose port 80
within the container
```

- **DaemonSets**: Ensure that a copy of a specific pod runs on every node (or selected nodes) in the cluster.

```
# Define a DaemonSet
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: my-daemonset  # Name of the DaemonSet
spec:
  selector:
    matchLabels:
      app: my-app  # Selector that matches pods based
on their labels
  template:
    metadata:
      labels:
        app: my-app    # Pod labels matching the
selector
    spec:
      containers:
        - name: my-container
          image: nginx  # Docker image used for the
pods
          ports:
            - containerPort: 80   # Expose port 80
within the container
```

```

```

- **Jobs and CronJobs**: Manage pods that run specific tasks to completion (Jobs) or at scheduled intervals (CronJobs).

```
# Define a CronJob
apiVersion: batch/v1
kind: CronJob
metadata:
  name: my-cronjob  # Name of the CronJob
spec:
  # Schedule for the job using cron syntax
  schedule: "*/5 * * * *"  # Run the job every 5 minutes
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: my-container
              image: busybox  # Docker image for the pod
              # Command to run within the container
              command: ["echo", "Hi, Kubernetes!"]
          # Restart policy for the CronJob's pod(s); "OnFailure" means the job restarts if it fails
          restartPolicy: OnFailure
```

Pod Lifecycle and Communication:

- Pods have a lifecycle that includes phases such as:
  - Pending
  - Running
  - Terminating.

- Pods can communicate with each other directly using IP addresses and ports or through services that provide stable network endpoints for a group of pods.

kubectl get pod:

This command by default will be search for pods in the default namespace which is created by default.

The above command can be run while explicitly specify the intended namespace with "-n" option in the command as thus:

```
kubectl get pod -n dev
```

Where **"dev"** is the intended namespace.

To completely change the default namespace context, run the command below:

```
kubectl config set-context  --current --namespace=dev
```

The above command will change the default namespace context to "dev" namespace

How do pods communicate with each other?

In Kubernetes, pods can communicate with each other even when they are running on different nodes. This is achieved through the cluster's networking model, which allows every pod to have a unique IP address, and through services, which provide a stable network endpoint for accessing a group of pods. Below are explanations and code examples demonstrating how communication between different pods works in a Kubernetes cluster.

1. Direct Communication Using Pod IPs:

- **Get Pod IP Addresses**:
  - Use **kubectl get pods -o wide** to list the IP addresses of pods.

```
kubectl get pods -o wide
```

- **Direct Communication**:
    - Once you have the IP addresses of the pods, you can communicate directly between them using the IP and a specific port.
    - For example, if a pod with IP address **10.0.0.1** has a service running on port **8080**, another pod can communicate with it by sending a request to

```
http://10.0.0.1:8080.
```

2. Communication via Services:

- **Create a Service**:
    - Services provide an abstraction layer over a set of pods, allowing other pods to communicate with them using a stable network endpoint.
    - Define a service in a YAML file (**service.yaml**):

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: my-app
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
```

    - Apply the service:

```
kubectl apply -f service.yaml
```

- **Communicate with the Service**:
    - Once the service is created, other pods can communicate with it using its DNS name (e.g., **my-service**) or its Cluster IP.
    - Example command (from within another pod):

```
curl http://my-service
```

3. Ingress for External Communication:

- **Ingress**:
    - Ingress provides a way to expose services externally with features such as load balancing, SSL termination, and path-based routing.
    - Define an Ingress in a YAML file (**ingress.yaml**):

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-ingress
spec:
  rules:
  - http:
      paths:
      - path: /my-app
        pathType: Prefix
        backend:
          service:
            name: my-service
            port:
              number: 80
```

    - Apply the Ingress:

```
kubectl apply -f ingress.yaml
```

- **Accessing a Pod via Ingress**:
    - Once the Ingress is set up, you can access the pods within **my-service** using the Ingress controller's external IP and the specified path.

Considerations:

- **Network Plugins**:
    - Kubernetes uses network plugins (e.g., Calico, Flannel, Weave) to provide networking capabilities. These plugins ensure connectivity and can impact how pods communicate with each other.
- **Network Policies**:
    - You can use Network Policies to control and restrict communication between pods based on specific criteria such as labels, namespaces, or IP ranges.

Some commonly used kubectl commands

Kubernetes provides a powerful command-line tool called **kubectl** that allows you to manage different components within a Kubernetes cluster. Below is a table of common **kubectl** commands for managing various Kubernetes resources, including pods, deployments, services, and more:

Resource	Action	Command
Pods	List all pods	kubectl get pods
	Get details of a specific pod	kubectl describe pod <pod-name>
	Create a pod from a YAML file	kubectl apply -f <pod-file.yaml>
	Delete a pod	kubectl delete pod <pod-name>
	Stream logs from a pod	kubectl logs <pod-name>
	Exec into a pod	kubectl exec -it <pod-name> -- <command>

	Port forward to a pod	kubectl port-forward <pod-name> <local-port>:<pod-port>
Deployments	List all deployments	kubectl get deployments
	Get details of a specific deployment	kubectl describe deployment <deployment-name>
	Create a deployment from a YAML file	kubectl apply -f <deployment-file.yaml>
	Scale a deployment	kubectl scale deployment <deployment-name> --replicas=<count>
	Roll back a deployment	kubectl rollout undo deployment/<deployment-name>
	Monitor deployment status	kubectl rollout status deployment/<deployment-name>
Services	List all services	kubectl get services
	Get details of a specific service	kubectl describe service <service-name>
	Create a service from a YAML file	kubectl apply -f <service-file.yaml>
	Delete a service	kubectl delete service <service-name>
Ingress	List all ingress resources	kubectl get ingress
	Create an ingress from a YAML file	kubectl apply -f <ingress-file.yaml>
	Delete an ingress	kubectl delete ingress <ingress-name>
ConfigMaps	List all config maps	kubectl get configmaps
	Create a config map from a file	kubectl create configmap <configmap-name> --from-file=<filename>
	Create a config map from YAML	kubectl apply -f <configmap-file.yaml>
	Get details of a specific config map	kubectl describe configmap <configmap-name>
	Delete a config map	kubectl delete configmap <configmap-name>
Secrets	List all secrets	kubectl get secrets

	Create a secret from a file	kubectl create secret generic <secret-name> --from-file=<filename>
	Create a secret from YAML	kubectl apply -f <secret-file.yaml>
	Get details of a specific secret	kubectl describe secret <secret-name>
	Delete a secret	kubectl delete secret <secret-name>
Namespaces	List all namespaces	kubectl get namespaces
	Create a namespace	kubectl create namespace <namespace-name>
	Delete a namespace	kubectl delete namespace <namespace-name>
Node Management	List all nodes	kubectl get nodes
	Get details of a specific node	kubectl describe node <node-name>
	Cordon a node	kubectl cordon <node-name>
	Uncordon a node	kubectl uncordon <node-name>
	Drain a node	kubectl drain <node-name> --ignore-daemonsets --delete-local-data
DaemonSets	List all daemonsets	kubectl get daemonsets
	Create a daemonset from YAML	kubectl apply -f <daemonset-file.yaml>
	Delete a daemonset	kubectl delete daemonset <daemonset-name>
StatefulSets	List all statefulsets	kubectl get statefulsets
	Create a statefulset from YAML	kubectl apply -f <statefulset-file.yaml>
	Delete a statefulset	kubectl delete statefulset <statefulset-name>
Jobs	List all jobs	kubectl get jobs
	Create a job from YAML	kubectl apply -f <job-file.yaml>
	Delete a job	kubectl delete job <job-name>
CronJobs	List all cronjobs	kubectl get cronjobs
	Create a cronjob from YAML	kubectl apply -f <cronjob-file.yaml>
	Delete a cronjob	kubectl delete cronjob <cronjob-name>

Differences between ReplicaSets and Replication Controller

ReplicaSet	Replication Controller
Supports equality-based selectors and also set based selectors.	Only support equality-based selectors
Supper set of replication controller	Subset of ReplicaSet

Equality based example:
Key == value(Equality condition)
	Set based example:
	`Key in [ value1, value2, value3]`
	`Selector:`
	`matchLabels: # Equality Based`
	`key: value`
	`matchExpressions: # Set Based`
	`- Key:app`
	`Operator: in`
	`Values:`
	`- javaWebApp`
	`- myApp`
	`-fe`

ReplicaSet Example:

```
Kind: ReplicaSet
apiVersion: apps/v1
metadata:
  name: <RSName>
  labels:
    <key>: <value>
spec:
   selector:
      matchLabels:  # Equality Based
        <key>: <value>
      matchExpressions:
         - key: <key>
           operator: <in / not in>
           values:
             - <value1>
             - <value2>
             - <value3>
   replicas: <RSNumber>
   template:
       metadata:
            name: <podName>
            labels:
              <key>: <values>
            spec:
              containers:
                - name: <containerName>
                  image: <imageName:tag>
                  ports:
                    - containerPort: <ContainerPodNumber>
```

```

```

Example: Deploy a nodeApp in a ReplicaSet

```
kind: ReplicaSet
apiVersion: apps/v1
metadata:
   name: nodeApp
spec:
   replicas: 2
   selector:
       matchLabels:  # Equality Based
          app: node
   template:
       metadata:
            name: nodeWebApp
            labels:
                app: node
            spec:
               imagePullSecrets:
                 - name: dockerhublogin
               containers:
                 - name: nodeApp
                   image: fewaitconsulting/nodejs-fe-app
                   ports:
                     - containerPort: 9981

# service to discover the ReplicaSet
---
apiVersion: v1
kind: Service
metadata:
  namespace: dev
  name: nodeAppsvc
spec:
  selector:
    app: node
    type: nodePort
  ports:
      targetPort: 8080
      nodePort: 31500 # Range of node ports 30000 - 32676
      port: 80
```

NB: When pulling the docker image from a private docker registry, Kubernetes will need to be authenticated so a secret need to be created and its reference include the above file.

```
kubectl create secret docker-registry dockerhublogin \
  --docker-server=docker.io \
  --docker-username=DOCKER_USER \
  --docker-password=DOCKER_PASSWORD \
  --docker-email=DOCKER_EMAIL
```

NB: To access this application on the browser, we need to use the nodePort address together with the server IP address.

DaemonSet:

It is almost the same as ReplicaSet apart of the fact that they don't have number of replicas. DaemonSet automatically schedule a copy of the same pod for all nodes or groups of nodes.

Example: Deploy a log management (logmgt) in a DaemonSet.

