Kubernetes (K8s)

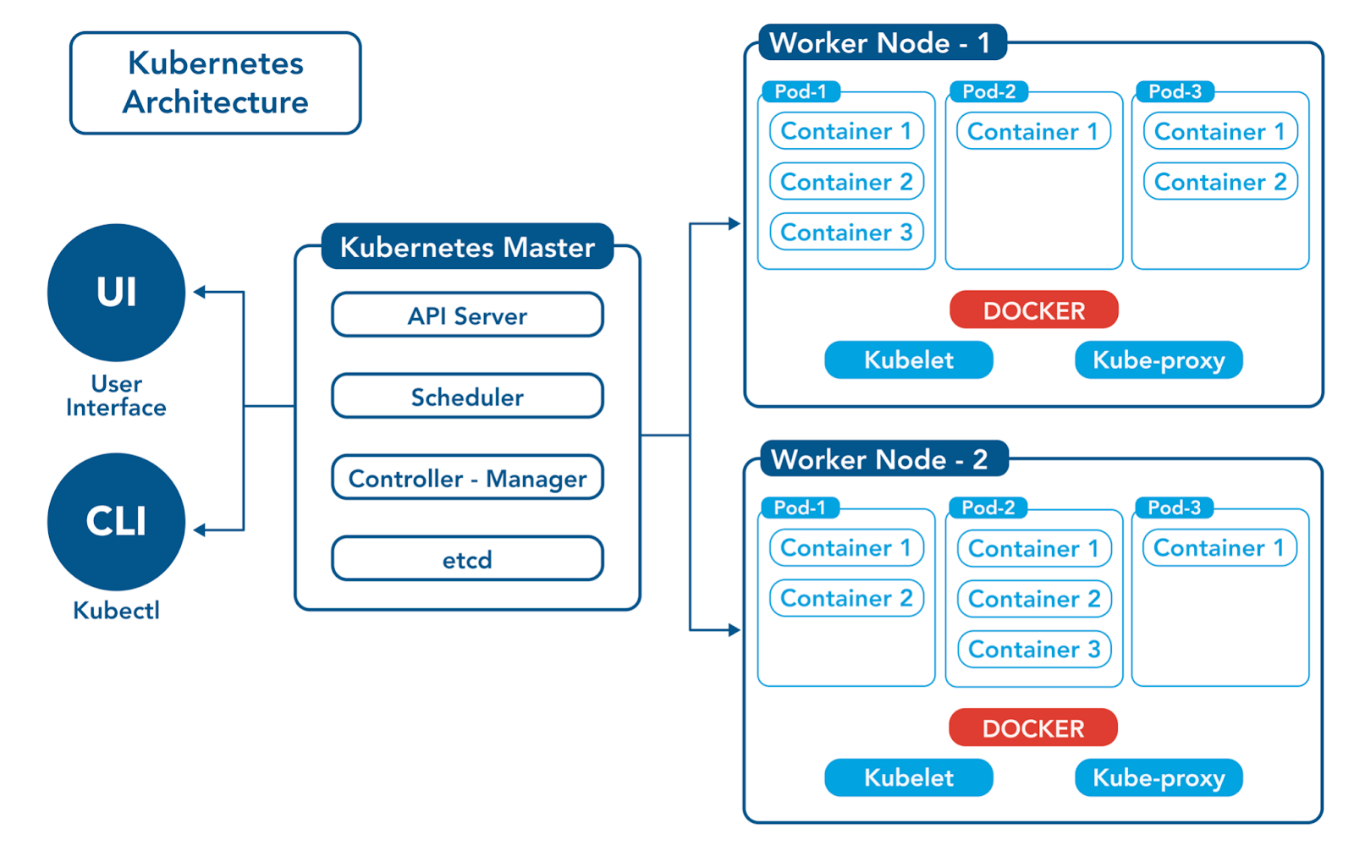
Kubernetes is an open source container **orchestration** engine for automating deployment, scaling, and management of containerized applications. The open source project is hosted by the Cloud Native Computing Foundation (CNCF).

It provides a scalable and resilient framework for automating the deployment, scaling, and management of applications across clusters of servers.

**A SMALL HISTORY OF K8S:**

* In the early 2000s, Google started developing a system called Borg to manage their internal containerized applications.
* Borg enabled Google to run applications at scale, providing features such as automatic scaling, service discovery, and fault tolerance.
* In 2014, Google open-sourced a version of Borg called Kubernetes.
* Kubernetes was donated to the Cloud Native Computing Foundation (CNCF), a neutral home for open-source cloud-native projects, in July 2015.
* Kubernetes 1.8 added significant enhancements for storage, security, and networking. Key features included the stable release of the stateful sets API, expanded support for volume plugins, and improvements in security policies.
* Check URL: <https://kubernetes.io/releases/> for more release details.

**KUBERNETES ARCHITECTURE**



**Control Plane /Master Node**

The control plane's components make global decisions about the cluster (for example, scheduling), as well as detecting and responding to cluster events (for example, starting up a new pod when a deployment's replicas field is unsatisfied).

Control plane components can be run on any machine in the cluster. Do not run user containers on this machine.

**Node Components / Worker Nodes**

Node components run on every node, maintaining running pods and providing the Kubernetes runtime environment.

1. **Master Node:** The master node is responsible for managing the cluster and coordinating the overall state of the system. It includes the following components:
   1. **API Server:** The API server is the *central control point for all interactions* with the cluster. It exposes the Kubernetes API and handles requests from users and other components.
   2. **Scheduler:** The scheduler is *responsible for assigning workloads* (pods) to individual worker nodes based on resource requirements, constraints, and other policies.
   3. **Controller Manager:** The controller manager runs various controllers that monitor the cluster state and drive it towards the desired state. Examples include the replication controller, node controller, and service controller.
   4. **etcd:** etcd is a *distributed key-value store* used by Kubernetes to store cluster state and configuration data.
2. **Worker Nodes:** Worker nodes are the machines where the containers run. They execute the tasks assigned to them by the master node. Each worker node consists of the following components:
   1. **Kubelet:** The kubelet is the primary *agent* running on each worker node. It *communicates with the master* node and manages the state of the node, ensuring that pods are running and healthy.
   2. **Container Runtime:** The container runtime, such as Docker or containerd, is responsible for pulling container images and running containers on the worker node.
   3. **Kube-proxy:** Kube-proxy is a network proxy that handles *routing and load balancing* for services within the cluster. It maintains *network* rules and enables service discovery.
   4. **Pods:** Pods are the basic units of deployment in Kubernetes. They encapsulate one or more containers and associated resources (such as shared volumes and network namespaces) that are co-located and co-scheduled on the same worker node.

**Networking:** Networking in Kubernetes allows containers and services to communicate with each other within the cluster. Networking plugins, such as *Calico, Flannel, or Cilium*, handle the routing and connectivity between pods and services.

**Storage:** Kubernetes provides various options for persistent storage. Storage classes, volume plugins, and persistent volume claims enable containers to request and use persistent storage resources.

**OBJECT IN KUBERNETES:**

In Kubernetes, objects are entities used to define and manage the state of the cluster.

They represent various components and resources within the cluster, such as applications, services, storage, and networking.

Kubernetes objects are defined using YAML or JSON configuration files and can be created, updated, or deleted using the Kubernetes API.

1. **Pod:** The basic building block of Kubernetes. A pod represents a single instance of a running process within the cluster. It can encapsulate one or more containers that share the same network and storage resources.
2. **ReplicaSet**: It ensures a specified number of pod replicas are running at all times. It is responsible for maintaining the *desired replica count of a pod* template in the cluster. If the actual number of replicas falls below the desired count, the ReplicaSet creates additional pods to match the desired count.
3. **Deployment:** A higher-level abstraction that manages and updates a set of replica pods. Deployments provide declarative updates, rolling updates, and rollback capabilities for application deployments.
4. **Service:** A service provides a stable network endpoint for accessing a group of pods. It enables load balancing and automatic service discovery within the cluster, allowing other applications to interact with the pods without needing to know their individual IP addresses.
5. **Namespace:** Namespaces are used to logically divide a cluster into virtual clusters. They provide a way to organize and *isolate resources*, allowing different teams or applications to have their own isolated environment within the cluster.
6. **ConfigMap:** A ConfigMap stores configuration data as *key-value pairs*. It allows you to separate configuration from application code and provides a centralized way to manage and update configuration settings.
7. **Secret:** Secrets are used to store sensitive information such as passwords, API keys, and TLS certificates. They provide a secure way to distribute and manage sensitive data within the cluster.
8. **Volume:** A volume provides a way to *store* and access data in a pod. It enables containers within a pod to share and persist data independently of the pod's lifecycle.
9. **StatefulSet:** StatefulSets manage stateful applications that require stable network identities and *persistent storage*. They ensure that pods are created in a predictable order and provide stable network identities and persistent storage volumes.
10. **DaemonSet:** A DaemonSet ensures that a pod runs on every node in the cluster. It is useful for deploying system daemons, log collectors, and other utilities that need to run on every node.
11. **Job:** Jobs manage batch or single-run tasks. They create one or more pods to run the task and ensure that the specified number of successful completions is achieved.

