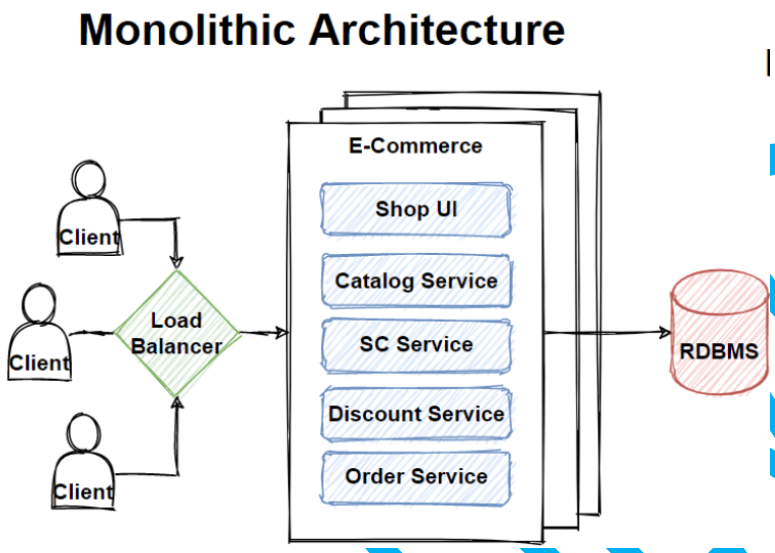
Prepared by PPNREDDY .... 😊 😊 😊

Monolithic application

A **monolithic application** is a software architecture pattern in which all components of the application are integrated and tightly coupled into a single codebase. In this type of architecture, the entire application is built, deployed, and scaled as a single unit. Monolithic applications often include components such as:

- User interface (UI)
- Business logic
- Database management
- Background processes



Key characteristics of monolithic applications:

1. **Single codebase**: The entire application is contained in one unified codebase.
2. **Tight coupling**: Different parts of the application (UI, backend logic, database, etc.) are closely linked together.
3. **Single deployment unit**: The entire application is deployed as a single entity.
4. **Shared resources**: All components share the same resources like memory, CPU, and storage.
5. **Challenges with scaling**: Scaling a monolithic application usually means scaling the entire application, even if only certain components are under load.

Advantages:

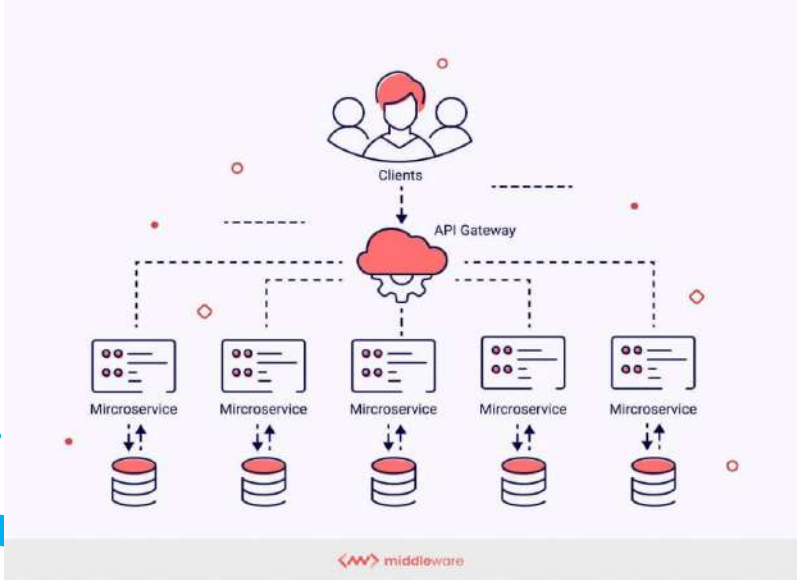
- **Simple deployment**: With only one unit to deploy, managing the application can be straightforward.
- **Easier to develop in the early stages**: For smaller teams or applications, developing a monolith can be simpler.
- **Fewer cross-service issues**: Since all components are in the same codebase, communication between components can be direct and without network-related issues.

Disadvantages:

- **Lack of flexibility**: As the application grows, it can become harder to manage and make changes without affecting other parts of the system.
- **Scaling limitations**: Monoliths can be difficult to scale, as they don't allow individual components to be scaled independently.
- **Slower development**: As the codebase grows, development can slow down due to tightly coupled components.
- **Harder to maintain**: Bugs or issues in one part of the application can impact the entire system.

Microservices architecture

Microservices architecture is a software design pattern where an application is structured as a collection of small, independent services that communicate with each other, typically over HTTP APIs. Each microservice is responsible for a specific function, operates independently, and can be deployed, scaled, and maintained separately from the others.



Key Characteristics of Microservices:

1. **Independent Services**: Each microservice represents a distinct business capability or functionality, such as user authentication, payment processing, or inventory management. They operate independently of each other.
2. **Decentralized Data Management**: Each microservice manages its own database or storage, leading to decentralized data management. This ensures that changes or failures in one service do not directly impact others.
3. **Communication via APIs**: Microservices typically communicate with each other using lightweight protocols like HTTP/REST, gRPC, or message queues, allowing loose coupling between services.

4. **Independent Deployment**: Each microservice can be developed, deployed, and scaled independently of others, enabling continuous delivery and deployment.
5. **Autonomy**: Microservices are usually developed and managed by small, independent teams, which improves the agility of development and maintenance.
6. **Polyglot Programming**: Different microservices can be written in different programming languages or frameworks depending on the needs of the service, allowing teams to choose the best tools for their task.

Advantages of Microservices:

1. **Scalability**: Since each microservice can be scaled independently, this architecture allows fine-grained scalability. You can scale only the services that need it, rather than scaling the entire application.
2. **Fault Isolation**: Failure in one microservice doesn't necessarily bring down the entire application. It improves system resilience.
3. **Flexibility in Technology**: Teams can choose different technologies, programming languages, and databases best suited to the requirements of each microservice.
4. **Faster Development and Deployment**: Microservices enable continuous deployment, where teams can deploy updates to one part of the system without affecting the rest of the application.
5. **Smaller Codebase per Service**: Each service has a smaller, more focused codebase, which is easier to manage, test, and maintain over time.
6. **Easier to Modify and Update**: Microservices architecture allows for quicker adaptation to new business needs. Individual services can be updated or replaced without requiring changes to the whole system.

Challenges of Microservices:

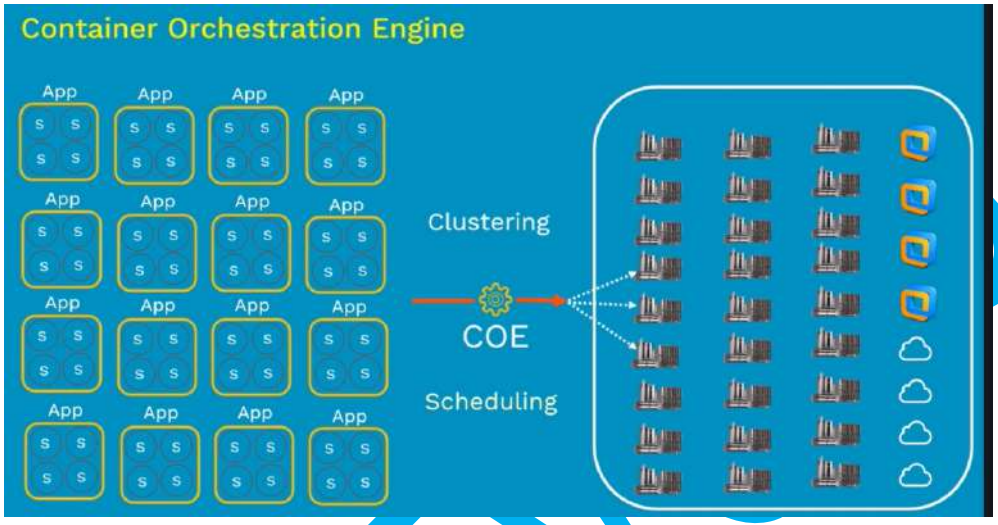
1. **Complexity in Communication**: As the number of microservices grows, managing communication between services becomes complex, requiring mechanisms like API gateways or service discovery.
2. **Increased Operational Overhead**: Microservices require a sophisticated infrastructure, including monitoring, logging, and deployment pipelines for each service, which increases operational complexity.
3. **Distributed Data Management**: Ensuring data consistency across multiple services with separate databases is more challenging compared to a monolithic application.
4. **Latency**: Communication between microservices typically happens over a network, which can introduce latency and potential bottlenecks, especially when services are spread over multiple locations.
5. **Testing and Debugging**: With many independent services interacting over a network, testing and debugging issues across microservices can be more difficult than in a monolithic architecture.