```
kind: DaemonSet
apiVersion: apps/v1
metadata:
   name: logmgt
spec:
   selector:
       matchLabels:  # Equality Based
          app: node
   template:
       metadata:
            name: hello
            labels:
                app: hello
            spec:
               imagePullSecrets:
                 - name: dockerhublogin
               containers:
                 - name: hello
                   image: fewaitconsulting/hello
                   ports:
                     - containerPort: 9981

# service to discover the ReplicaSet
---
apiVersion: v1
kind: Service
metadata:
  namespace: dev
  name: nodeAppsvc
spec:
  selector:
    app: hello
    type: nodePort
  ports:
      targetPort: 8080
      nodePort: 31500 # Range of node ports 30000 - 32676
      port: 80
```

NB: The DaemonSet will schedule the application in the other nodes except the master node which is by default tainted i.e. The node doesn't accept application scheduling in it.

A node is tainted if it needs:

- Recommissioning
- Upgrades
- Updating
- Patching

How to taint a node:

```
kubectl taint nodes node1 key1=value1:NoSchedule
```

How to untaint a node:

```
kubectl taint nodes node1 key1=value1:NoSchedule-
```

NB: An application can be scheduled on all the nodes whether tainted or not by adding the tolerations to the daemonSet manifest file. Example: Let us modify the above daemonSet file to do that:

```
kind: DaemonSet
apiVersion: apps/v1
metadata:
   name: logmgt
spec:
   selector:
       matchLabels:  # Equality Based
          app: node
   template:
       metadata:
            name: hello
            labels:
                app: hello
            spec:
               imagePullSecrets:
                 - name: dockerhublogin
               containers:
                 - name: hello
                   image: fewaitconsulting/hello
                   ports:
                     - containerPort: 9981
               tolerations:
                   operator: Exists
                   effect: "NoSchedule"
```

Deployment controller

Deployment will exhibit the following characteristics:

- Deploy a ReplicatSet (RS)
- Update pods (PodTemplateSpec)
- Rollback to older deployment versions
- Scale deployment up or down
- Pause and resume deployment.
- Use the status of the deployment to determine state of replicas
- Clean up older RS that aren't needed anymore

Deployment has two strategies:

1. **Recreate (Blue-Green) Deployment**:
    - In this strategy, a new version of the application is deployed by replacing the existing version entirely.
    - The old version (blue) is shut down, and the new version (green) is started.
    - While simple, it may cause downtime during the switch from the old to the new version.
2. **Rolling Update Deployment**:
    - Rolling updates gradually replace instances of the old version with instances of the new version, one at a time.
    - Pods are replaced gradually, ensuring that the application remains available throughout the update process.
    - Rollback is straightforward if any issues are detected during the update.

Example: Deployment of a hello app in a deployment using Recreate strategy.

```
kind: Deployment
apiVersion: apps/v1
metadata:
   name: hello
spec:
   replicas: 2
   selector:
       matchLabels:  # Equality Based
```

```
          app: hello
   strategy:
       type: recreate
   template:
       metadata:
            name: hellopod
            labels:
                app: hello
            spec:
               imagePullSecrets:
                 - name: dockerhublogin
               containers:
                 - name: helloworld
                   image: fewaitconsulting/hello-app
                   ports:
                     - containerPort: 80

# service to discover the Deployment
---
apiVersion: v1
kind: Service
metadata:
  namespace: dev
  name: hellosvc
spec:
  selector:
    app: hello
    type: nodePort
  ports:
      targetPort: 80
      nodePort: 31000 # Range of node ports 30000 - 32676
      port: 80
```

Example 2: Deployment of javawebapp application using RollingUpdate strategy

```
kind: Deployment
apiVersion: apps/v1
metadata:
   name: javawebapp
spec:
   replicas: 2
   selector:
       matchLabels:  # Equality Based
          app: javawebapp
   strategy:
       type: RollingUpdate
       maxUnavailable: 1
       maxSurge: 1
       minReadySeconds: 30
   template:
       metadata:
            name: javawebapppod
            labels:
```

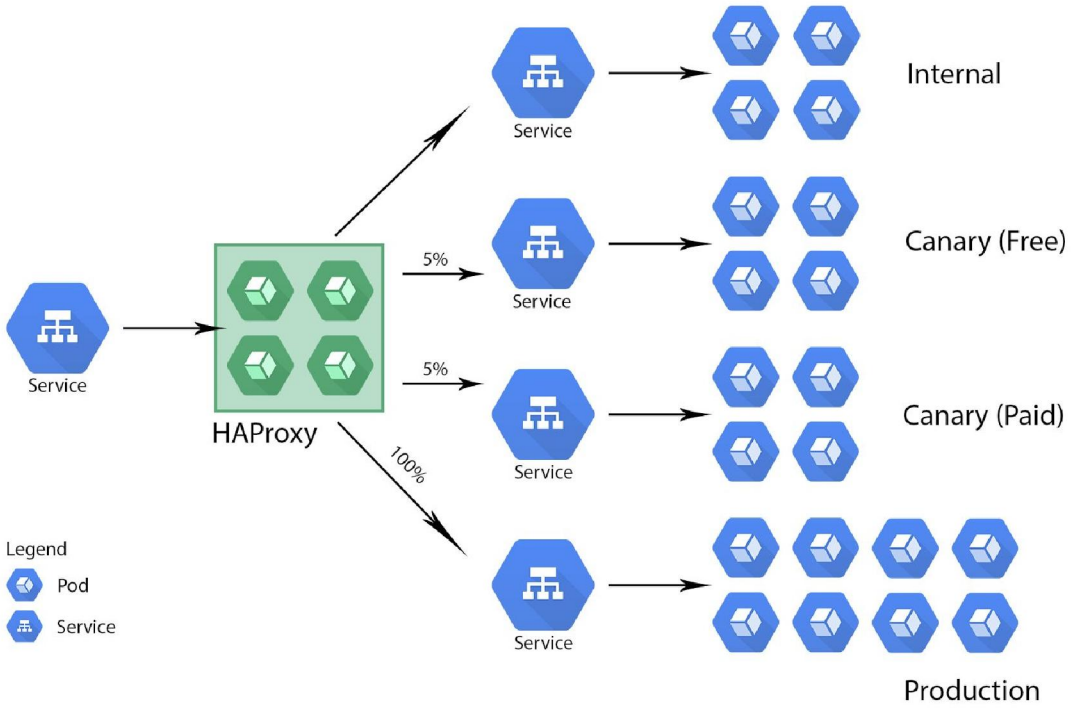
```
                app: javawebapp
            spec:
               imagePullSecrets:
                 - name: dockerhublogin
               containers:
                 - name: helloworld
                   image: fewaitconsulting/java-web-app
                   ports:
                     - containerPort: 8080