**KUBERNETES CLUSTER SETUP:**

1. **Minikube**: Minikube is a tool that allows you to run a single-node Kubernetes cluster on your local machine. It is ideal for development, testing, and learning purposes. Minikube provides an easy way to get started with Kubernetes without the need for a full-scale cluster.
2. **Kubeadm:** Kubeadm is a command-line tool provided by the Kubernetes project for bootstrapping a cluster on your own infrastructure. It simplifies the setup process by handling the initialization of control plane components, joining worker nodes, and configuring networking. Kubeadm is commonly used for creating custom Kubernetes clusters on virtual machines or bare-metal servers.
3. **Kubernetes as a Service (KaaS):** Some cloud providers offer managed Kubernetes services, which handle the cluster setup and management on your behalf. These services abstract away the underlying infrastructure and provide an easy way to deploy and scale Kubernetes clusters without worrying about the operational complexities. Examples include Amazon EKS, GKE, AKS, and DigitalOcean Kubernetes.

**MINIKUBE INSTALLATION IN AMAZON LINUX 2023:**

**Type: t3.medium (2-CPU and 4-GB RAM)**

**[STOP INSTANCE AFTER YOUR USAGE IS DONE, IT IS A BILLABLE INSTANCE]**

**1. Install Docker**

$ sudo yum update -y

$ sudo yum install docker -y

**2. Start and Enable Docker**

$ sudo systemctl start docker

$ sudo systemctl enable docker

**3. Add Current user to docker group**

$ sudo usermod -a -G docker $USER

$ sudo systemctl status docker

**4. Exit from current session and re-connect**

$ exit

**Press 'R' to re-connect**

**5. Download and Install Kubectl Client**

$ sudo curl --silent --location -o /usr/local/bin/kubectl https://dl.k8s.io/release/$(curl --silent --location https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl

$ sudo chmod +x /usr/local/bin/kubectl

$ kubectl version --client

**6. Download and Install Minikube Software**

$ curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64

$ sudo install minikube-linux-amd64 /usr/local/bin/minikube

**7. Start Minikube software**

$ minikube start

**8. Check Installation status using these commands**

$ minikube status

$ minikube version

$ kubectl version

$ kubectl cluster-info

$ kubectl get nodes

**9. If you stop and start your EC2 Instance then you need to start again Minikube software**

# Check Status (it may show **Stopped** message)

$ minikube status

# Wait 1-2 minutes to start

$ minikube start

**KUBEADM INSTALLATION IN AWS - UBUNTU 20.04 LTS:**

**Type: t3.medium (2-CPU and 4-GB RAM)**

**[STOP INSTANCE AFTER YOUR USAGE IS DONE, IT IS A BILLABLE INSTANCE]**

**INSTALL IN ALL NODES (BOTH MASTER AND WORKER NODES)**

1. **Upgrade apt packages**

$ sudo apt-get update

1. **Create configuration file for containerd:**

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

EOF

1. **Load modules:**

$ sudo modprobe overlay

$ sudo modprobe br\_netfilter

1. **Set system configurations for Kubernetes networking:**

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

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

net.ipv4.ip\_forward = 1

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

EOF

1. **Apply new settings:**

$ sudo sysctl --system

1. **Install containerd:**

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

1. **Create default configuration file for containerd:**

$ sudo mkdir -p /etc/containerd

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

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

1. **Restart containerd to ensure new configuration file usage:**

$ sudo systemctl restart containerd

1. **Verify that containerd is running (optional)**

$ sudo systemctl status containerd (presss q for exit)

1. **Disable swap:**

$ sudo swapoff -a

1. **Disable swap on startup in /etc/fstab:**

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

1. **Install dependency packages.**

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

1. **Download and add GPG key.**

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

1. **Add Kubernetes to repository list.**

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

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

EOF

1. **Update package listings**

$ sudo apt-get update

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

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

1. **Turn off automatic updates:**

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

**MASTER NODE SETUP (EXECUTE ONLY AT MASTER NODE)**

1. **Initialize the Cluster**

$ sudo kubeadm init

1. **Set kubectl access:**

$ mkdir -p $HOME/.kube

$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config

$ sudo chown $(id -u):$(id -g) $HOME/.kube/config

1. **Test access to cluster:**

$ kubectl get nodes

1. **Install the Calico Network Add-On - On the Control Plane Node, install Calico Networking:**

$ kubectl apply -f <https://raw.githubusercontent.com/projectcalico/calico/master/manifests/calico.yaml>

[check for the latest here: <https://github.com/projectcalico/calico/blob/master/manifests/calico.yaml>]

$ kubectl get nodes

1. **Join the Worker Nodes to the Cluster**

$ kubeadm token create --print-join-command

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

$ sudo kubeadm join ...

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

1. Validate the setup by executing below command in master-node

$ kubectl get nodes

**Pods:**

In Kubernetes, a pod is the smallest and simplest unit of deployment.

It represents a single instance of a running process within a cluster.

Pods are used to encapsulate one or more containers, along with shared resources and configuration that are co-located and share the same context.

* **Container encapsulation:** Pods provide a way to encapsulate and manage one or more containers together as a single unit. Containers within a pod share the same network namespace, IP address, and ports. They can communicate with each other using localhost.
* **Atomic scheduling:** Pods are the basic unit of scheduling in Kubernetes. When scheduling pods, Kubernetes ensures that all containers within a pod are scheduled on the same node. This helps ensure that containers in the same pod have low-latency communication and can share resources efficiently.
* **Shared resources:** Pods share certain resources, such as storage volumes and IP addresses. Each pod has its own unique IP address within the cluster, enabling direct communication between pods using IP-based protocols.
* **Lifecycle coordination:** Pods provide a way to coordinate the lifecycle of multiple containers. When a pod is created, all containers within the pod are started simultaneously. Similarly, when a pod is deleted, all containers within the pod are terminated together.
* **Single service endpoint:** Pods are not directly exposed outside the cluster. Instead, they are typically used as building blocks for higher-level abstractions like services and deployments. Services provide a stable endpoint to access pods, allowing external traffic to be load-balanced across multiple pod instances.
* **Scalability and resilience:** Pods can be *horizontally scaled* by creating multiple replicas of the same pod template. Kubernetes handles the distribution of replicas across nodes and ensures load balancing. If a pod fails, Kubernetes can automatically reschedule and recreate the pod to maintain the desired replica count.
* **Metadata and labels:** Pods can be labelled and annotated with metadata, allowing for easy *grouping, selection, and management*. Labels are key-value pairs attached to pods, enabling flexible pod selection for operations like scaling, deployment, and networking.

**Note: We can create any object in Kubernetes using command or YAML Files. Best one is YAML file.**

**K8s Pod Commands: (short name = po )**

**1. Create a pod using run command**

$ kubectl run <pod-name> --image=<image-name> --port=<container-port>

$ kubectl run my-pod --image=nginx --port=80

**2. View all the pods**

**(In default namespace)**

$ kubectl get pods

**(In All namespace)**

$ kubectl get pods -A

**# For a specific namespace**

$ kubectl get pods -n kube-system

**# For a specific type**

$ kubectl get pods <pod-name>

$ kubectl get pods <pod-name> -o wide

$ kubectl get pods <pod-name> -o yaml

$ kubectl get pods <pod-name> -o json

**3. Describe a pod (View Pod details)**

$ kubectl describe pod <pod-name>

$ kubectl describe pod my-pod

**4. View Logs of a pod**

$ kubectl logs <pod-name>

$ kubectl logs my-pod

**5. Execute any command inside Pod (Inside Pod OS)**

$ kubectl exec <pod-name> -- <command>

$ kubectl exec my-pod -- cat /etc/os-release

$ kubectl exec my-pod -- ls /etc/

$ kubectl exec my-pod -- echo $SHELL

**6. to link with HOST PORT**

$ kubectl port-forward <pod-name> <local-port>:<pod-port>

$ kubectl port-forward my-pod 8080:80 --address 0.0.0.0 &

# Here 0.0.0.0. Indicates access from anywhere (internet location) & means run in background

**7. use commands to stop Nginx Service**

$ ps –aux | grep ‘8080’

$ kill <process-id>

**8. Get Pod IP Address**

$ kubectl get pod <pod-name> -o wide

$ kubectl get pod <pod-name> -o jsonpath='{.status.podIP}'

**9. Delete pod**

$ kubectl delete pod <pod-name>

$ kubectl delete pod my-pod

**TASK#1 Pull javabyraghu/maven-web-app image from docker and expose using Pod IP:PORT**

$ kubectl run my-app --image javabyraghu/maven-web-app --port=8080

$ kubectl port-forward --address 0.0.0.0 pod/my-app 8080:8080 &

# Enable Port 8080 in EC2 Security Group and Enter URL

**http://<Public-IP>:8080/maven-web-app/**

**CREATING POD USING YAML FILE**

**1. nginx-pod.yml creation**

**apiVersion: v1**

**kind: Pod**

**metadata:**

**name: my-pod**

**labels:**

**app: my-web-app**

**type: backend**

**spec:**

**containers:**

**- name: nginx-container**

**image: nginx**

**ports:**

**- containerPort: 80**

**# Execute command to create pod**

$ kubectl apply -f nginx-pod.yml

**# To delete configure pods from YAML**

$ kubectl delete -f nginx-pod.yml

**# to create PODS from YAML**

$ kubectl create -f nginx-pod.yml

2. Create Maven Web Application

**apiVersion: v1**

**kind: Pod**

**metadata:**

**name: my-app**

**spec:**

**containers:**

**- name: my-app-container**

**image: javabyraghu/maven-web-app**

**ports:**

**- containerPort: 80**

**ReplicaSet:**

ReplicaSet is a resource object used to ***ensure that a specified number of identical Pods are running at all times***. It is part of the Kubernetes Replication Controller family and provides a higher-level abstraction for managing and scaling sets of Pods.