Microservices vs Monolithic:

Feature	Microservices	Monolithic
Code Structure	Small, independent services	Single, unified codebase
Deployment	Independent deployment of each service	Entire application deployed as a single unit
Scalability	Fine-grained, service-specific scaling	Scaling the entire application at once
Fault Tolerance	Fault isolation per service	A failure in one component can affect the whole app
Technology Stack	Polyglot, different services can use different stacks	Typically, a single tech stack for the whole app
Data Management	Decentralized, each service can have its own DB	Centralized, single database

Container Orchestration Engine

A **Container Orchestration Engine** is a platform or tool that automates the deployment, management, scaling, and networking of containerized applications. Containers are lightweight, portable units of software that include everything needed to run an application (code, runtime, libraries, etc.). Container orchestration engines are used to manage multiple containers across different hosts and ensure that applications run efficiently and reliably.



Key Functions of a Container Orchestration Engine:

1. **Container Deployment**: Automates the process of deploying containers across a cluster of servers or nodes.
2. **Scaling**: Automatically adjusts the number of running containers based on application demand.
3. **Load Balancing**: Distributes network traffic across multiple containers to ensure no single container is overwhelmed.
4. **Fault Tolerance**: Detects failures in containers or nodes and restarts or replaces them to maintain application uptime.
5. **Service Discovery**: Manages how different services (containers) communicate with each other, ensuring that the right containers are found and connected.
6. **Resource Management**: Ensures efficient utilization of hardware resources (CPU, memory, etc.) by assigning the necessary resources to each container.
7. **Monitoring and Logging**: Provides tools to monitor container health and resource usage, along with logging features for troubleshooting.

Popular Container Orchestration Engines:

1. **Kubernetes**: One of the most widely used orchestration platforms, Kubernetes automates container deployment, scaling, and operations across clusters. It is open-source and has broad support across cloud providers.
2. **Docker Swarm**: A native clustering and orchestration tool for Docker containers, Swarm allows developers to deploy, scale, and manage containers. It is simpler to set up than Kubernetes but offers less advanced features.

3. **Apache Mesos with Marathon**: Mesos is a cluster manager that can be used to manage containers, and Marathon is its container orchestration tool. Mesos supports both Docker containers and other types of workloads.
4. **Amazon ECS (Elastic Container Service)**: A fully managed container orchestration service offered by AWS. It integrates with other AWS services and provides built-in security and scalability features.
5. **OpenShift**: A Kubernetes-based container platform developed by Red Hat, which offers additional tools for managing containers in enterprise environments.

Importance of Container Orchestration:

Scaling Applications: It helps to automatically scale applications up or down based on usage or traffic.

Multi-Cloud and Hybrid Deployment: Orchestration engines allow containers to be run across different cloud providers or on-premises environments, providing flexibility and portability.

Efficient Resource Utilization: It ensures that resources such as CPU and memory are optimally used by distributing container workloads.

Automation: Automates routine tasks such as deployment, scaling, and recovery, which reduces manual intervention and speeds up application delivery.

Container orchestration engines are critical for managing large-scale containerized applications, particularly in environments with high availability, scalability, and automation needs.

Kubernetes

Kubernetes (often abbreviated as K8s) is an open-source platform for **automating the deployment, scaling, and management of containerized applications**. It was originally developed by Google and is now maintained by the Cloud Native Computing Foundation (CNCF). Kubernetes helps developers manage containers across clusters of machines, providing automated orchestration for large-scale applications.

Key Features of Kubernetes:

1. **Automated Container Deployment**: Kubernetes automates the process of deploying containers, ensuring that they are running on the right nodes and environments.
2. **Scaling**: It automatically scales applications up or down based on demand, either by increasing the number of container instances or reducing them during low usage.
3. **Self-Healing**: Kubernetes automatically monitors the health of containers, restarts failed containers, and reschedules them if nodes go down.
4. **Load Balancing and Service Discovery**: It automatically balances network traffic across containers and provides mechanisms to ensure that containers can find and communicate with each other.
5. **Rolling Updates**: Kubernetes allows for seamless updates to applications by deploying changes gradually across containers without downtime.
6. **Secret and Configuration Management**: Kubernetes securely manages sensitive information like passwords, SSH keys, and other configuration data needed by the containers.
7. **Persistent Storage**: Kubernetes manages and attaches storage resources to containers, making it easy to handle stateful applications.

Benefits of Using Kubernetes:

1. **Portability and Flexibility**: Kubernetes runs on different environments like on-premise servers, private clouds, and public clouds (AWS, Google Cloud, Azure), providing portability for containerized applications.
2. **Efficient Resource Management**: Kubernetes intelligently distributes container workloads across nodes based on resource availability, improving overall resource utilization.
3. **DevOps and CI/CD Integration**: Kubernetes makes continuous integration/continuous deployment (CI/CD) pipelines easier by allowing frequent updates to applications with minimal downtime.
4. **Scalability**: It allows for horizontal scaling of applications by simply adding more nodes or increasing the number of pod replicas as needed.
5. **High Availability and Disaster Recovery**: Kubernetes automatically detects failures and redeploys containers to keep the application running, ensuring high availability.

Use Cases of Kubernetes:

Microservices architecture: Running microservices where each service is deployed as a separate container.

Cloud-Native Applications: Managing scalable and dynamic applications in cloud environments.

CI/CD Pipelines: Managing application deployment and rollback processes.

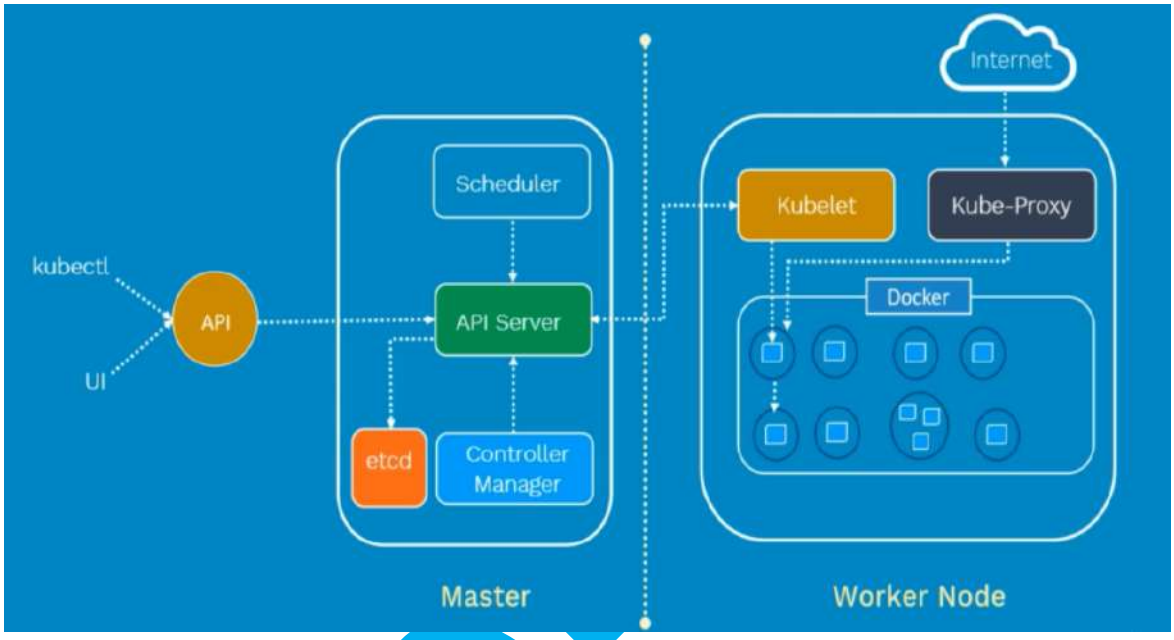
Hybrid Cloud Deployments: Running a combination of on-premises and cloud-based applications.