# service to discover the Deployment
---
apiVersion: v1
kind: Service
metadata:
  namespace: dev
  name: javawebappsvc
spec:
  selector:
    app: javawebapp
    type: nodePort
  ports:
      targetPort: 80
      nodePort: 31000 # Range of node ports 30000 - 32676
      port: 80
```

Deployment techniques:

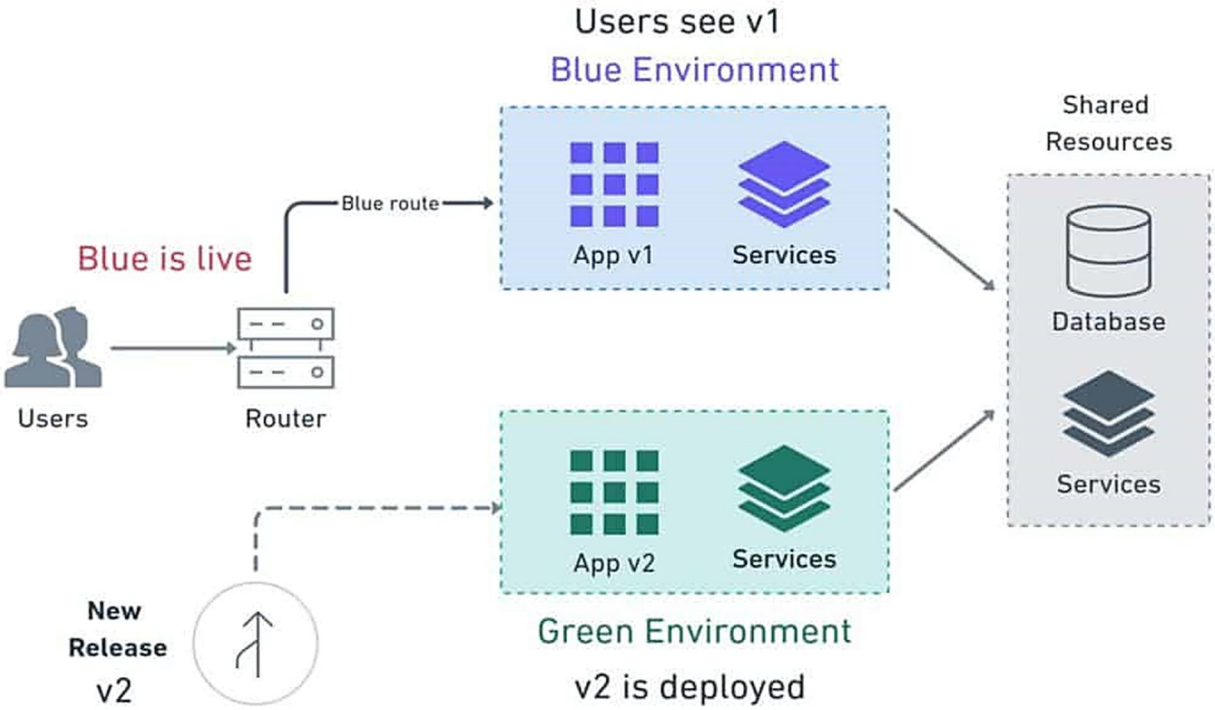
1. **Canary Deployments**:

   - Canary deployments involve gradually rolling out a new version of an application to a subset of users or traffic while monitoring its performance and stability.
   - Traffic routing mechanisms, such as Kubernetes Ingress controllers or service meshes, are used to direct a portion of the traffic to the new version.
   - If the new version performs well, more traffic is gradually shifted to it. Otherwise, the deployment can be rolled back.



2. **Blue-Green Deployments**:

- Blue-green deployments involve maintaining two identical environments (blue and green) running simultaneously, with only one environment serving live traffic at a time.
- New versions of the application are deployed to the inactive environment, and traffic is switched from the active environment to the updated one once the deployment is successful.
- This approach enables zero-downtime deployments and allows for easy rollback in case of issues.



Resource Request and Limit

- Resource requests define the minimum amount of CPU and memory that a container needs to run.
- Kubernetes scheduler uses resource requests to decide where to place pods in the cluster based on available resources.
- Limit is the maximum number of resources that Kubernetes will allow the container to use.

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: nginx
    resources:
      requests:   # Resource requests specify the minimum required
resources for the container
        cpu: "100m"       # 100 millicpu (0.1 CPU)
        memory: "128Mi"   # 128 Mebibytes
```

```
      limits:     # Resource limits specify the maximum allowed resources
for the container
        cpu: "200m"       # 200 millicpu (0.2 CPU)
        memory: "256Mi"   # 256 Mebibytes
```

Pod Auto-Scaling

Pod autoscaling, also known as **Horizontal Pod Autoscaler** (HPA), is a Kubernetes feature that dynamically adjusts the number of pod replicas based on observed resource utilization metrics, such as CPU and memory usage.