* **Pod Replication:** ReplicaSets define the *desired number of Pod replicas* that should be running. If the actual number of Pods deviates from the desired state, the ReplicaSet takes action to reconcile the difference by either creating new Pods or terminating existing ones.
* **Selector-based Matching:** ReplicaSets use a selector to identify the Pods it manages. The selector uses labels to match Pods based on their metadata, allowing the ReplicaSet to control and manage a set of Pods with specific labels.
* **Automatic Scaling:** ReplicaSets support automatic scaling. You can scale the number of replicas up or down to handle changes in demand by modifying the ReplicaSet's replica count.
* **Pod Template:** ReplicaSets use a Pod template to create new Pods or replace terminated ones. The template specifies the container image, resource requirements, labels, and other configuration options for the Pods.
* **Immutable Updates:** ReplicaSets treat updates to the Pod template as immutable. To update a ReplicaSet, you typically create a new ReplicaSet with the updated template, and Kubernetes handles the process of scaling up the new ReplicaSet and scaling down the old one.
* **Owner-Reference:** ReplicaSets are the owner of the Pods they manage. They set themselves as the owner of the Pods using an owner reference, which helps maintain the relationship between the ReplicaSet and its managed Pods.

ReplicaSets are typically used for stateless applications where individual Pods are interchangeable. They provide high availability and fault tolerance by ensuring that the desired number of Pods are always running, even in the face of failures or node disruptions.

Note: ***It is not recommended to use ReplicaSet, better use Deployment*** in favour of the more powerful and flexible Deployment resource. Deployments provide declarative updates, rolling updates, and rollback capabilities on top of ReplicaSets.

We cannot create rs(ReplicaSet) using command without YAML file. We need to define one YAML File and then execute as create/apply command.

**K8s ReplicaSet Commands: (short name = rs )**

**1. Create one YAML file (ex: rs-test.yml) and paste below content in that file**

$ sudo nano rs-test.yml

**apiVersion: apps/v1**

**kind: ReplicaSet**

**metadata:**

**name: my-rs**

**labels:**

**name: my-rs**

**spec:**

**replicas: 4**

**selector:**

**matchLabels:**

**apptype: web-backend**

**template:**

**metadata:**

**labels:**

**apptype: web-backend**

**spec:**

**containers:**

**- name: my-app**

**image: javabyraghu/maven-web-app**

**ports:**

**- containerPort: 8080**

**2. Create ReplicaSet by executing above YAML file**

$ kubectl create -f rs-test.yml

**# Do necessary modifications if exist, else create new**

$ kubectl apply -f rs-test.yml

**# Completely Modify Pod Template**

$ kubectl replace -f rs-test.yml

**3. View ReplicaSets**

$ kubectl get replicasets

$ kubectl get rs

$ kubectl get rs -o wide

$ kubectl get rs <replica-set-name> -o json

$ kubectl get rs <replica-set-name> -o yaml

**4. View ReplicaSet Description**

$ kubectl describe rs <replica-set-name>

**5. We can modify generated/updated YAML file**

$ kubectl edit rs <replica-set-name>

**## change replicas: count to any other value then (ESC):wq**

**# We can modify our YAML file and then execute apply command**

$ kubectl apply -f rs-test.yml

**## We can Even scale using command also**

$ kubectl scale replicaset <replicaset-name> --replicas=<desired-replica-count>

**6. Delete ReplicaSet**

$ kubectl delete rs <replica-set-name>

$ kubectl delete -f rs-test.yml

**Note: kubectl logs and kubectl exec works only on pod, not on rs**

**Deployment:**

Deployments are one of the most commonly used resources in Kubernetes for managing application deployments and scaling. They provide an abstraction layer that simplifies the management and updates of pods, making it easier to ensure the desired state of your application.

Deployments offer built-in support for rolling updates, allowing you to update your application while minimizing downtime. You can define strategies for how many new pods to create and how many old pods to terminate at each step of the update.

* **Recreate Deployment:**

This strategy involves *terminating all instances* of the existing application and *creating new pods* with the updated version *all at once*. It results in a brief downtime during the update process but can be useful for stateless applications with fast startup times.

* **RollingUpdate Deployment:**

This is the *default deployment strategy* in Kubernetes. It gradually replaces old pods with new ones, ensuring a smooth transition. You can define parameters such as the maximum number of unavailable pods and the maximum number of new pods created simultaneously.

**Deployment Models:**

* **Blue-Green Deployment:**

In a Blue-Green deployment, you maintain ***two identical environments*** (blue and green) running different versions of your application. The traffic is routed to one environment (e.g., blue), while the other (e.g., green) is updated with a new version. Once the update is complete and verified, the traffic is switched to the updated environment.

* **Canary Deployment:**

Canary deployments involve releasing a new version of your application to ***a small subset of users or traffic***, allowing you to test and validate the new version before rolling it out to the entire user base. It helps identify issues or performance problems before affecting a larger audience.

* **A/B Testing:**

A/B testing involves deploying ***multiple versions of an application*** simultaneously and directing a portion of the traffic to each version. This allows you to ***compare the performance, features, or user experience*** of different versions and make data-driven decisions about which version to continue with.

**K8s Deployment Commands: (short name = deploy )**

**Hint: $ kubectl create deployment --help (use this for more options)**

**1. Create Deployment using command (Not recommended)**

$ kubectl create deployment <name> --image <image-name> --replicas <count>

$ kubectl create deployment my-depl --image nginx --replicas 3

**2. Create one YAML file (ex: deploy-test.yml) and paste below content in that file**

$ sudo nano web-deploy.yml

**apiVersion: apps/v1**

**kind: Deployment**

**metadata:**

**name: my-deploy**

**labels:**

**name: my-deploy**

**spec:**

**replicas: 4**

**selector:**

**matchLabels:**

**apptype: web-backend**

**strategy:**

**type: RollingUpdate**

**template:**

**metadata:**

**labels:**

**apptype: web-backend**

**spec:**

**containers:**

**- name: my-app**

**image: javabyraghu/maven-web-app**

**ports:**

**- containerPort: 8080**

**2. Create Deployment by executing above YAML file**

$ kubectl create -f web-deploy.yml

**# Do necessary modifications if exist, else create new**

$ kubectl create -f web-deploy.yml

**# Completely Modify Pod Template**

$ kubectl replace –f web-deploy.yml

**3. View Deployments**

$ kubectl get deployments

$ kubectl get deploy

$ kubectl get deploy -o wide

$ kubectl get deploy <deployment-name> -o json

$ kubectl get deploy <deployment-name> -o yaml

**4. View Deployment Description**

$ kubectl describe deploy <deployment-name>

**5. We can modify generated/updated YAML file**

$ kubectl edit deploy <deployment-name>

## change replicas: count to any other value then (ESC):wq

**# We can modify our YAML file and then execute apply command**

$ kubectl apply -f web-deploy.yml

**## We can Even scale using command also**

$ kubectl scale deploy <deployment-name> --replicas=<desired-replica-count>

**6. Delete Deployment**

$ kubectl delete deploy <deployment-name>

$ kubectl delete -f web-deploy.yml

**Note: A way to generate YAML file without wiring it manually**

$ kubectl create deployment my-deploy --image nginx --replicas 4 **--dry-run=client** -o yaml > test-deployment.yml

**Then use command,**

$ kubectl apply -f test-deployment.yml

**KUBECTL ROLLOUT COMMANDS:**

1. **Check the rollout status of a Deployment:**

$ kubectl rollout status deployment my-deployment

This command displays the rollout status of the my-deployment Deployment, showing whether the update has completed or is still in progress.

1. **View the revision history of a Deployment:**

$ kubectl rollout history deployment my-deployment

This command shows the revision history of the my-deployment Deployment, displaying the timestamps and revision numbers of each update.

1. **Pause a Deployment rollout:**

$ kubectl rollout pause deployment my-deployment

This command pauses the rollout of the my-deployment Deployment, suspending any further updates.

1. **Resume a paused Deployment rollout:**

$ kubectl rollout resume deployment my-deployment

This command resumes the rollout of the my-deployment Deployment, allowing updates to proceed.

1. **Undo the most recent Deployment update:**

$ kubectl rollout undo deployment my-deployment

This command undoes the most recent update of the my-deployment Deployment, rolling back to the previous revision.

1. **Rollback to a specific revision:**

$ kubectl rollout undo deployment my-deployment --to-revision=<revision>

This command rolls back the my-deployment Deployment to a specific revision specified by <revision>. Replace <revision> with the desired revision number.

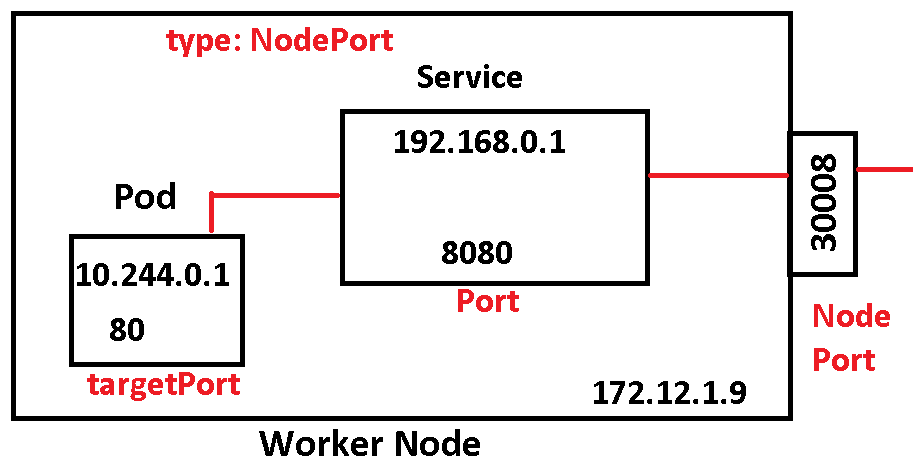
**Services (short name = svc):**

Service is an abstraction that defines a logical set of pods and a policy to access them. Services enable network connectivity and load balancing to the pods that are part of the service, allowing other components within or outside the cluster to interact with the application.