Kubernetes Vs Docker Swarm

Features	Kubernetes	Docker Swarm
Installation & Cluster Configuration	Installation is complicated; but once setup, the cluster is very strong	Installation is very simple; but cluster is not very strong
GUI	GUI is the Kubernetes Dashboard	There is no GUI
Scalability	Highly scalable & scales fast	Highly scalable & scales 5x faster than Kubernetes
Auto-Scaling	Kubernetes can do auto-scaling	Docker Swarm cannot do auto-scaling
Load Balancing	Manual intervention needed for load balancing traffic between different containers in different Pods	Docker Swarm does auto load balancing of traffic between containers in the cluster
Rolling Updates & Rollbacks	Can deploy Rolling updates & does automatic Rollbacks	Can deploy Rolling updates, but not automatic Rollbacks
Data Volumes	Can share storage volumes only with other containers in same Pod	Can share storage volumes with any other container
Logging & Monitoring	In-built tools for logging & monitoring	3rd party tools like ELK should be used for logging & monitoring

Kubernetes architecture

Kubernetes comes with a client-server architecture. It consists of **master** and **worker nodes**, with the master being installed on a single Linux system and the nodes on many Linux workstations.
The master node, contains the components such as **API Server, controller manager, scheduler, and etcd** database for stage storage. **kubelet** to communicate with the master, the **kube-proxy** for networking, and a container runtime such as Docker to manage containers.



Master Node Components

API Server

APIs make system fast, easy and reliable. Everything in Kubernetes is handled by API server running in system. Every request in system is served by API. API Server is horizontally scalable. So, it can grow and load balance everything as cluster's size increase.
API server is the gatekeeper for entire cluster. If we want to create, delete, update and display any Kubernetes objects it must go the API Server. API server validates and configure API objects such as PODS, Services, Replication Controllers and Deployments. It is responsible for exposing various API's, it exposes API's almost every operation. By using *kubectl* we can interact with API Server.

Scheduler

Scheduler is responsible for physically scheduling PODS across multiple nodes. It's depends up on the constraints mentioned in configuration file scheduler schedules this PODS accordingly. There are hundreds of thousands of pods are running in Kubernetes and scheduling them on different nodes is an important task for the system. scheduler helps system to identified pods which are not assigned to any nodes and then it will find suitable node and run it on the selected node.

Controller Manager

Controller Manager responsible for overall health of entire cluster. It ensures nodes up and running all the times and correct number of PODS running as mentioned in the SPEC file. A reliable system always needs good controllers to control it's running components. kube-controller-manager consists different controllers.

Node Controller: It's controlled the numbers of nodes in the cluster. If one or more nodes goes down it will create new node.

Replication Controller: It maintains the total number of running pods for the deployment. If any pod gets destroyed it will make a new one.

Endpoints Controller: Populates the Endpoints object (that is, joins Services & Pods).

Service Account & Token Controllers: It will manage different service account and access tokens required for other components.

etcd

etcd is light-weight key value distributed database. It's developed by CORE-OS and it stores the current cluster state at any point of time. The Kubernetes always need fast and highly available storage mechanism to store data for the clusters. It stores meta information for any kind of deployments inside the Kubernetes server.

Worker Node Components

Kubelet

The kubelet is primary node agent that runs in every worker node inside the cluster. The primary objective of the kubelet is it's looked the POD spec that was submitted by the API server on the Kubernetes master and ensures that containers describe PODS spec are running and healthy. In case if any issues in PODS running on worker node it's try to restart. If problem having worker node it will try to recreate the POD in other worker nodes.

Kube-Proxy

Kube-Proxy is the critical element inside Kubernetes cluster. It's responsible for maintaining the entire network configuration. It essentially maintains the distributed network across all the nodes, PODS and Containers and it also exposes the services outside world. It essentially core networking component inside the Kubernetes cluster.

Kubernetes installation on aws

Launch three ubuntu servers and run the below instructions

```
sudo apt update && sudo apt upgrade -y

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

sudo tee /etc/modules-load.d/containerd.conf <<EOF
overlay
br_netfilter
EOF
sudo modprobe overlay
sudo modprobe br_netfilter

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

sudo sysctl --system

sudo apt install -y curl gnupg2 software-properties-common apt-transport-https ca-certificates

sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmour -o /etc/apt/trusted.gpg.d/docker.gpg
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"

sudo apt update
sudo apt install -y containerd.io

containerd config default | sudo tee /etc/containerd/config.toml >/dev/null 2>&1
sudo sed -i 's/SystemdCgroup \= false/SystemdCgroup \= true/g' /etc/containerd/config.toml

sudo systemctl restart containerd
sudo systemctl enable containerd

echo "deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] https://pkgs.k8s.io/core:/stable:/v1.30/deb/ /" | sudo tee /etc/apt/sources.list.d/kubernetes.list
```

```
curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.30/deb/Release.key | sudo gpg --dearmor -o /etc/apt/keyrings/kubernetes-apt-keyring.gpg


sudo apt update
sudo apt install -y kubelet kubeadm kubectl
sudo apt-mark hold kubelet kubeadm kubectl

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

master --> sudo kubeadm init --pod-network-cidr=10.244.0.0/16 --ignore-preflight-errors=NumCPU,Mem

kubeadm join 10.1.1.87:6443 --token zy05t2.ev85g0ia1ozb04sj \
	--discovery-token-ca-cert-hash sha256:a923bd39832553bb157e1a20714f3cadcc287e81d14d34f536cdec33e5b084be


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

==============================
Worker
======
kubeadm join 10.1.1.87:6443 --token zy05t2.ev85g0ia1ozb04sj \
	--discovery-token-ca-cert-hash sha256:a923bd39832553bb157e1a20714f3cadcc287e81d14d34f536cdec33e5b084be


kubectl get nodes
```

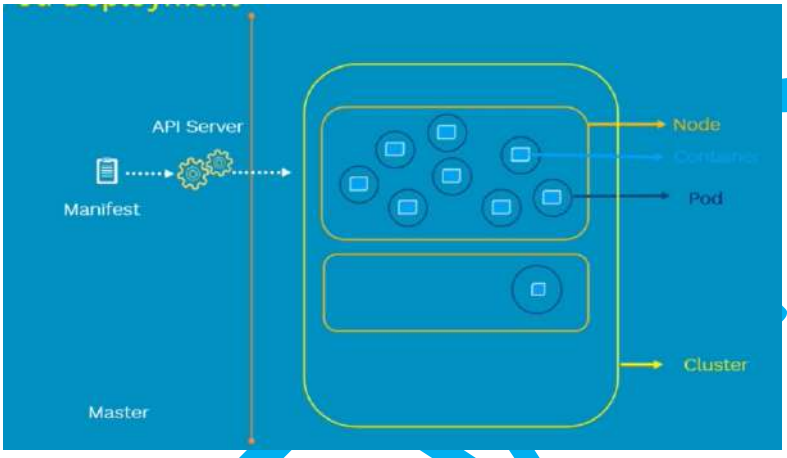
Prepared by PPNREDDY .... 😊 😊 😊

Pod

In Kubernetes, a **Pod** is the smallest and simplest unit in the Kubernetes object model. A Pod represents a single instance of a running process in your cluster. Containers in a pod share the same IP address. A POD is a group of one or more application containers.