The HorizontalPodAutoscaler controller continuously monitors these metrics and scales the number of pod replicas up or down to meet specified target resource utilization levels.

Pod autoscaling helps ensure optimal performance, availability, and cost efficiency by automatically adapting the infrastructure to changing workloads.

Differences between Pod Auto-scaling and AWS Auto-scaling

Aspect	Pod Autoscaling (Kubernetes)	AWS Autoscaling
Scope	Operates at the level of individual pods	Operates at the level of EC2 instances or other AWS resources
Managed Resources	Manages scaling of pod replicas within Kubernetes cluster	Manages scaling of EC2 instances or other AWS resources within AWS environment
Resource Metrics	Scales pods based on resource utilization metrics such as CPU and memory	Can scale resources based on various metrics including CPU utilization, network traffic, and custom CloudWatch metrics
Integration	Integrated with Kubernetes' native scaling mechanisms (e.g., Horizontal Pod Autoscaler)	Integrated with various AWS services including EC2, ECS, and ELB
Environment	Typically used in containerized environments managed by Kubernetes	Used within AWS cloud environment to manage scalability of AWS resources

How to check resource usage in Kubernetes

1. **kubectl top**:
    - Use the **kubectl top** command to view resource usage of pods, nodes, or namespaces.
    - For example, to view CPU and memory usage of pods:

```
kubectl top pods
```

    - To view CPU and memory usage of nodes:

```
kubectl top nodes
```

NB: The two above commands require the Metrics Server to be installed in your cluster.

2. **Dashboard**:
    - Kubernetes Dashboard provides a web-based UI for monitoring cluster resources.
    - You can view resource usage metrics, including CPU and memory, for pods, nodes, and namespaces.
3. **Metrics Server**:
    - The Metrics Server collects resource usage metrics from pods and nodes in the cluster.
    - You can access metrics exposed by the Metrics Server through the Kubernetes API.
4. **Prometheus**:
    - Prometheus is a popular monitoring and alerting tool used with Kubernetes.
    - You can configure Prometheus to scrape metrics from Kubernetes components and applications, including resource usage metrics.
5. **Third-party Monitoring Solutions**:

- There are many third-party monitoring solutions available for Kubernetes, such as Datadog, Grafana, and Prometheus Operator.
- These tools offer advanced monitoring capabilities and customizable dashboards for visualizing resource usage metrics.

Installation of Metric Server

1. **Download Metric Server YAML files**: You can clone the Metric Server repository from GitHub:

```
wget https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml
```

2. **Deploy the Metric Server**: Apply the YAML files to deploy the Metric Server components:

```
kubectl apply -f kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml
```

3. **Verify the deployment**: Check the status of Metric Server components to ensure they are running:

```
kubectl get pods -n kube-system
```

You should see the **metrics-server** deployment running in the **kube-system** namespace.

4. **Verify Metric Server metrics**: Check if Metric Server metrics are available:

```
kubectl top nodes
```

If Metric Server is correctly installed and configured, you should see resource usage metrics for nodes in the cluster.

Deployment with HPA

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hpadeployment
spec:
  selector:
    matchLabels:
      name: hpapod
  template:
    metadata:
      labels:
        name: hpapod
    spec:
      containers:
      - name: hpacontainer
        image: registry.k8s.io/hpa-example
        ports:
        - containerPort: 80
        resources:
          limits:
            cpu: “100m”
            memory:”64Mi”
          requests:
            cpu: 200m
            memory: “256Mi”
---
apiVersion: v1
kind: Service
metadata:
  name: hpaclusterservice
  labels:
    name: hpaservice
spec:
  ports:
  - port: 80
  selector:
    name: hpapod
  type: NodePort
```

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: hpadeploymentautoscaler
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: hpadeployment
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
  - type: Resource
    resource:
      name: memory
```

```
      target:
        type: Utilization
        averageUtilization: 50
```

Example: Deployment of Spring app and MongoDB as pod without volumes

Requirements:

- Stateless application deployment.
- Deployment is the best choice Kubernetes object
- MongoDB:
  - ✓ Use StatefulSet as the choice Kubernetes object since it is stateful application.

```
# mongodb-replicaset.yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: mongodb
  namespace: mongodb
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: mongodb
    spec:
      containers:
      - name: mongodbcontainer
        image: mongo:4.2
        ports:
        - containerPort: 27017
        volumeMounts:
        - name: mongodb-storage
          mountPath: /data/db
        env:
         name: MONGO_INITDB_ROOT_USERNAME: admin
         value: MONGO_INITDB_ROOT_PASSWORD: admin123
         name: MONGO_INITDB_DATABASE: mydatabase
         value: devdb@123
---
# mongodb-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: mongosvc
spec:
  type: ClusterIP
  selector:
```

```
    app: mongo
  ports:
    port: 27017
    targetPort: 27017
---
# springapp-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: springappsvc
spec:
  type: NodePort
  selector:
    app: springapp
  ports:
    port: 80
    targetPort: 8080
```