**Service Types:** Kubernetes supports different types of services:

1. NodePort: Exposes the service on a static port on each selected node's IP. This type makes the service accessible from outside the cluster by the <NodeIP>:<NodePort> combination.
2. ClusterIP: Exposes the service on a cluster-internal IP. This type makes the service only reachable within the cluster.
3. LoadBalancer: Creates an external load balancer in cloud environments, which routes traffic to the service.

**Node Port:** A Service is created with generated IP and Provided Port, that is connected to Pod and Node. Where Node gets generated port in between 30000-32767 (if not given manually).



**a. Create one pod using below YAML file.**

$ sudo nano my-pod.yml

apiVersion: v1

kind: Pod

metadata:

  name: my-pod

  labels:

    title: my-pod

    type: web-app

spec:

  containers:

  - name: my-pod

    image: javabyraghu/maven-web-app

    ports:

     - containerPort: 8080

**Execute below command to create POD using YAML file.**

$ kubectl apply -f my-pod.yml

**b. Create one service using below YAML file.**

**$ sudo nano my-service.yml**

apiVersion: v1

kind: Service

metadata:

  name: my-service

  labels:

    app: my-service

    type: backend-app

spec:

  type: NodePort

  ports:

   - targetPort: 8080

     port: 8080

     nodePort: 30002

  selector:

    title: my-pod

    type: web-app

**Execute below command to create SERVICE using YAML file**

$ kubectl apply -f my-service.yml

$ kubectl create -f my-service.yml

**View all Services created.**

$ kubectl get services

$ kubectl get svc

**Describe existed Services.**

$ kubectl describe svc <service-name>

**Delete a Service by name.**

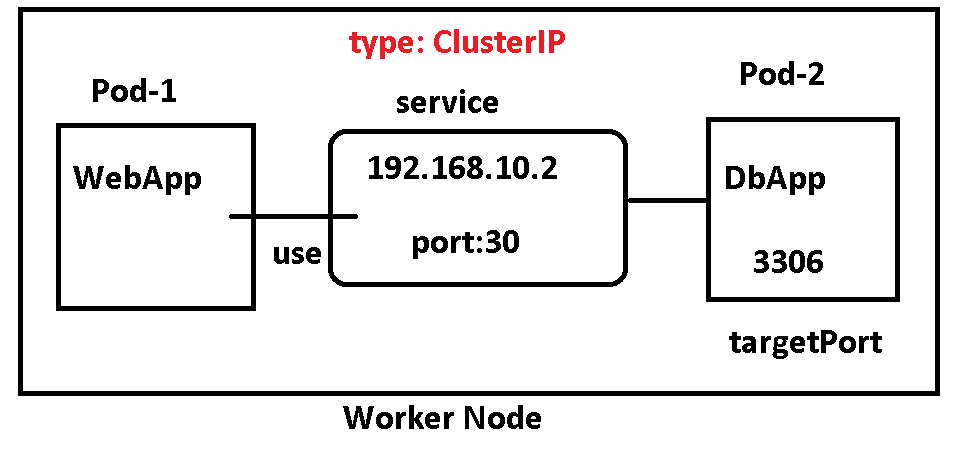
$ kubectl delete service <service-name>

**EXPOSE URL (in case of minikube)**

$ minikube service <service-name> --url

$ curl http://URL

**ClusterIP:** If one POD wants to connect with another POD within the cluster (or inside nodes only). Like connecting backend application with database or cache, frontend application with backend application, then we can use this. This one creates a service with one generated IP and PORT assigned, mapped to POD container port.



**Sample YAML FILE:**

apiVersion: v1

kind: Service

metadata:

  name: my-service

  labels:

    app: my-service

    type: backend-app

spec:

  type: ClusterIP

  ports:

   - targetPort: 8080

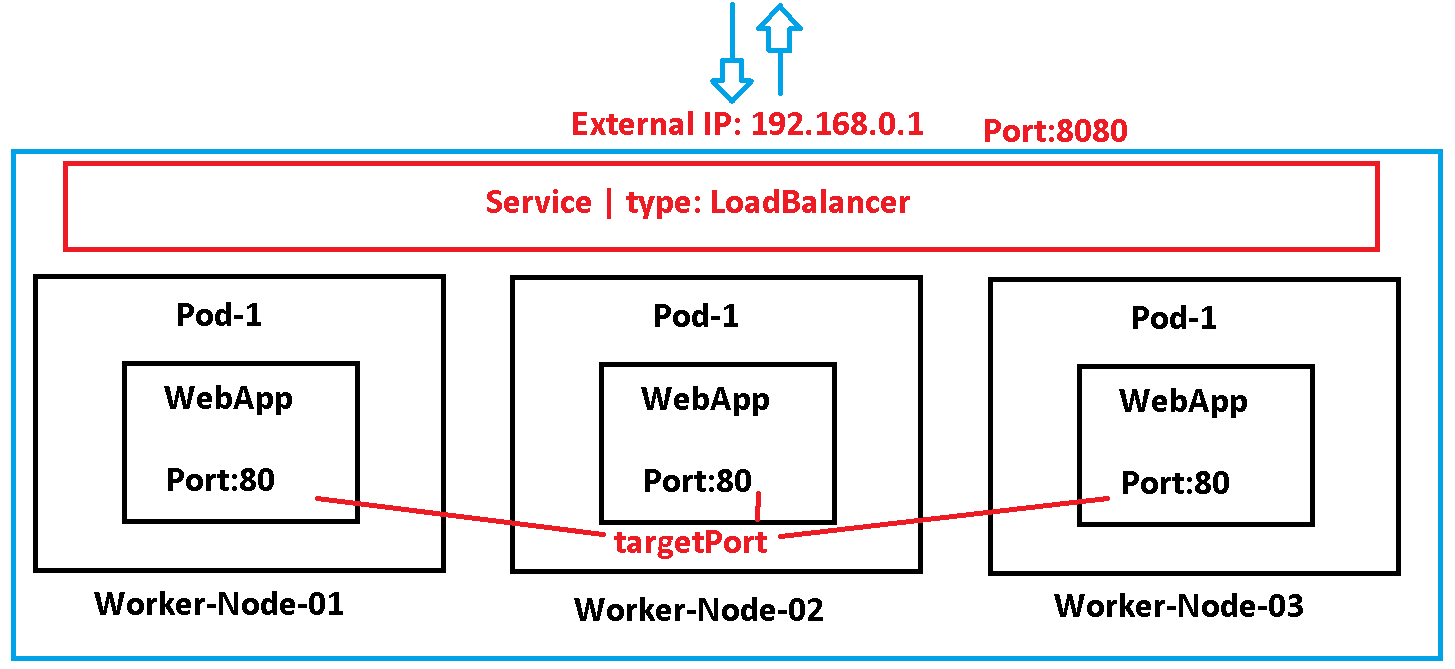
     port: 8080

  selector:

    title: my-pod

    type: web-app

**LoadBalancer**: In case of multiple pods are created and want to access them using one URL/IP and supports distributed load across the multiple PODs/Nodes.



**SAMPLE YAML FILE:**

apiVersion: v1

kind: Service

metadata:

  name: my-service

spec:

  type: LoadBalancer

  selector:

    app: my-app

  ports:

    - port: 80

      targetPort: 8080

**Namespace (short name = ns):**

namespace is a virtual cluster or logical partition within a cluster that provides a way to organize and isolate resources. It allows multiple teams or projects to share the same physical cluster while maintaining resource separation and access control.

1. **Resource Organization:** Namespaces provide a way to organize and categorize resources such as pods, services, deployments, and more. It helps to logically group related resources and provides better manageability.
2. **Resource Isolation:** Namespaces provide resource isolation by creating separate virtual clusters within a physical cluster. Resources within a namespace are scoped to that namespace and cannot directly interact with resources in other namespaces.
3. **Resource Quotas:** Namespaces can have resource quotas defined to limit the amount of compute resources (CPU, memory) that can be consumed by the resources within the namespace. This ensures that one namespace cannot monopolize all available resources in the cluster.

**# To create a namespace:**

$ kubectl create namespace <namespace-name>

$ kubectl create ns my-bank

**# To switch to a specific namespace: (make this as default type)**

$ kubectl config set-context --current --namespace=<namespace-name>

**# To list all namespaces:**

$ kubectl get namespaces

**# To get resources within a specific namespace:**

$ kubectl get <resource-type> -n <namespace-name>

$ kubectl get deploy -n my-bank

$ kubectl get deploy --namespace my-bank

$ kubectl get all --namespace my-bank

**# To delete a namespace and all associated resources:**

$ kubectl delete namespace <namespace-name>

$ kubectl delete ns my-bank

**# We can even create using YAML file**

$ sudo nano ns-test.yml

$ kubectl apply -f ns-test.yml

apiVersion: v1

kind: Namespace

metadata:

  name: my-bank

**# We need to provide namespace under metadata tag in every resource to indicate namespace.**

**Example Pod with Namespace**

$ sudo nano test-pod.yml

$ kubectl apply -f test-pod.yml

apiVersion: v1

kind: Pod

metadata:

  name: my-pod

  namespace: my-bank

spec:

  containers:

  - name: my-container

    image: nginx:1.23

**# Check Pods from above namespace**

$ kubectl get po -n my-bank

$ kubectl get po --namespace my-bank

**# command to create a new resource under a given namespace**

$ kubectl create deployment my-dep --image=busybox **--namespace my-bank**

**# View all Resources(pod, deployment, service..etc) under a given namespace**

$ kubectl get all -n my-bank

**ConfigMap (short name = cm):**

ConfigMap is an API object used to store configuration data as *key-value pairs*. It provides a way to separate configuration from the application code, making it easier to manage and update configurations *without rebuild and redeploying* the application.