POD Deployment

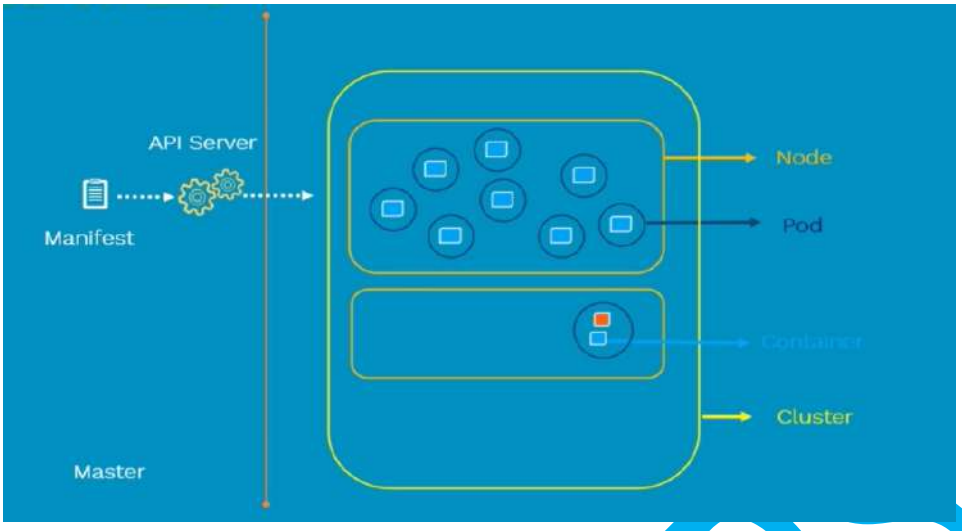
The pod has been going from a different phase since its birth. It can be terminated with Succeeded phase or it might be gone into the Failed state if container terminated with exit code other than 0.



- Kubectl or any other tool submit the meta information for the Pod to the API server.
- API server stores that information into the storage called etcd.
- Using watcher functionality, scheduler can check that there is a new pod available to be deployed and hasn't yet on to any node.
- Scheduler assign a node to the new pod and send acknowledge API server.
- API server updates these changes in to the etcd.
- Kublet in every node gets information from their watcher tool and deploy a pod's container on them if it gets assigned to them from the scheduler.
- Once containers get started on the node, kublet acknowledge API server about the container's state.
- This information is being stored into the etcd by the API server.
- The etcd acknowledge again to API server and this acknowledgment is being sent back to the kublet to tell it that changes are accepted.

Multi Container POD

Containers in a Pod run on a "logical host"; they use the same network namespace (in other words, the same IP address and port space), and the same IPC namespace. They can also use shared volumes. These properties make it possible for these containers to efficiently communicate, ensuring data locality. Also, Pods enable you to manage several tightly coupled application containers as a single unit.



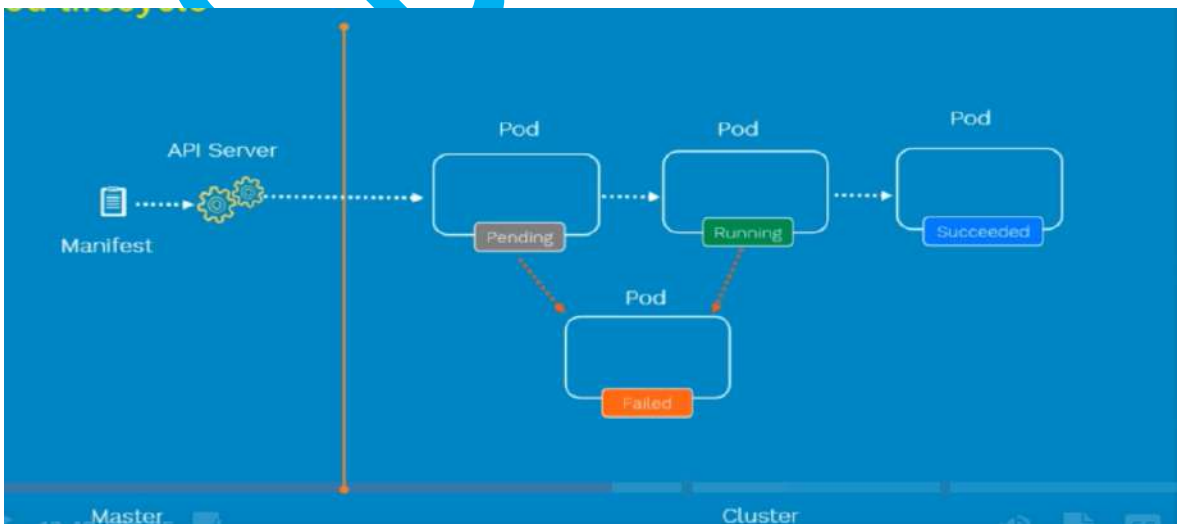
POD Structure

A Pod can contain one or more containers, typically Docker containers. All containers within a Pod share the following resources:

- **Network:** All containers in a Pod share the same IP address and network ports. They can communicate with each other using localhost and port numbers.
- **Storage:** Containers in a Pod can share storage volumes, making it easier to share files between containers.
- **Configuration:** Pods can be configured with environment variables, secrets, and ConfigMaps to provide the necessary runtime configuration to the containers.

POD Lifecycle

Pods are the smallest artifact in the Kubernetes System. There are always containers running inside the pod. These containers contain the package of the application and it's compatible environment. There are some cases in which pods face failure or other states during its life cycle.



A deployed pod goes through different phase or we can say states during its lifespan.

Pending:

When the Pod's metadata is accepted by the Kubernetes but still hasn't been deployed to any node, Pod goes into the Pending state. E.g. When Nodes do not have enough resources in the cluster, it causes the pod to go in pending state.

Running:

When the pod gets scheduled on any node and containers inside the pod started, Pod stays in Running phase.

Succeeded:

When containers inside the the pod get terminated with exit code 0 then the pod goes into the Succeeded State.

Failed:

When one of the containers inside the pod gets terminated with exit code other than 0, It causes the pod to go into the Failed state.

CrashLoopBackoff:

When the containers inside the pod get failed to start then the pod is being recreated again and again

POD Use Cases

Single Container Pod: In the simplest case, a Pod runs a single container. This is the most common scenario.

Multi-container Pod: Multiple containers can run in a Pod. In this case, the containers are tightly coupled and typically communicate via localhost. For example, one container might be a web server while the other might serve as a logging sidecar.

POD Communication

Inter-Pod Communication: Pods are assigned a unique IP address within the cluster's virtual network. This allows for communication between Pods.

Service Discovery: Pods can discover each other using Services. Services provide stable network endpoints for Pods, allowing them to communicate even if Pods are rescheduled or replaced.

POD Scheduling

- Kubernetes decides which node a Pod runs on based on various factors like available resources, taints, and tolerations. The **Kube-scheduler** component is responsible for this.
- Once scheduled on a node, the **Kubelet** ensures that the Pod's containers are running.

POD Health Monitoring

Kubernetes offers two primary mechanisms for checking the health of containers in a Pod:

Liveness Probes: To check if the container is still running. If the probe fails, Kubernetes will restart the container.

Readiness Probes: To determine if the container is ready to serve traffic. If a readiness probe fails, the Pod is temporarily removed from service discovery.

POD Lifecycle Management

Pods are usually not created directly by users but are created indirectly by higher-level controllers, such as:

Deployments: Ensure a specified number of replicas of the Pod are running at all times.

ReplicaSets: Manage the replication of Pods.

DaemonSets: Ensure that all (or some) nodes run a copy of a Pod.

Jobs: Ensure that a specified number of Pods complete successfully.

POD Termination and Deletion

When a Pod is deleted (either manually or by a controller), Kubernetes gracefully handles the termination of the containers inside the Pod:

- **Termination Grace Period:** Kubernetes sends a SIGTERM signal to the containers, giving them a grace period to finish their work before shutting down.
- **Graceful Shutdown:** During this time, the container can perform cleanup tasks. After the grace period, Kubernetes forcibly kills the containers.