Kubernetes Volumes:

HostPath Volume:

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: mongodb
spec:
  replicas: 3
  selector:
    matchLabels:
      app: mongo
  template:
    metadata:
      name: myapp
      labels:
        app: mongo
    spec:
      containers:
        volumes:
        - name: mongodbhostvol
          hostpath:
            path: /mongodata
        - name: vol2
          hostPath:
            path: /tmp/data
      containers:
       - name: mongodbcontainer
         image: mongo
         ports:
         - containerPort: 27017
         env:
          - name: MONGO_INITDB_ROOT_USERNAME: admin
            value: MONGO_INITDB_ROOT_PASSWORD: admin123
```

```
          - name: MONGO_INITDB_DATABASE: mydatabase
            value: devdb@123
        volumeMounts:
        - name: mongodbhostvol
          mountPath: /data/db
        - name: vol2
          mountPath: /data/db
```

Configuration of NFS Server

Step 1: Create one server for NFS

NFS = Network File System. It is a distributed file system shared file system.

Update our system's repository index that of the internet through the following apt command as sudo:

```
sudo apt-get update
```

The above command let us install the latest available version o a software through the ubuntu repositories.

Now, run the following command in order to install the NFS kernel server on your system:

```
sudo apt install nfs-kernel-server -y
```

Step 2: Create the export directory

```
sudo mkdir -p /mnt/share
```

NB: As we want all clients to access the directory, we will remove restrictive permissions.

```
sudo chown nobody:nogroup /mnt/share
sudo chmod 777 /mnt/share
```

Step 3: Assign server access to client(s) through NFS export file

```
sudo vi /etc/exports
```

```
# /mnt/share <YourClientIP or clients>(rw,sync,no_subtree_check) OR
# /mnt/share *(rw,sync,no_subtree_check) # To permit all IP addresses
```

```
/mnt/share 192.168.1.0/24(rw,sync,no_subtree_check)
```

Step 4: Export the shared directory

```
sudo exportfs -a
sudo systemctl restart nfs-kernel-server
```

Step 5: Open firewall for the client (s) port 2049

Configuring the NFS Client Machine

Step 1: Install NFS common on all the worker nodes that need the NFS server.

Install the NFS client package

```
sudo apt update
sudo apt install nfs-common -y
```

Deployment of a MongoDB using the NFS

```
# mongo-nfs.yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: mongodb
spec:
  replicas: 3
  selector:
    matchLabels:
      app: mongo
  template:
    metadata:
      name: myapp
      labels:
        app: mongo
    spec:
      containers:
        volumes:
        - name: mongodbvol
          nfs:
            server: #Your server IP
            path: /mnt/share
      containers:
       - name: mongodbcontainer
         image: mongo
         ports:
         - containerPort: 27017
        env:
         - name: MONGO_INITDB_ROOT_USERNAME: admin
           value: MONGO_INITDB_ROOT_PASSWORD: admin123
```

```
          - name: MONGO_INITDB_DATABASE: mydatabase
            value: devdb@123
        volumeMounts:
        - name: mongodbvol
          mountPath: /data/db
        - name: vol2
          mountPath: /data/db
```

Setting up Kubernetes cluster on AWS using KOPS

- It is a software use to create production ready k8s cluster in a cloud provider like AWS
- It supports multiple cloud providers
- It competes with managed k8s services like EKS, AKS and GKE
- It is cheaper than the others
- It creates production ready k8s
- It creates resources like: LoadBalancer, AutoScaling Groups, Launch configuration, worker node, master node (Control plane)
- It is IaaC

#!/bin/bash

1) Create Ubuntu EC2 instance in AWS
2) a) Create kops user

```
sudo adduser kops
sudo echo "kops ALL=(ALL) NOPASSWD:ALL" | sudo tee /etc/sudoers.d/kops
sudo su - kops
```

Install AWSCLI using the apt package manager

```
sudo apt install awscli -y
```

OR

2b) Install AWSCLI using the script below

```
    sudo apt update -y
    sudo apt install wget unzip -y
```

```
    sudo curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o
"awscliv2.zip"
    sudo apt install unzip python -y
    sudo unzip awscliv2.zip
    sudo ./awscli-bundle/install -i /usr/local/bin/aws -b /usr/local/bin/aws
```

3) Install kops software on an ubuntu instance by running the commands below:

```
sudo apt install wget -y
sudo wget https://github.com/kubernetes/kops/releases/download/v1.22.0/kops-
linux-amd64
chmod +x kops-linux-amd64
sudo mv kops-linux-amd64 /usr/local/bin/kops
```

4) Install kubectl kubernetes client if it is not already installed

```
curl -LO curl -LO https://dl.k8s.io/release/v1.30.0/bin/linux/arm64/kubectl

chmod +x ./kubectl
sudo mv ./kubectl /usr/local/bin/kubectl
```

5) Create an IAM role from AWS console or CLI with the below policies

- ✓ AmazonEC2FullAccess
- ✓ AmazonS3FullAccess
- ✓ IAMFullAccess
- ✓ AmazonVPCFullAccess

Then attach IAM role to ubuntu server from the server console. Select created KOPS server➔Actions➔Instance Setting➔Attach/Replace IAM Role➔Select the role which you created➔Save

6) Create an S3 bucket

   Execute the commands below in your KOPS control server. Use unique S3 bucket name, if you get a bucket name exist error.