1. **Data Storage:** A ConfigMap stores configuration data in key-value pairs. The data can include environment variables, command-line arguments, configuration files, or any other configuration values.
2. **Decoupling Configuration:** By using ConfigMaps, you can separate configuration data from the application code. This allows you to manage and update configuration independently, without modifying or redeploying the application.
3. **Multiple Use Cases:** ConfigMaps can be used to configure applications running in pods, as well as provide configuration data to other Kubernetes objects such as Deployments, StatefulSets, and DaemonSets.
4. **Immutability:** ConfigMaps are immutable, meaning they cannot be modified once created. To update configuration data, you need to create a new ConfigMap with the updated values. We need to specify **immutable: true** inside configuration file.

To create a ConfigMap in Kubernetes, you can use either the `kubectl` command-line tool or a YAML configuration file. Here's an example of creating a ConfigMap using a YAML file:

**$ sudo nano config.yml**

$ kubectl apply -f config.yml

apiVersion: v1

kind: ConfigMap

metadata:

  name: my-configmap

immutable: true

data:

  key1: value1

  key2: value2

**In this example:**

* The ConfigMap name is specified as `my-configmap`.
* The configuration data is defined within the `data` section, where `key1` and `key2` are the keys, and `value1` and `value2` are the corresponding values.

**# Creating a ConfigMap from literal values**

$ kubectl create configmap my-configmap --from-literal=key1=value1 --from-literal=key2=value2

**# Creating a ConfigMap from a file:**

$ kubectl create configmap my-configmap --from-file=app.properties

**# To View all ConfigMaps**

$ kubectl get cm

$ kubectl get configmap

$ kubectl get cm <config-map-name>

**# To describe ConfigMap**

$ kubectl describe configmap my-configmap

**# to Delete a ConfigMap**

$ kubectl delete cm my-configmap

**Secrets (short name = secret):**

Secrets are a way to securely store and manage sensitive information, such as passwords, tokens, and API keys. They are like ConfigMaps but are specifically designed to handle confidential or sensitive data.

1. **Data Storage:** Secrets store sensitive information as key-value pairs, just like ConfigMaps. However, Secrets *encode the data in base64* format to provide a basic level of obfuscation.
2. **Multiple Use Cases:** Secrets can be used to store credentials for accessing external services, API keys for integrations, TLS certificates, SSH keys, and any other confidential information required by applications running in Kubernetes.
3. **Immutability:** Like ConfigMaps, Secrets are also immutable. Once created, they cannot be modified directly. To update Secrets, you need to create a new Secret with the updated data. We need to specify **immutable: true** inside configuration file.

To create a Secret in Kubernetes, you can use the `kubectl` command-line tool or define a YAML configuration file. Here's an example of creating a Secret using a YAML file:

$ sudo nano secrets.yml

apiVersion: v1

kind: Secret

metadata:

  name: my-secret

type: Opaque

data:

  username: dXNlcm5hbWU=

  password: cGFzc3dvcmQ=

$ kubectl apply -f secrets.yml

**In this example:**

* The Secret name is specified as `my-secret`.
* The data is encoded in base64 format. `username` and `password` are the keys, and `dXNlcm5hbWU=` and `cGFzc3dvcmQ=` are the base64-encoded values for the username and password, respectively.

Once the Secret is created, you can reference it in your Kubernetes resources, such as pods or deployments, to securely access the sensitive data stored within the Secret.

**# To create a Secret using literal values**

$ kubectl create secret generic my-secret --from-literal=username=user --from-literal=password=pass

**# To list all the Secrets in a namespace**

$ kubectl get secrets

**# To list all the Secrets in a namespace**

$ kubectl describe secret my-secret

**# To delete a Secret**

$ kubectl delete secret my-secret

# To use ConfigMap and Secret values inside Deployment (or POD) Manifest files, we need to add env: block with kerRef using Resource name and Key.

apiVersion: apps/v1

kind: Deployment

metadata:

  name: myapp-deployment

spec:

  replicas: 1

  selector:

    matchLabels:

      app: myapp

  template:

    metadata:

      labels:

        app: myapp

    spec:

      containers:

        - name: myapp-container

          image: myapp:latest

          env:

            - name: APP\_NAME

              valueFrom:

                configMapKeyRef:

                  name: myapp-config

                  key: app-username

            - name: DB\_PASSWORD

              valueFrom:

                secretKeyRef:

                  name: myapp-secret

                  key: db-password

**Persistent Volumes (PV) and Persistent Volume Claims (PVC)**

Persistent Volumes (PV) and Persistent Volume Claims (PVC) are used to manage and provide persistent storage for applications running in containers.

**Persistent Volume (PV):**

A Persistent Volume (PV) is a representation of a physical storage resource in the cluster. It abstracts the underlying storage implementation and provides a way to manage and consume storage in a platform-independent manner. PVs are created and ***managed by administrators***.

**Key characteristics of a PV include:**

* **Capacity**: The amount of storage available in the PV.
* **Access modes**: The supported access modes for the PV (e.g., ReadWriteOnce, ReadOnlyMany, ReadWriteMany).
* Persistent Volume **Reclaim Policy**: Specifies what happens to the data when a PV is released. It can be set to "Retain", "Delete" or "Recycle".
* **Storage class**: An optional parameter that defines the provisioner and parameters for **dynamically** provisioning PVs.

**Persistent Volume Claim (PVC):**

A Persistent Volume Claim (PVC) is a **request for storage by a user or a deployment**. It acts as a request for a specific amount of storage with certain access modes. PVCs are created and managed by ***users or developers***.

**Key characteristics of a PVC include:**

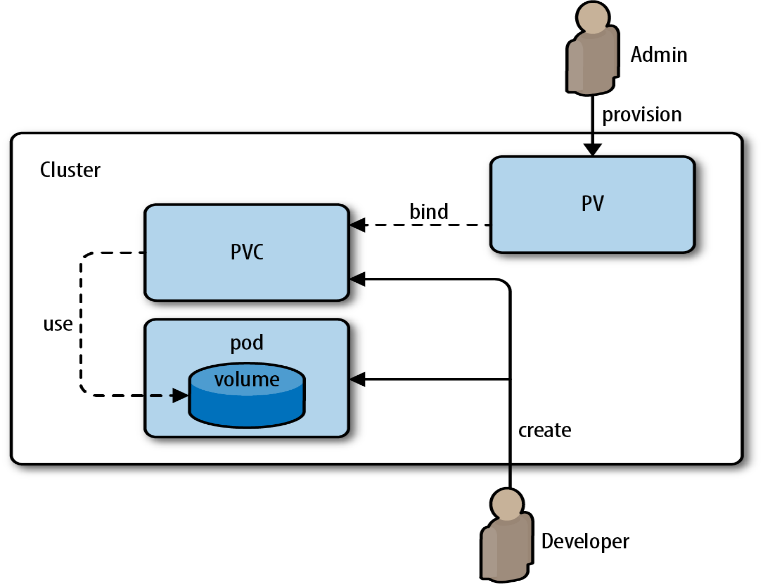
* **Capacity request**: The amount of storage requested by the PVC.
* **Access modes**: The required access modes for the PVC.
* **Storage class**: An optional parameter that specifies the desired storage class to fulfill the PVC. It is used to dynamically provision a matching PV.

1. The administrator creates a PV, which represents a physical storage resource in the cluster. The PV can be provisioned statically or dynamically.

2. The user or developer creates a PVC to request storage for their application. The PVC specifies the desired storage capacity and access modes.

3. When a PVC is created, the Kubernetes control plane looks for a matching PV based on the requested capacity, access modes, and storage class (if specified). If a matching PV is found, the PVC is bound to that PV.

4. Once the PVC is bound to a PV, the user or developer can use the PVC in their application's deployment configuration.



**Some common storage class values that are often used:**

**1. `standard`:** This is a commonly used storage class that represents the default or standard storage provisioned by the cluster. It usually provides balanced performance and can be used for general-purpose storage.

**2. `gp2` (Amazon EBS):** This storage class is specific to Amazon Elastic Block Store (EBS) volumes on AWS. It provides General Purpose SSD (gp2) volumes with a balance of price and performance.

**3. `io1` (Amazon EBS):** This storage class is also specific to Amazon EBS volumes on AWS. It provides provisioned IOPS (io1) volumes, which are designed for applications that require higher performance and predictable I/O.

**4. `ssd` or `premium`:** These storage class names are often used by cloud providers or storage vendors to represent solid-state drive (SSD) storage with higher performance characteristics.

**5. `hdd` or `standard`:** These storage class names are often used by cloud providers or storage vendors to represent hard disk drive (HDD) storage with lower performance characteristics but higher storage capacity.

**6. `local`:** This storage class is used to represent local storage attached directly to the worker nodes in your cluster. It can be useful for certain types of workloads that benefit from low-latency access to local storage.