POD Templates

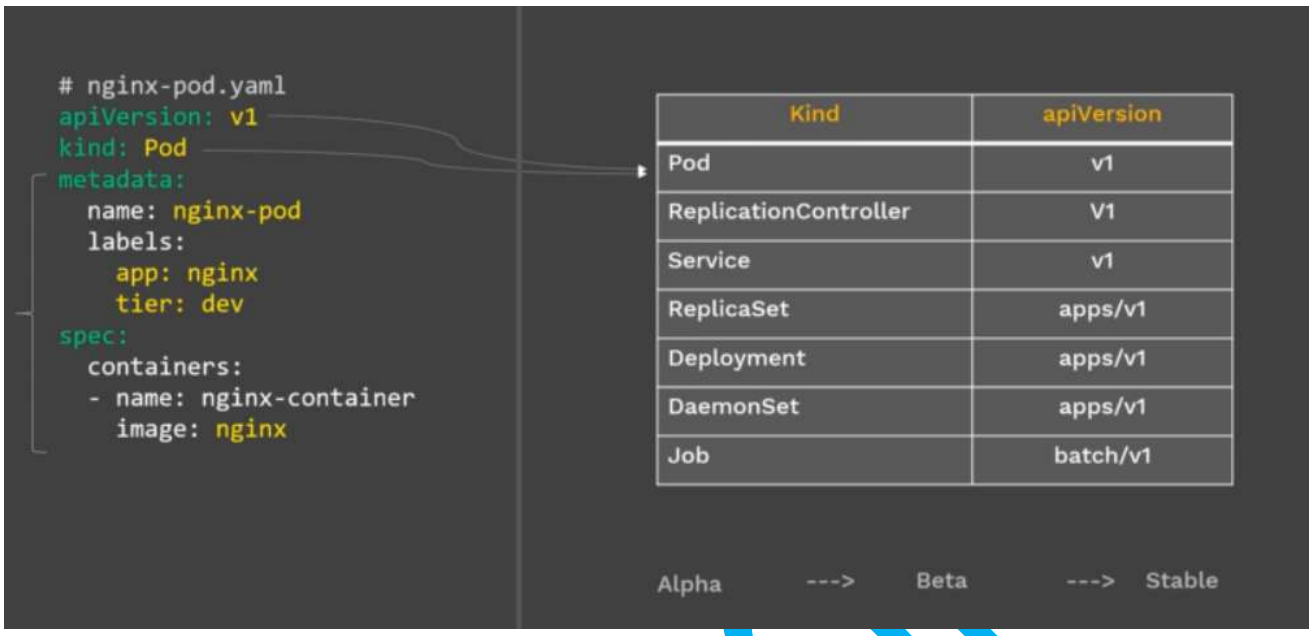
Pod templates are used in higher-level objects (like Deployments and ReplicaSets) to define the Pod's configuration. This template is essentially a blueprint for creating Pods.

POD Annotations and Labels

Labels: Key-value pairs that can be attached to Pods to organize and select them in queries. For example, you can label Pods as app=frontend.

Annotations: Annotations store non-identifying information such as metadata or details like a build version.

POD Configuration



To create a pod with yml we have to define few properties in yml file. those are

- apiVersion
- kind
- metadata
- spec

apiVersion:

apiVersion represents which version api you are using

kind:

kind represents what you are going to create with this yml. Here we can mention pod, replication controller, deployment, replication set…etc.

metadata:

in metadata we can add name to this pod and we can add labels to this pod

spec:

in spec we have to add the docker image information. using this docker image pod will be created

Kubectl commands

1. Create resource from a file
   - kubectl create -f pod-example.yaml
   - kubectl create -f pod-example.json
   - kubectl create -f deploy-example.yaml
   - kubectl create -f <directory>
   - kubectl apply -f pod-example.yaml

2. List out one OR more resources
   - kubectl get pods <pod-name>
   - kubectl get pods -o wide
   - kubectl get pods,deploy

3. Display detailed state of one or more resources
   - kubectl describe nodes <node-name>
   - kubectl describe pods <pod-name>
   - kubectl describe pods

4. Delete resources
   - kubectl delete -f pod-example.yaml
   - kubectl delete pods,services -l name=<label-name>
   - kubectl delete pods –all

5. Executing a command against a container in a pod
   - kubectl exec <pod-name> date
   - kubectl exec <pod-name> -c <container-name> date
   - kubectl exec -it <pod-name> /bin/bash

6. Print the logs for a container in a pod
   - kubectl logs <pod-name>
   - kubectl logs -f <pod-name> (Running logs like tail -f in linux)

Checking cluster status

Kubectl get nodes

```
ubuntu@ip-10-1-1-187:~$ kubectl get nodes
NAME            STATUS   ROLES           AGE   VERSION
ip-10-1-1-18    Ready    <none>          78m   v1.30.5
ip-10-1-1-187   Ready    control-plane   79m   v1.30.5
ip-10-1-1-188   Ready    <none>          78m   v1.30.5
ubuntu@ip-10-1-1-187:~$
```

Creating POD

```
---
 kind: Pod                  # Object Type
 apiVersion: v1             # API version
 metadata:                  # Set of data which describes the Object
  name: samplepod           # Name of the Object
 spec:                      # Data which describes the state of the Object
  containers:               # Data which describes the Container details
   - name: samplecont       # Name of the Container
     image: ubuntu:latest   # Base Image which is used to create Container
     command: ["/bin/bash", "-c", "while true; do echo Hello-Kubernetes; sleep 5 ; done"]
  restartPolicy: Never      # Defaults to Always
```

Commands

```
kubectl apply -f testpod.yaml
kubectl get pods
kubectl get pods -o wide
kubectl describe pod samplepod
kubectl exec samplepod -it /bin/bash
kubectl exec samplepod -it -c samplecont /bin/bash
kubectl exec samplepod -- hostname -i
kubectl exec samplepod date
kubectl exec samplepod -c samplecont date
kubectl exec samplepod -c samplecont ls
kubectl logs -f samplepod
kubectl logs -f samplepod -c samplecont
kubectl delete -f testpod.yaml
```

Passing environment variables to the POD

```
---
 kind: Pod
 apiVersion: v1
 metadata:
  name: environments
 spec:
  containers:
   - name: mycont
     image: ubuntu:latest
     command: ["/bin/bash", "-c", "while true; do echo Hello-kubernetes; sleep 5 ; done"]
     env:                  # List of environment variables to be used inside the pod
      - name: MYNAME
        value: PPNREDDY

Command:
kubectl exec environments -- /bin/bash -c  'echo $MYNAME'
```

Multi container pod

```
kind: Pod
apiVersion: v1
metadata:
 name: multicontpod
spec:
 containers:
  - name: container-1
    image: ubuntu:latest
    command: ["/bin/bash", "-c", "while true; do echo Hello-India; sleep 5 ; done"]
  - name: container-2
    image: ubuntu:latest
    command: ["/bin/bash", "-c", "while true; do echo Hello-Karnataka; sleep 5 ; done"]

Commands
========
kubectl get pods
kubectl exec multicontpod -it -c container-1 /bin/bash
kubectl exec multicontpod -it -c container-2 /bin/bash
kubectl logs -f multicontpod -c container-1
kubectl logs -f multicontpod -c container-2
```

Exposing port

```
---
 kind: Pod
 apiVersion: v1
 metadata:
  name: http-pod
 spec:
  containers:
   - name: mycont
     image: httpd:latest
     ports:
      - containerPort: 80

 Commands
 kubectl exec http-pod -- hostname -i
 curl 10.244.54.7:80
 <html><body><h1>It works!</h1></body></html>
```

Pods with labels

```
---
 kind: Pod
 apiVersion: v1
 metadata:
  name: labelspod
  labels:                # Specifies the Label details under it
   env: development
   class: pods
 spec:
  containers:
   - name: labelcont
     image: ubuntu:latest
     command: ["/bin/bash", "-c", "while true; do echo Hello-kubernetes; sleep 5 ; done"]

Commands
========

kubectl get pods --show-labels
kubectl label pods <podname> <labelkey>=<value>
kubectl get pods -l env=development
```

Pod with annotations

```
---
 kind: Pod
 apiVersion: v1
 metadata:
  name: annotations-pod
  labels:
   app: apache
  annotations:
   description: "Deployment for Apache HTTP Server"
   maintainer: "ppreddy@hcltech.com"
 spec:
  containers:
   - name: mycont
     image: httpd:latest
     ports:
      - containerPort: 80

Commands
========
kubectl get pods --show-labels
kubectl get pod annotations-pod -o jsonpath='{.metadata.annotations}'
kubectl get pod annotations-pod  -o yaml | grep annotations -A 5
```

NodeSelector

In Kubernetes, a **NodeSelector** is used to control on which nodes a pod can be scheduled. It allows you to select specific nodes based on their labels. This feature helps ensure that pods are deployed only on nodes that meet specific criteria (e.g., hardware type, environment, or region).