```
aws s3 mb s3://fewaitconsulting
aws s3 ls # To verify if s3 bucket is created
```

7) Expose environmental variables of the created bucket

```
#Add env variables in bashrc

sudo vi .bashrc

#!/bin/bash
```

```
# Give unique name to the created s3 bucket
export NAME=fewaitconsulting.k8s.local
export KOPS_STATE_STORE=s3://fewaitconsultingkops

source ~/.bashrc
```

8) Create ssh keys before creating cluster

```
ssh-keygen
```

9) Create kubernetes cluster definitions on s3 bucket

```
kops create cluster --zones us-east-1a --networking weave --master-size
t2.medium --master-count 1 --node-size t2.medium --node-count=2 ${NAME}
#Copy your sshkey into your cluster to be able to access your kubernetes
nodes
kops create secret --name ${NAME} sshpublickey admin -i ~/.ssh/id_rsa.pub
```

10) Initialize your kops kubernetes cluster by running the command below

```
kops update cluster {NAME} --yes
```

11) Validate your cluster (KOPS will take some time to create cluster. Execute commands below after 3 or 4 mins)

```
kops validate cluster
```

OR

Export the kubeconfig file to manage your kubernetes cluster from a remote server. For this demo, our remote server shall be our kops server.

```
kops export kubecfg $NAME --admin
```

12) To list nodes and pods to ensure that you can make calls to the kubernetes apiServer and run workloads, use the command below:

```
kubectl get nodes

#Number and stopping of nodes can be altered in the autoscaling group
service back in AWS
```

Persistent Volumes

- It is a piece o storage (hostPath, nfs, ebs,azurefile,azuredisk) in k8s cluster
- It exists independently from pod life cycle form which it is consuming

PersistentVolumeClaim

- It requests for storage (volume).
- Using PVC, we can request (specify) how much storage we need and with what access mode we need the volume.
- Persistent volumes are provisioned in two ways: Statistically or Dynamically.

1) Static volumes (manual provisioning)

- A k8s administrator can create a PV manually so that PV can be available for pods which requires.
- Create a PVC so that PVC will attach PV. We can use PVC with pods to get an access to PV.

2) Dynamic volumes (Dynamic provisioning)

- It's possible to have k8s provision (create) volumes (PV) as required, provided we have configured storage.

So, when we create PVC, if PV is not available, storage class will create PV dynamically.

PVC: If pod requires access to storage (PV), it will get an access using PVC. PVC will be attached to PV.

- ✓ **Persistentvolume:** The low-level representation of a storage volume.
- ✓ **PeristentVolumeClaim:** The binding between a pod and persistentvolume
- ✓ **Pod**: A running container that will consume a persistentvolume.
- ✓ **StorageClass:** It allows for dynamic provisioning of persistentvolumes.

PersistentVolume will have access modes:

- ReadWriteOnce →The volume can be mounted as read-write by a single mode
- ReadOnlyMany →The volume can be mounted as read-only by many modes
- ReadWriteMany →The volume can be mounted as read-write by many nodes

Claim Policies

A persistent volume claim can have several different claims policies associated with it including:

- Retain→When the claim (PVC) is deleted, the volume (PV) will exist
- Recycle→ When the claim is deleted, then the volume remains but, in a state, where the data can be manually recovered.
- Delete→ The persistent volume is deleted when the claim is deleted.

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

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-hostpath
  labels:
    type: local
spec:
 storageClassName: manual
 capacity:
   storage: 1Gi
 accessModes:
   - ReadWriteMany
 nfs:
  server: <nfs server ip>
  path: "/mnt/share"


---
apiVersion: v1
kind: PeristentVolumeClaim
metadata:
  name: pvc-nfs-pv1
  labels:
    type: local
spec:
 storageClassName: manual
 capacity:
   storage: 10Gi
 accessModes:
   - ReadWriteMany
 resources:
   requests:
     storage: 500Mi
```

Example: Deployment of Mongodb application with the use of PVC

```
# pv-pvc-mongo-hostpath.yaml

apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-hostpath
  labels:
    type: local
spec:
 storageClassName: manual
 capacity:
   storage: 10Gi
 accessModes:
   - ReadWriteOnce
 hostPath:
   path: "/kube"


---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pv-hostpath
  labels:
    type: local
spec:
 storageClassName: manual
 capacity:
   storage: 300Mi
 accessModes:
   - ReadWriteOnce
 resources:
   requests:
     storage: 500Mi
 hostPath:
   path: "/kube"

---
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: mongodbrs
spec:
  replicas: 3
  selector:
    matchLabels:
      app: mongodb
  template:
    metadata:
      name: mongodbpod
      labels:
```

```
        app: mongodb
    spec:
      containers:
        volumes:
        - name: mongodbpvc
          persistentVolumeClaim:
            claimName: pvc-hostpath
      containers:
       - name: mongodbcontainer
         image: mongodb
         ports:
         - containerPort: 27017
        env:
         - name: MONGO_INITDB_ROOT_USERNAME: admin
           value: MONGO_INITDB_ROOT_PASSWORD: admin123
         - name: MONGO_INITDB_DATABASE: mydatabase
           value: devdb@123
        volumeMounts:
        - name: mongodbvol
          mountPath: /data/db
        - name: vol2
          mountPath: /data/db
```

```
pv-pvc-mongo-nfs.yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-nfs-pv1
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteMany
  nfs:
    server: <nfs server ip>
    path: "/mnt/share"
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-nfs-pv1
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 500Mi
---
apiVersion: apps/v1
```

```
kind: ReplicaSet
metadata:
  name: mongodbrs
spec:
  selector:
    matchLabels:
      app: mongodb
  template:
     metadata:
       name: mongodbpod
       labels:
         app: mongodb
     spec:
       volumes:
       - name: mongodb-pvc
         persistentVolumeClaim:
           claimName: pvc-nfs-pv1
       containers:
       - name: mongodbcontainer
         image: mongo
         ports:
         - containerPort: 27017
         env:
         - name: MONGO_INITDB_ROOT_USERNAME
           value: devdb
         - name: MONGO_INITDB_ROOT_PASSWORD
           value: devdb@123
         volumeMounts:
         - name: mongodb-pvc
           mountPath: /data/db
```