**Here are the possible values for `accessModes`:**

**1. `ReadWriteOnce` (RWO):** This access mode allows the volume to be mounted as read-write by a single node (Pod) in the cluster. It is typically used for scenarios where the volume should be exclusively mounted and written to by a single node at a time.

**2. `ReadOnlyMany` (ROX):** This access mode allows the volume to be mounted as read-only by multiple nodes (Pods) simultaneously. It is useful for scenarios where multiple Pods need read-only access to the same data, such as for sharing configuration files or static content.

**3. `ReadWriteMany` (RWX):** This access mode allows the volume to be mounted as read-write by multiple nodes (Pods) simultaneously. It enables multiple Pods to read from and write to the same volume concurrently. However, not all storage solutions support this access mode. It's important to ensure that your storage provisioner supports RWX if you plan to use it.

**EXAMPLE PV and PVC YAML Files:**

$ sudo nano pv-test.yml

apiVersion: v1

kind: PersistentVolume

metadata:

  name: mysql-pv

spec:

  capacity:

    storage: 5Gi

  accessModes:

    - ReadWriteOnce

  persistentVolumeReclaimPolicy: Retain

  storageClassName: standard

  hostPath:

    path: /home/ec2-user/mysql/data

$ kubectl apply -f pv-test.yml

**# more commands about PV**

$ kubectl get persistentvolume

$ kubectl get pv

$kubectl get pv -o wide

$kubectl get pv -o json

$kubectl get pv -o yaml

$kubectl describe pv mysql-pv

$ kubectl delete pv mysql-pv

**$ sudo nano pvc-test.yml**

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

  name: mysql-pvc

spec:

  accessModes:

    - ReadWriteOnce

  resources:

    requests:

      storage: 5Gi

  storageClassName: standard

$ kubectl apply -f pvc-test.yml

$ kubectl get pvc

$ kubectl describe pvc mysql-pvc

$ kubectl delete pvc mysql-pvc

**TASK: Deployment of MySQL using PV,PVC and Secret:**

apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

  name: standard

provisioner: kubernetes.io/hostpath

---

apiVersion: v1

kind: PersistentVolume

metadata:

  name: mysql-pv

spec:

  capacity:

    storage: 5Gi

  accessModes:

    - ReadWriteOnce

  persistentVolumeReclaimPolicy: Retain

  storageClassName: standard

  hostPath:

    path: /home/ec2-user/mysql/data

---

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

  name: mysql-pvc

spec:

  accessModes:

    - ReadWriteOnce

  resources:

    requests:

      storage: 5Gi

  storageClassName: standard

---

apiVersion: v1

kind: Secret

metadata:

  name: mysql-secret

type: Opaque

data:

  password: UmFnaHVAMTIzNA==

---

apiVersion: apps/v1

kind: Deployment

metadata:

  name: mysql-deployment

spec:

  replicas: 1

  selector:

    matchLabels:

      app: mysql

  template:

    metadata:

      labels:

        app: mysql

    spec:

      containers:

        - name: mysql

          image: mysql:latest

          env:

            - name: MYSQL\_ROOT\_PASSWORD

              valueFrom:

                secretKeyRef:

                  name: mysql-secret

                  key: password

          ports:

            - containerPort: 3306

          volumeMounts:

            - name: mysql-persistent-storage

              mountPath: /var/lib/mysql

      volumes:

        - name: mysql-persistent-storage

          persistentVolumeClaim:

            claimName: mysql-pvc

---

apiVersion: v1

kind: Service

metadata:

  name: mysql-service

spec:

  selector:

    app: mysql

  ports:

    - protocol: TCP

      port: 3306

      targetPort: 3306

**Try these incase of any issue:**

minikube stop

minikube delete

docker system prune --all

sudo sync && echo 3 | sudo tee /proc/sys/vm/drop\_caches

docker volume ls

(if exist any delete)

minikube start

**StatefulSet (short name=sts)**

A StatefulSet in Kubernetes is a higher-level controller that manages the deployment and scaling of stateful applications. It provides guarantees about the ordering and uniqueness of Pods, which makes it suitable for managing stateful applications ***like databases, message queues, and distributed storage.***

Compared to a Deployment, which is commonly used for stateless applications, a StatefulSet offers the following features:

**1. Stable network identity:** Each Pod in a StatefulSet receives a unique and stable network identity. The Pods are named in a predictable manner with an index appended to the Pod name, such as `myapp-0`, `myapp-1`, and so on. This enables other applications or services to consistently locate and communicate with individual Pods.

**2. Ordered deployment and scaling:** StatefulSets ensure that Pods are created or scaled in a predictable order. This is important for stateful applications that may have dependencies on other Pods or require sequential operations during startup or shutdown.

**3. Stable storage:** StatefulSets work seamlessly with PersistentVolumes and PersistentVolumeClaims, providing stable storage for each Pod in the set. Each Pod in a StatefulSet can have its own unique volume that persists even if the Pod is rescheduled or restarted.

**4. Stateful Pod termination:** StatefulSets support ordered and graceful termination of Pods. When a Pod is deleted or scaled down, StatefulSets ensure that the Pods are terminated in reverse order, allowing the application to gracefully handle any necessary cleanup or data replication.

apiVersion: apps/v1

kind: StatefulSet

metadata:

  name: myapp-statefulset

spec:

  replicas: 3

  serviceName: myapp-service

  selector:

    matchLabels:

      app: myapp

  template:

    metadata:

      labels:

        app: myapp

    spec:

      containers:

        - name: myapp-container

          image: nginx:latest

          ports:

            - containerPort: 80

          volumeMounts:

            - name: myapp-volume

              mountPath: /var/www/html

  volumeClaimTemplates:

    - metadata:

        name: myapp-volume

      spec:

        accessModes: [ "ReadWriteOnce" ]

        storageClassName: standard

        resources:

          requests:

            storage: 1Gi

**In this example:**

* `replicas` specifies the desired number of Pods to be created.
* `serviceName` sets the name of the associated Headless Service that provides network identity for the Pods.
* The `selector` field defines the labels that are used to match the Pods controlled by the StatefulSet.
* `template` specifies the Pod template used to create new Pods.
* The `volumeClaimTemplates` section defines the PersistentVolumeClaim template that will be used to dynamically create individual PersistentVolumeClaims for each Pod.

**Here are some common `kubectl` commands used to manage StatefulSets in Kubernetes:**

**1. Create or apply a StatefulSet:**

$ kubectl apply -f statefulset.yaml

This command creates or updates a StatefulSet defined in the `statefulset.yaml` file.

**2. Get information about StatefulSets:**

$ kubectl get statefulsets

$ kubectl describe statefulset <statefulset-name>

The first command lists all StatefulSets in the current namespace, while the second command provides detailed information about a specific StatefulSet.

**3. Scale the number of replicas in a StatefulSet:**

$ kubectl scale statefulset <statefulset-name> --replicas=<desired-replicas>

This command scales the number of replicas in a StatefulSet to the specified `<desired-replicas>`.

**4. Update a StatefulSet:**

$ kubectl edit statefulset <statefulset-name>

This command opens the StatefulSet definition in the default editor, allowing you to make modifications. Save and exit the editor to apply the changes.

**5. Delete a StatefulSet:**

$ kubectl delete statefulset <statefulset-name>

This command deletes the StatefulSet and its associated Pods, but the PersistentVolumes and PersistentVolumeClaims are not automatically deleted.

**6. Check the status of StatefulSet Pods:**

$ kubectl get pods -l <selector-label-key>=<selector-label-value> -w

The `-l` flag is used to filter Pods based on a label selector that matches the StatefulSet's selector. The `-w` flag provides a continuous stream of updates, allowing you to monitor the status of the Pods.

**7. Delete StatefulSet Pods (e.g., for scaling down):**

$ kubectl delete pod <pod-name>

This command deletes a specific Pod belonging to a StatefulSet. The StatefulSet controller will automatically recreate the Pod to ensure Replicas Count.

**DaemonSet (short name=ds)**

DaemonSet is a type of controller that ensures a specific *Pod runs on every node* within a cluster. It is used for deploying system daemons or *monitoring agents* that need to be present on every node.

**Key characteristics of a DaemonSet:**

**1. One Pod per node:** A DaemonSet guarantees that one instance of the specified Pod runs on each node within the cluster.

**2. Automatic creation and deletion:** As nodes are added or removed from the cluster, the DaemonSet automatically manages the creation or deletion of Pods to maintain the desired state.

**3. Rolling updates:** DaemonSets support rolling updates, allowing you to update the Pod template and roll out the changes to the Pods one by one, ensuring continuous availability.

DaemonSets are commonly used for various system-level tasks, such as log collection, monitoring, network plugins, or other background processes that require node-level execution.

Example YAML configuration for a DaemonSet that deploys a simple Fluentd log collector on each node:

$ sudo nano ds.yml

apiVersion: apps/v1

kind: DaemonSet

metadata:

  name: fluentd-daemonset

spec:

  selector:

    matchLabels:

      app: fluentd

  template:

    metadata:

      labels:

        app: fluentd

    spec:

      containers:

        - name: fluentd

          image: fluentd:v1.11.5

          resources:

            limits:

              memory: 200Mi

          volumeMounts:

            - name: varlog

              mountPath: /var/log

            - name: varlibdockercontainers

              mountPath: /var/lib/docker/containers

              readOnly: true

      volumes:

        - name: varlog

          hostPath:

            path: /var/log

        - name: varlibdockercontainers

          hostPath:

            path: /var/lib/docker/containers

**In this example:**

* The `selector` field specifies the label selector used to identify the nodes where the Pods should be scheduled.
* The `template` section defines the Pod template used to create the Pods.
* The `containers` section specifies the container within the Pod, including the container name, image, resource limits, and any required volume mounts.
* The `volumes` section defines the volumes that will be mounted inside the Pods. In this case, it mounts the host's `/var/log` and `/var/lib/docker/containers` directories.