Labelling a Node

First, you need to label a node to define its characteristics.

kubectl label nodes <node-name> <label-key>=<label-value>

kubectl label nodes node-1 disktype=ssd

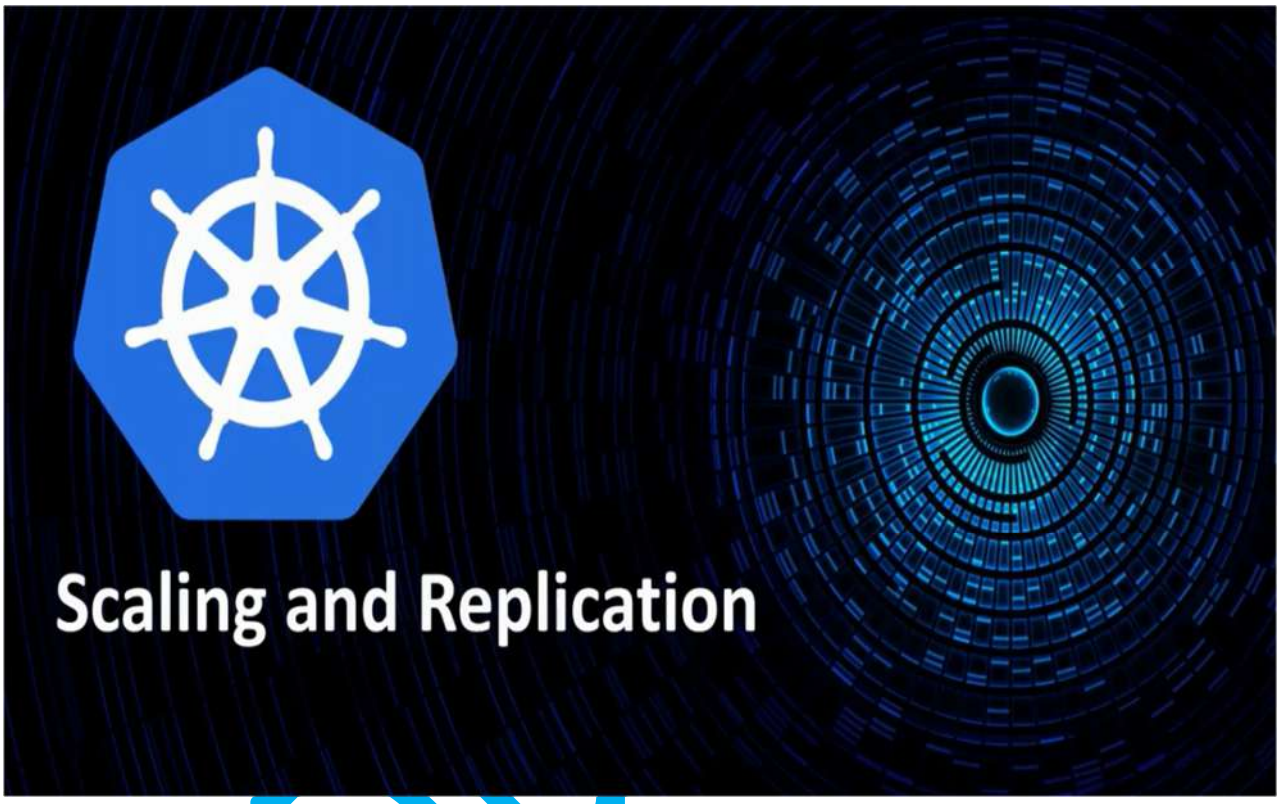
```
kubectl label nodes ip-10-1-1-18 disktype=ssd
---
 apiVersion: v1
 kind: Pod
 metadata:
  name: my-pod
 spec:
  containers:
   - name: container-1
     image: ubuntu:latest
     command: ["/bin/bash", "-c", "while true; do echo Hello-India; sleep 5 ; done"]
  nodeSelector:
   disktype: ssd
```

Commands:

```
kubectl apply -f testpod.yaml
kubectl get pods -o wide
kubectl describe node ip-10-1-1-18
```

Note:

- **NodeSelector** is a simple way to constrain pods to nodes with specific labels.
- It's a strict requirement: if no node matches the nodeSelector, the pod will remain unscheduled.
- It provides basic node selection functionality compared to more advanced mechanisms like **Node Affinity**.



Prepared by PPNREDDY .... 😊 😊 😊

Scaling & Replication

- Kubernetes was designed to orchestrate multiple containers and replication.
- Need for multiple containers/replications helps us with these:

Reliability: By having multiple versions of an application, you prevent problems if one or more fails.

Load balancing: Having multiple versions of a container enables you to easily send traffic to different instances to prevent overloading of a single instance or node.

Scaling: When load does become too much for the number of existing instances, Kubernetes enables you to easily scale up your application, adding additional instances as needed

Replication Controller

- Replication Controller is one of the key features of Kubernetes, which is responsible for managing the pod lifecycle. It is responsible for making sure that the specified number of pod replicas are running at any point of time.
- A Replication Controller is a structure that enables you to easily create multiple pods, then make sure that that number of pods always exists. If a pod does crash, the Replication Controller replaces it.
- If there are excess Pods, they get killed and vice versa.
- New Pods are launched when they get fail, get deleted or terminated.
- Replication Controllers and Pods are associated with " labels ".
- Creating a "rc" with count of 1 ensure that a pod is always available.

Advantages

Pod Availability and Reliability

The primary function of a Replication Controller is to ensure that a certain number of identical Pods are always running in your cluster. If any Pods fail or are terminated, the Replication Controller automatically replaces them. This guarantees availability and resiliency of the application, improving reliability.

Automatic Scaling

You can manually or programmatically adjust the number of replicas, and the Replication Controller will automatically scale up or down the number of Pods to match the specified count.

Load Distribution

With multiple replicas, Kubernetes distributes the traffic across all Pods. This ensures that traffic is not concentrated on a single Pod, improving both performance and reliability.

Self-healing

The Replication Controller constantly monitors the health of the Pods it manages. If a Pod crashes or becomes unhealthy, it will automatically replace it with a new, healthy one. This self-healing mechanism ensures higher uptime.

Declarative Configuration

Like other Kubernetes resources, the desired state of the Pods managed by the Replication Controller is declared in a configuration file (YAML/JSON). The system then works to maintain that state, providing consistency and predictability across environments.

Automated Rollout and Rollback (using higher-level constructs like Deployments)

While the Replication Controller itself doesn't handle versioning or rolling updates directly, it can be used in conjunction with higher-level objects like Deployments. This ensures smooth rollouts and rollbacks of new versions of an application, without downtime.

Use Cases of Replication Controller:

Simple Stateless Applications: For basic applications that don't require complex rollouts or state management, a Replication Controller provides a straightforward way to maintain application availability.

Batch Processing: If you need to run a specific number of jobs or batch processes in parallel, a Replication Controller can ensure that the appropriate number of Pods are always running to handle the tasks.

Replication Controller Demo

```
## ReplicationController Demo
---
 kind: ReplicationController          # This defines to create the object of replication type
 apiVersion: v1
 metadata:
  name: myrc
 spec:
  replicas: 2          # This element defines the desired number of pods
  selector:        # Tells the controller which pods to watch/belong to this Replication Controller
   myname: ppreddy     # These must match the labels
  template:        # Template element defines a template to launch a new pod
   metadata:
    name: rcpod
    labels:
     myname: ppreddy
   spec:
    containers:
     - name: rccontainer
       image: ubuntu:latest
       command: ["/bin/bash", "-c", "while true; do echo Hello-ppreddy; sleep 5 ; done"]
```

Commands

```
kubectl apply -f rc-demo.yaml
kubectl get rc
kubectl get pods
kubectl describe rc myrc
kubectl get pods -o wide
kubectl delete pod myrc-pt9d6
kubectl get pods -o wide
kubectl get pods -l myname=ppreddy
kubectl scale rc myrc --replicas=10
kubectl scale rc myrc --replicas=5
kubectl delete -f rc-demo.yaml
```

ReplicaSet

- ReplicaSet is the next-generation Replication Controller.
- The only difference between a ReplicaSet and a Replication Controller right now is the selector support.
- ReplicaSet supports the new set-based selector requirements as described in the label's user guide where as a Replication Controller only supports equality-based selector requirements.
- ReplicaSet and Pods are associated with " labels ".
- The key improvement in ReplicaSet over Replication Controller is its support for label selectors that allow for more sophisticated selection criteria for managing pods.

Key Features of a ReplicaSet

Ensures Desired Number of Pods

- A ReplicaSet guarantees that a specified number of pod replicas are always running. If any pod fails or is deleted, a new one is automatically created to maintain the desired state.