Complete Manifest Where in single yml we defined Deployment & Service for SpringApp & PVC (with default StorageClass),ReplicaSet & Service For Mongo.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: springappdeployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: springapp
  template:
    metadata:
      name: springapppod
      labels:
        app: springapp
    spec:
      containers:
      - name: springappcontainer
        image: fewaitconsulting/spring-boot-mongo
```

```
        ports:
        - containerPort: 8080
        env:
        - name: MONGO_DB_USERNAME
          value: devdb
        - name: MONGO_DB_PASSWORD
          value: devdb@123
        - name: MONGO_DB_HOSTNAME
          value: mongo
---
apiVersion: v1
kind: Service
metadata:
  name: springapp
spec:
  selector:
    app: springapp
  ports:
  - port: 80
    targetPort: 8080
  type: LoadBalancer
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mongodbpvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
---
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: mongodbrs
spec:
  selector:
    matchLabels:
      app: mongodb
  template:
     metadata:
       name: mongodbpod
       labels:
         app: mongodb
     spec:
       volumes:
       - name: pvc
         persistentVolumeClaim:
           claimName: mongodbpvc
       containers:
       - name: mongodbcontainer
         image: mongo
         ports:
         - containerPort: 27017
         env:
```

```
          - name: MONGO_INITDB_ROOT_USERNAME
            value: devdb
          - name: MONGO_INITDB_ROOT_PASSWORD
            value: devdb@123
          volumeMounts:
          - name: pvc
            mountPath: /data/db
---
apiVersion: v1
kind: Service
metadata:
  name: mongo
spec:
  type: ClusterIP
  selector:
    app: mongodb
  ports:
  - port: 27017
    targetPort: 27017
```

Kubernetes ConfigMaps and Secrets

- **ConfigMaps** are used to pass additional configurations needed to orchestrate containers including environmental variables. Examples is hostname or password.
- **ConfigMaps** pass values in plain text key: value pairs
- **Kubernetes Secrets** let you store and manage sensitive information, such as passwords, OAuth tokens, and ssh keys.
- Storing confidential information in a Secret is safer and more flexible than putting it directly in a Pod definition or in a container image.

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: mongo-configmap
data:
  # Configuration values can be set as key-value properties
  db-password: devdb@123
  db-hostname: devdb
 ---
apiVersion: v1
kind: Secret
metadata:
  name: mongo-db-password
#type: Opaque means that from kubernetes's point of view the contents of
this Secret is unstructured.
#It can contain arbitrary key-value pairs.
type: Opaque
```

```
data:
  # Output of "echo -n 'devdb@123' | base64"
  db-password: ZGV2ZGJAMTIz
```

Example: Using of ConfigMaps and Secrets to deploy a springapp which uses a mongodb database

```
# Complete Manifest Where in single yml we defined Deployment & Service
for SpringApp & PVC(with default  StorageClass),ReplicaSet & Service For
Mongo.
apiVersion: apps/v1
kind: Deployment
metadata:
  name: springappdeployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: springapp
  template:
    metadata:
      name: springapppod
      labels:
        app: springapp
    spec:
      containers:
      - name: springappcontainer
        image: mylandmarktech/spring-boot-mongo
        ports:
        - containerPort: 8080
        env:
        - name: MONGO_DB_USERNAME
          valueFrom:                #Make sure to create the respectively
configMap and
            configMapKeyRef:         # secret yaml files accordingly
before creating this file.
              name: springappconfig
              key: db-username
        - name: MONGO_DB_PASSWORD
          secreteKeyRef:
              name: springappsecret
              key: db-password
        - name: MONGO_DB_HOSTNAME
          valueFrom:
           configMapKeyRef:
              name: springappconfig
              key: db-hostname
---
apiVersion: v1
kind: Service
```

```
metadata:
  name: springapp
spec:
  selector:
    app: springapp
  ports:
  - port: 80
    targetPort: 8080
  type: LoadBalancer
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mongodbpvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
---
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: mongodbrs
spec:
  selector:
    matchLabels:
      app: mongodb
  template:
     metadata:
       name: mongodbpod
       labels:
         app: mongodb
     spec:
       volumes:
       - name: pvc
         persistentVolumeClaim:
           claimName: mongodbpvc
       containers:
       - name: mongodbcontainer
         image: mongo
         ports:
         - containerPort: 27017
         env:
         - name: MONGO_INITDB_ROOT_USERNAME
           value: devdb
         - name: MONGO_INITDB_ROOT_PASSWORD
           value: devdb@123
         volumeMounts:
         - name: pvc
           mountPath: /data/db
---
apiVersion: v1
kind: Service
metadata:
  name: mongo
```

```
spec:
  type: ClusterIP
  selector:
    app: mongodb
  ports:
  - port: 27017
    targetPort: 27017
```