This will create the DaemonSet, and the specified Pod will be deployed on each node in the cluster.

Here are some common `kubectl` commands used to manage DaemonSets in Kubernetes:

**1. Create or apply a DaemonSet:**

$ kubectl apply -f daemonset.yaml

This command creates or updates a DaemonSet defined in the `daemonset.yaml` file.

**2. Get information about DaemonSets:**

$ kubectl get ds

$ kubectl get daemonsets

$ kubectl describe daemonset <daemonset-name>

The first command lists all DaemonSets in the current namespace, while the second command provides detailed information about a specific DaemonSet.

**3. Scale the number of Pods in a DaemonSet:**

DaemonSets do not have a replica count like Deployments or StatefulSets since they are designed to have *one Pod per node*. To scale a DaemonSet, you typically update the Pod template and apply the changes.

**4. Update a DaemonSet:**

$ kubectl edit daemonset <daemonset-name>

This command opens the DaemonSet definition in the default editor, allowing you to make modifications. Save and exit the editor to apply the changes.

**5. Delete a DaemonSet:**

$ kubectl delete ds <daemonset-name>

$ kubectl delete daemonset <daemonset-name>

This command deletes the DaemonSet and its associated Pods from all nodes in the cluster.

**6. Delete DaemonSet Pods:**

$ kubectl delete pod <pod-name>

This command deletes a specific Pod belonging to a DaemonSet. The DaemonSet controller will automatically recreate the Pod on the same node.

**Resource Limits in Pods**

Resource limits are used to control and allocate resources (such as CPU and memory) to containers running within a cluster. Resource limits play a crucial role in maintaining stability, preventing resource contention, and ensuring fair allocation of resources among different workloads.

Here are some key aspects of resource limits in Kubernetes:

**1. CPU limits**: CPU limits define the *maximum amount of CPU* time that a container can use. They are specified using a decimal value, representing the number of CPU cores or a percentage of the total CPU available in the cluster. For example, setting a CPU limit to "500m" means the container can use up to 500 milliCPU units (or 0.5 CPU cores), and setting a limit to "50%" means the container can use up to half of the total CPU resources.

**2. Memory limits:** Memory limits define the *maximum amount of memory* that a container can use. They are specified using a decimal value, followed by a memory unit (e.g., Mi, Gi) to represent bytes, mebibytes, or gibibytes. For example, setting a memory limit to "512Mi" means the container can use up to 512 mebibytes of memory.

**Example: resource-limits.yml**

**apiVersion: apps/v1**

**kind: Deployment**

**metadata:**

**name: my-app-deployment**

**spec:**

**replicas: 3**

**selector:**

**matchLabels:**

**app: my-app**

**template:**

**metadata:**

**labels:**

**app: my-app**

**spec:**

**containers:**

**- name: my-app-container**

**image: javabyraghu/maven-web-app**

**resources:**

**limits:**

**cpu: "1"**

**memory: "512Mi"**

**requests:**

**cpu: "500m"**

**memory: "256Mi"**

**In this example:**

* The ***resources*** section within the container configuration sets the resource limits and requests.
* The ***limits*** field sets the ***maximum*** CPU and memory that the container can use (1 CPU core and 512MiB of memory in this case).
* The ***requests*** field sets the ***initial*** CPU and memory resources that the container requires (500 milliCPU units and 256MiB of memory in this case).

**ResourceQuota (shortname=quota Namespace Based)**

ResourceQuota is a resource management feature that allows you to specify limits and constraints on resource usage within a namespace. It helps you allocate and control resources for applications and users in a more granular manner.

**1. Namespace scope:** ResourceQuota is applied at the namespace level, which means you can define different quotas for each namespace in your cluster. Each namespace can have its own ResourceQuota configuration.

**2. Resource types:** ResourceQuota can be defined for various resource types, including CPU, memory, persistent volume claims (PVCs), pods, services, and more. You can set limits on these resources individually or collectively.

**3. Resource limits:** ResourceQuota allows you to specify resource limits for each resource type. The limits can be defined in absolute values or as a percentage of the total cluster capacity. For example, you can set a CPU limit of 4 cores or 50% of the total cluster CPU capacity.

**4. Resource requests:** ResourceQuota also enables you to set constraints on resource requests. Resource requests define the amount of resources that a pod requires to run. By setting request limits in a ResourceQuota, you can ensure that pods within the namespace don't consume more resources than specified.

**5. Quota enforcement:** Once a ResourceQuota is defined, Kubernetes enforces the limits and constraints specified within the namespace. If a pod or user exceeds the defined limits, attempts to create new resources that breach the quotas will be rejected.

**6. Viewing and modifying quotas:** You can use the `kubectl` command-line tool or Kubernetes API to view and modify ResourceQuota settings. For example, you can run `kubectl describe resourcequota` or `kubectl edit resourcequota` to see and edit the quotas.

$ sudo nano quota-test.yml

**apiVersion: v1**

**kind: ResourceQuota**

**metadata:**

**name: my-quota**

**namespace: production-space**

**spec:**

**hard:**

**cpu: "4"**

**memory: 4Gi**

**pods: "10"**

**persistentvolumeclaims: "5"**

**services.loadbalancers: "2"**

$ kubectl apply -f quota-test.yml

$ kubectl get quota

# we can get from a specific namespace

$ kubectl get resourcequota -n production-space

In this example, the ResourceQuota named "my-quota" defines the following limits:

- Maximum of 4 CPU cores

- Maximum of 4GiB of memory

- Maximum of 10 pods

- Maximum of 5 persistent volume claims (PVCs)

- Maximum of 2 load balancer services

**`kubectl` commands:**

**1. Create a ResourceQuota:**

$ kubectl create -f resourcequota.yaml

Replace `resourcequota.yaml` with the file path or URL of the ResourceQuota manifest that you want to create.

**2. Get information about ResourceQuotas:**

$ kubectl get resourcequotas

This command retrieves information about all the ResourceQuotas in the current namespace.

$ kubectl describe resourcequota <resourcequota-name>

This command provides detailed information about the specified ResourceQuota.

**3. Edit a ResourceQuota:**

$ kubectl edit resourcequota <resourcequota-name>

This command opens the ResourceQuota in the default editor specified by your environment and allows you to modify the configuration.

**4. Delete a ResourceQuota:**

$ kubectl delete resourcequota <resourcequota-name>

This command removes the specified ResourceQuota from the cluster.

**LimitRange (shortname=limits Namespace Based)**

LimitRange is a resource management feature that allows you to define limits and defaults for compute resources (such as CPU and memory) at the container level within a namespace. LimitRange ensures that *containers running in a namespace* adhere to the specified resource constraints.

**1. Namespace scope:** LimitRange is applied at the namespace level, allowing you to define different LimitRange configurations for each namespace in your cluster. Each namespace can have its own LimitRange settings.

**2. Resource types:** LimitRange can be defined for various resource types, including CPU, memory, and storage. You can set limits for these resources individually or collectively.

**3. Resource limits:** LimitRange enables you to set minimum and maximum resource limits for each resource type. You can specify absolute values or use the "default" value to set a default limit if the container doesn't define its own limits. If a container exceeds the specified limits, the creation or update of the container will be rejected.

**4. Default requests:** In addition to limits, LimitRange allows you to set default requests for resources. Resource requests define the amount of resources that a container requires to run. If a container doesn't specify its own requests, the default value defined in the LimitRange will be applied.

**apiVersion: v1**

**kind: LimitRange**

**metadata:**

**name: my-limit-range**

**spec:**

**limits:**

**- default:**

**cpu: "500m"**

**memory: "256Mi"**

**defaultRequest:**

**cpu: "200m"**

**memory: "128Mi"**

**type: Container**

In the context of LimitRange in Kubernetes, the `type` field specifies the scope to which the resource limits apply. The `type` field accepts the following possible values:

**1. Container:** This is the most common value for the `type` field. It indicates that the resource limits defined in the LimitRange apply at the container level. It means the limits and defaults specified within the LimitRange configuration are specific to individual containers running within the namespace.

**2. Pod:** When the `type` field is set to "Pod", the resource limits defined in the LimitRange apply at the pod level. This means that the limits and defaults specified within the LimitRange are shared by all containers within the same pod. Using this type allows for setting consistent resource limits for all containers within a pod.

**3. PersistentVolumeClaim:** If the `type` field is set to "PersistentVolumeClaim", the resource limits defined in the LimitRange apply to the persistent volume claims (PVCs) within the namespace. It allows setting constraints and limits on the storage resources requested by PVCs.

**`kubectl` commands :**

**1. Create a LimitRange:**

$ kubectl create -f limitrange.yaml

Replace `limitrange.yaml` with the file path or URL of the LimitRange manifest that you want to create.

**2. Get information about LimitRanges:**

$ kubectl get limitranges

This command retrieves information about all the LimitRanges in the current namespace.

$ kubectl describe limitrange <limitrange-name>

This command provides detailed information about the specified LimitRange.

**3. Edit a LimitRange:**

$ kubectl edit limitrange <limitrange-name>

This command opens the LimitRange in the default editor specified by your environment and allows you to modify the configuration.

**4. Delete a LimitRange:**

$ kubectl delete limitrange <limitrange-name>

This command removes the specified LimitRange from the cluster.