Pod Matching via Label Selectors

- ReplicaSet use **label selectors** to identify and manage pods. This allows them to be more flexible than Replication Controllers because they can manage pods based on more complex criteria.
- **Example:** Instead of just managing pods by name, a ReplicaSet can select pods that match a combination of labels (e.g., app=nginx, version=v1).

Scaling

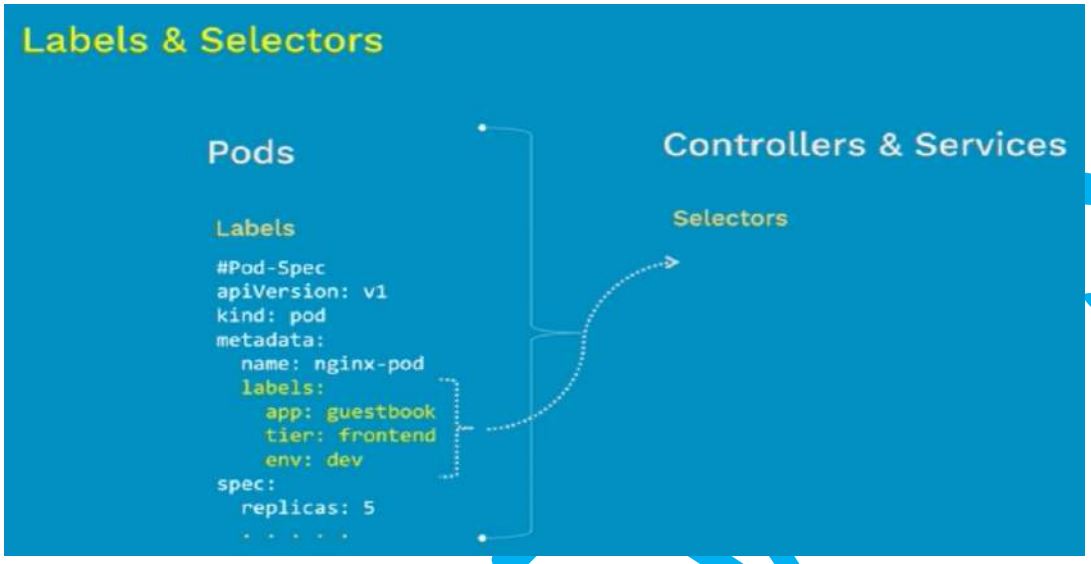
- You can easily scale a ReplicaSet up or down by changing the number of replicas either manually or automatically via Horizontal Pod Autoscaler (HPA).
- Example command for scaling:

```
kubectl scale replicaset my-app --replicas=5
```

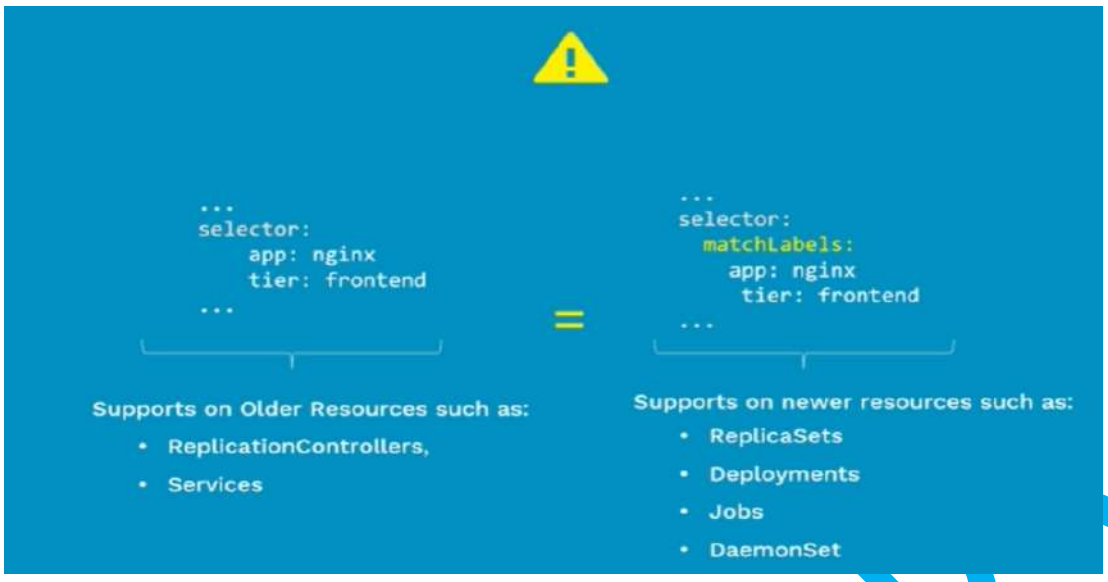
Self-Healing

- Similar to the Replication Controller, a ReplicaSet is constantly monitoring the pods it manages. If any pod crashes or is terminated, the ReplicaSet automatically replaces it, ensuring application uptime.

Labels && Selectors



Equality-based Vs Set-based



Comparison with Replication Controller

Feature	ReplicaSet	Replication Controller
Label selector support	**Advanced** (supports set-based and equality-based label selectors)	**Basic** (only equality-based selectors)
Recommended usage	Yes (often used via Deployments)	No (deprecated in favor of ReplicaSet)
Flexibility in managing pods	Higher flexibility	Lower flexibility

Note:

- Replication Controller is old generation Replication Controller it's supports equality-based selectors.
- ReplicaSet is new and it is next generation Replication Controller it's supports set based selectors.

ReplicaSet -Demo

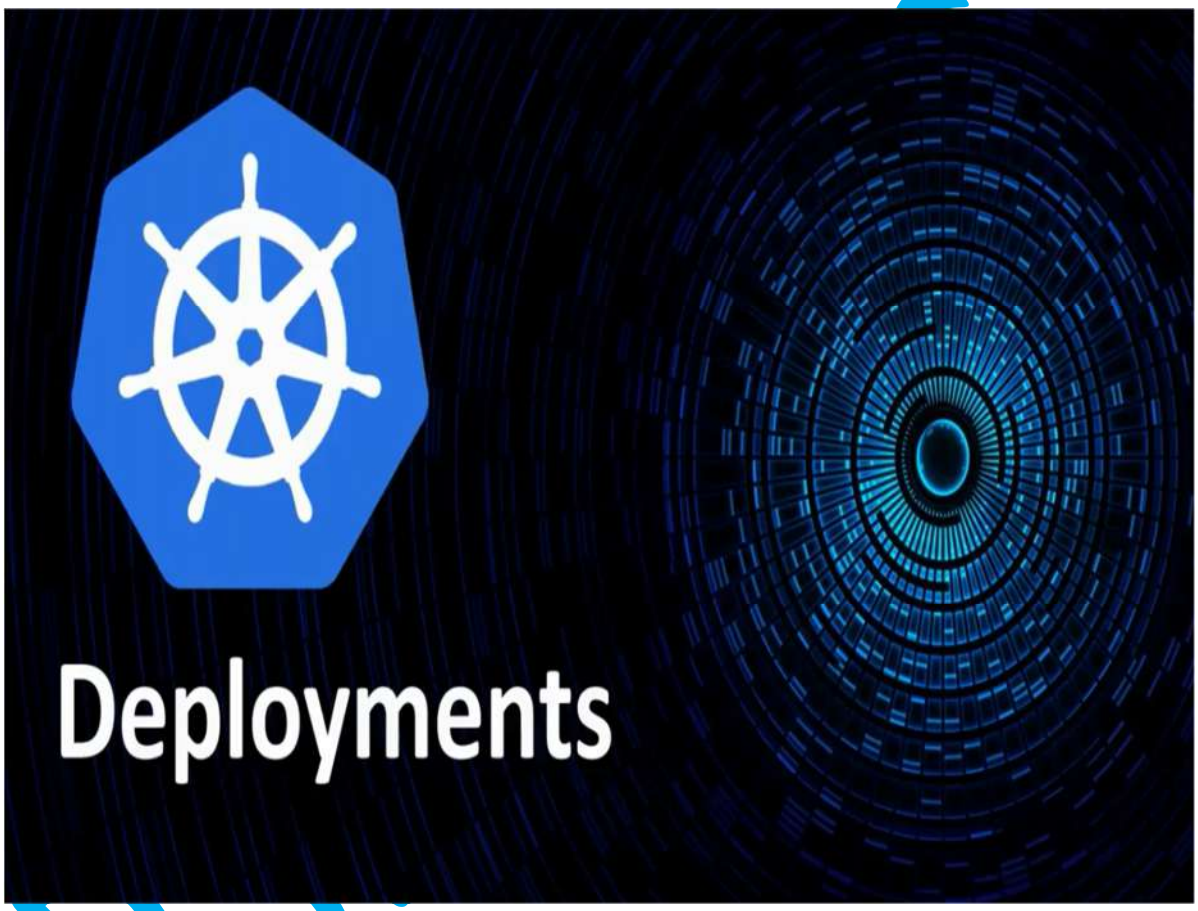
```
## ReplicaSet Demo

---
  kind: ReplicaSet                # Defines the object to be ReplicaSet
  apiVersion: apps/v1             # Replicaset is not available on v1
  metadata:
    name: myrs
  spec:
```

```
  replicas: 2
  selector:
    matchExpressions:                    # These must match the labels
      - {key: myname, operator: In, values: [ppreddy, ppnreddy, preddy]}
      - {key: env, operator: NotIn, values: [production]}
  template:
    metadata:
      name: rspod
      labels:
        myname: ppreddy
        env: dev
    spec:
      containers:
        - name: rscontainer
          image: ubuntu:latest
          command: ["/bin/bash", "-c", "while true; do echo Hello-ppreddy; sleep 5 ; done"]
```