**Horizontal Pod Autoscaler (HPA)**

Horizontal Pod Autoscaler (HPA) is a feature that automatically adjusts the number of replica pods in a deployment, replication controller, or replica set based on the ***observed CPU utilization***. The HPA ensures that the application scales up or down to meet the desired performance and resource utilization targets.

**1. Metrics:** HPA monitors the resource utilization of the target pods, typically focusing on CPU utilization. It uses metrics provided by the Metrics Server, which collects resource usage data from the Kubernetes API server.

**2. Target resource:** HPA is configured to scale based on a specific resource metric, typically CPU utilization, but it can also use custom metrics. CPU utilization is expressed as a percentage of the total available CPU resources across all nodes.

**3. Scaling behaviour:** HPA adjusts the number of replica pods based on the observed resource utilization compared to the defined target. If the utilization exceeds the target, HPA increases the number of replicas. If the utilization is below the target, HPA reduces the number of replicas.

**4. Integration with deployments:** HPA works in conjunction with deployments, replication controllers, or replica sets, where it manages the scaling of pods. It adjusts the replica count within the specified minimum and maximum limits.

**# View minikube add-ons**

$ minikube addons list

**# Add Metric servers to minikube to view load details**

$ minikube addons enable metrics-server

$ sudo nano test-deploy.yml

**apiVersion: apps/v1**

**kind: Deployment**

**metadata:**

**name: php-apache**

**spec:**

**selector:**

**matchLabels:**

**run: php-apache**

**template:**

**metadata:**

**labels:**

**run: php-apache**

**spec:**

**containers:**

**- name: php-apache**

**image: registry.k8s.io/hpa-example**

**ports:**

**- containerPort: 80**

**resources:**

**limits:**

**cpu: 500m**

**requests:**

**cpu: 200m**

**---**

**apiVersion: v1**

**kind: Service**

**metadata:**

**name: php-apache**

**labels:**

**run: php-apache**

**spec:**

**ports:**

**- port: 80**

**selector:**

**run: php-apache**

$ kubectl apply -f test-deploy.yml

**# create HPA over deployment**

$ kubectl autoscale deployment php-apache --cpu-percent=50 --min=1 --max=10

**# increase load on Service (use ctrl+c to stop load)**

$ kubectl run -i --tty load-generator --rm --image=busybox:1.28 --restart=Never -- /bin/sh -c "while sleep 0.01; do wget -q -O- http://php-apache; done"

**# Watch HAP Details (Result may be slow in Minikube)**

$ kubectl get hpa php-apache --watch

**# To view generated HPA YAML file**

$ kubectl get hpa php-apache -o yaml > /tmp/hpa-v2.yaml

In this example, the HPA named "php-apache" is configured to scale the pods in the deployment named "nginx-deployment". The HPA is set to maintain an average CPU utilization of 50%. The minimum number of replicas is set to 2, and the maximum number of replicas is set to 10.

**`kubectl` commands:**

**1. Create an HPA:**

$ kubectl create -f hpa.yaml

Replace `hpa.yaml` with the file path or URL of the HPA manifest that you want to create.

**2. Get information about HPAs:**

$ kubectl get hpa

This command retrieves information about all the HPAs in the current namespace.

$ kubectl describe hpa <hpa-name>

This command provides detailed information about the specified HPA.

**3. Edit an HPA:**

$ kubectl edit hpa <hpa-name>

This command opens the HPA in the default editor specified by your environment and allows you to modify the configuration.

**4. Delete an HPA:**

$ kubectl delete hpa <hpa-name>

This command removes the specified HPA from the cluster.

**Blue-Green Deployment (use kubeadm only)**

Blue-Green Deployment is a deployment strategy used in Kubernetes to achieve zero-downtime updates by creating two identical environments, one representing the current production environment (blue) and the other representing the new version or release (green). Here's an overview of the steps involved in a Blue-Green Deployment:

**1. Set up the blue environment:** Initially, you will have your production environment running, which represents the blue environment.

**2. Create the green environment:** Deploy the new version or release of your application in a separate Kubernetes namespace or cluster, which represents the green environment. This environment should be identical to the blue environment.

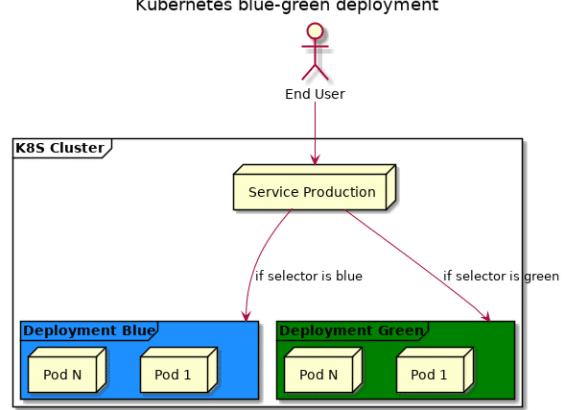
**3. Perform testing and verification:** Conduct thorough testing and verification in the green environment to ensure that the new version is working as expected and meets the necessary quality criteria.

**4. Switch traffic to the green environment**: Once the green environment has been successfully tested, you can route a portion of the production traffic to the green environment. This can be achieved through various methods such as load balancers, ingress controllers, or DNS changes.

**5. Monitor and validate:** Monitor the green environment to ensure that it handles the traffic correctly and performs as expected. Validate that there are no issues or errors occurring in the green environment.

**6. Gradually shift traffic:** If the green environment performs well and meets expectations, gradually increase the traffic to the green environment while simultaneously reducing traffic to the blue environment.

**7. Complete the cutover:** Once all traffic has been shifted to the green environment and you are confident that it is stable and functioning correctly, you can decommission the blue environment.



**Blue Environment Setup:**

1. Create Blue Environment Deployment and configure service for that.

$ sudo nano blue-deploy.yml

apiVersion: apps/v1

kind: Deployment

metadata:

  name: k8s-boot-demo-deployment-blue

spec:

  replicas: 3

  strategy:

    type: RollingUpdate

  selector:

    matchLabels:

      app: k8s-boot-demo

      version: v1

      color: blue

  template:

    metadata:

      labels:

        app: k8s-boot-demo

        version: v1

        color: blue

    spec:

      containers:

        - name: k8s-boot-demo

          image: javabyraghu/java-web-app:latest

          imagePullPolicy: Always

          ports:

            - containerPort: 8080

$ kubectl apply -f blue-deploy.yml

$ sudo nano service-live.yml

apiVersion: v1

kind: Service

metadata:

  name: k8s-boot-live-service

spec:

  type: NodePort

  selector:

    app: k8s-boot-demo

    version: v1

  ports:

    - name: app-port-mapping

      protocol: TCP

      port: 8080

      targetPort: 8080

      nodePort: 30002

$ kubectl apply -f service-live.yml

$ kubectl get svc

After creating the service access our application using below URL

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

2. Configure Green Environment using Deployment and new service file

$ sudo nano green-deploy.yml

apiVersion: apps/v1

kind: Deployment

metadata:

  name: k8s-boot-demo-deployment-green

spec:

  replicas: 3

  strategy:

    type: RollingUpdate

  selector:

    matchLabels:

      app: k8s-boot-demo

      version: v2

      color: green

  template:

    metadata:

      labels:

        app: k8s-boot-demo

        version: v2

        color: green

    spec:

      containers:

        - name: k8s-boot-demo

          image: javabyraghu/maven-web-app:latest

          imagePullPolicy: Always

          ports:

            - containerPort: 8080

$ kubectl apply -f green-deploy.yml

$ sudo nano service-prepod.yml

apiVersion: v1

kind: Service

metadata:

  name: k8s-boot-service-preprod

spec:

  type: NodePort

  selector:

    app: k8s-boot-demo

    version: v2

  ports:

    - name: app-port-mapping

      protocol: TCP

      port: 8080

      targetPort: 8080

      nodePort: 30092

$ kubectl apply -f service-prepod.yml

$ kubectl get svc

Access the application using pre-prod service

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

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

**Q) How can we make Green PODS as Live?**

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

$ kubectl apply -f service-live.yml

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

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

**K8s Dashboard (use kubeadm)**

The Kubernetes Dashboard is a web-based graphical user interface (GUI) that provides a visual representation of your Kubernetes cluster. It allows you to manage and monitor your cluster's resources, deployments, pods, services, and more. You can use the dashboard to view and interact with various aspects of your Kubernetes environment.

**# Apply Dashboard Configuration**

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

**# View dashboard pods and service**

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

$ kubectl -n kubernetes-dashboard get svc

**# Edit k8s dashboard service and change it to NodePort**

$ kubectl -n kubernetes-dashboard edit svc kubernetes-dashboard

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

**# Check in which node kubernetes-dashboard POD is running**

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

**# Access k8s web ui dashboard using below URL**

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

**#create admin user with below yml**

$ vi create-user.yml

**apiVersion: v1**

**kind: ServiceAccount**

**metadata:**

**name: admin-user**

**namespace: kubernetes-dashboard**

$ kubectl apply -f create-user.yml

**# create cluster role binding**

$ vi cluster-role-binding.yml

**apiVersion: rbac.authorization.k8s.io/v1**

**kind: ClusterRoleBinding**

**metadata:**

**name: admin-user**

**roleRef:**

**apiGroup: rbac.authorization.k8s.io**

**kind: ClusterRole**

**name: cluster-admin**

**subjects:**

**- kind: ServiceAccount**

**name: admin-user**

**namespace: kubernetes-dashboard**

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

**# Get the bearer token**

$ kubectl -n kubernetes-dashboard create token admin-user

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