Commands

========

```
kubectl apply -f rs-demo.yaml
kubectl get rs
kubectl describe rs myrs
kubectl get pods
kubectl delete pod myrs-v7ggf
kubectl get pods
kubectl get pods -l myname=ppreddy
kubectl get pods -l env=dev
kubectl scale rs myrs --replicas=10
kubectl scale rs myrs --replicas=5
kubectl delete -f rs-demo.yaml
```

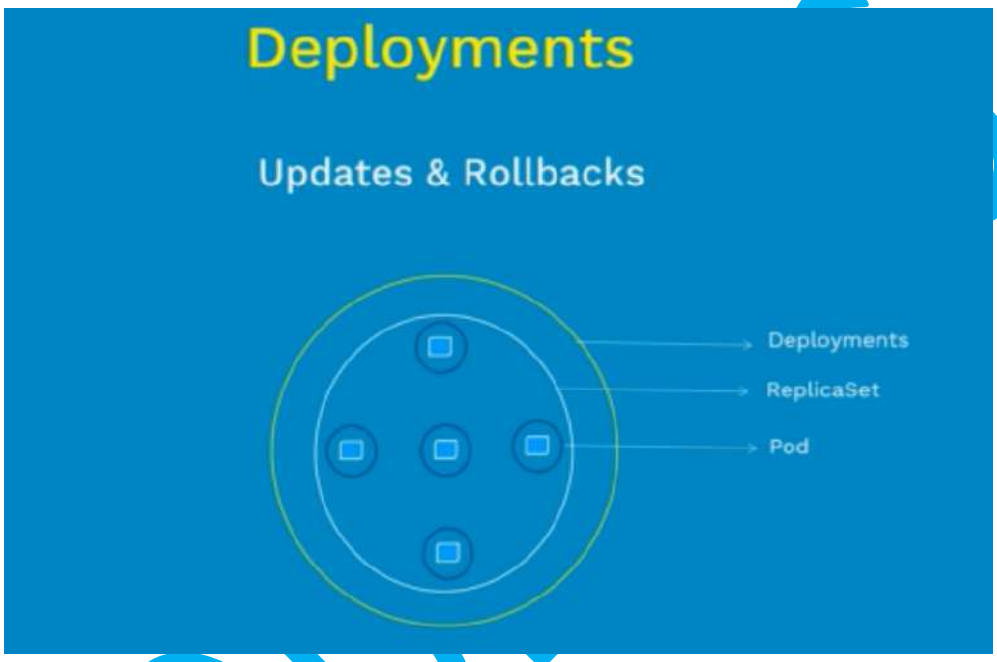


Prepared by PPNREDDY .... 😊 😊 😊

Deployments

A Kubernetes Deployment is a resource that provides declarative updates to applications. Deployments ensure that a specified number of pod replicas are running and available at all times. If a pod fails or is deleted, the Deployment controller will replace it to maintain the desired state.

Deployments are controller similar to ReplicationController and ReplicaSet. Deployment implements the features like rolling updates, roll-back of updates. Deployments have the capability to update the docker image which you are using in the pods and are also capable of rolling back to the previous docker image versions.



Advantages

Kubernetes deployments offer several advantages for managing containerized applications, providing scalability, efficiency, and ease of management.

1. Automated Rollouts and Rollbacks

Kubernetes deployments allow for automated rollouts of new versions of applications. If something goes wrong, you can easily roll back to a previous version, minimizing downtime.

2. Self-healing

Kubernetes automatically restarts failed containers, replaces containers, kills containers that don't respond to your user-defined health checks, and doesn't advertise them to clients until they are ready to serve.

3. Scaling

Deployments make it easy to scale your applications up and down by changing the number of replicas of your service. Kubernetes handles distributing the load across the available resources automatically.

4. Declarative Updates

Kubernetes allows you to define the desired state of your application, and it will work to ensure that the current state matches the desired state. This simplifies updates and configuration management.

5. Load Balancing

Kubernetes deployment automatically balances the load across your pods, ensuring that traffic is evenly distributed for optimal performance and reliability.

6. Resource Optimization

Kubernetes can automatically allocate resources such as CPU and memory to your containers based on their demands, optimizing resource usage and reducing waste.

7. Portable Across Cloud Providers

Kubernetes can run on any cloud platform (AWS, Azure, Google Cloud) or on-premise, giving you flexibility and avoiding vendor lock-in.

8. Efficient Resource Utilization

With features like horizontal pod autoscaling, Kubernetes allows applications to automatically scale based on demand, ensuring efficient use of compute resources.

9. Service Discovery and Load Balancing

Kubernetes can automatically discover and expose services via DNS names or IP addresses, reducing manual effort in service discovery and integration.

10. Canary Deployments and Blue-Green Deployments

Kubernetes supports advanced deployment techniques like canary and blue-green deployments, which allow you to test changes with minimal risk and downtime.

Deployment Demo

```
---
 kind: Deployment
 apiVersion: apps/v1
 metadata:
  name: mydeployments
 spec:
  replicas: 2
  selector:     # tells the controller which pods to watch/belong to
    matchLabels:
      name: deployment
  template:
    metadata:
      name: deppod
      labels:
        name: deployment
    spec:
```

```
      containers:
        - name: dep-cont
          image: ubuntu:latest
          command: ["/bin/bash", "-c", "while true; do echo Hello-DevOps; sleep 5; done"]
```

Commands

```
========
kubectl apply -f deploy-demo.yaml
kubectl get deployment
kubectl describe deployment mydeployments
kubectl get rs
kubectl get pods
kubectl get pods -o wide

kubectl delete pod  mydeployments-84877d89c9-vw9gd
kubectl get pods

kubectl delete rs mydeployments-84877d89c9
kubectl get rs
kubectl get pods

kubectl scale deployment mydeployments --replicas=10
kubectl get deployment
kubectl describe deployment mydeployments
kubectl get rs
kubectl get pods

kubectl scale deployment mydeployments --replicas=1
kubectl get deployment
kubectl describe deployment mydeployments
kubectl get rs
kubectl get pods

kubectl logs -f mydeployments-84877d89c9-7pk2m
```

Let's deploy another version

```
-------------------------
---
  kind: Deployment
  apiVersion: apps/v1
```

```
  metadata:
    name: mydeployments
  spec:
    replicas: 2
    selector:      # tells the controller which pods to watch/belong to
      matchLabels:
        name: deployment
    template:
      metadata:
        name: deppod
        labels:
          name: deployment
      spec:
        containers:
          - name: dep-cont
            image: centos:latest
            command: ["/bin/bash", "-c", "while true; do echo Hello-DevOps-centos; sleep 5; done"]
```

Commands

========

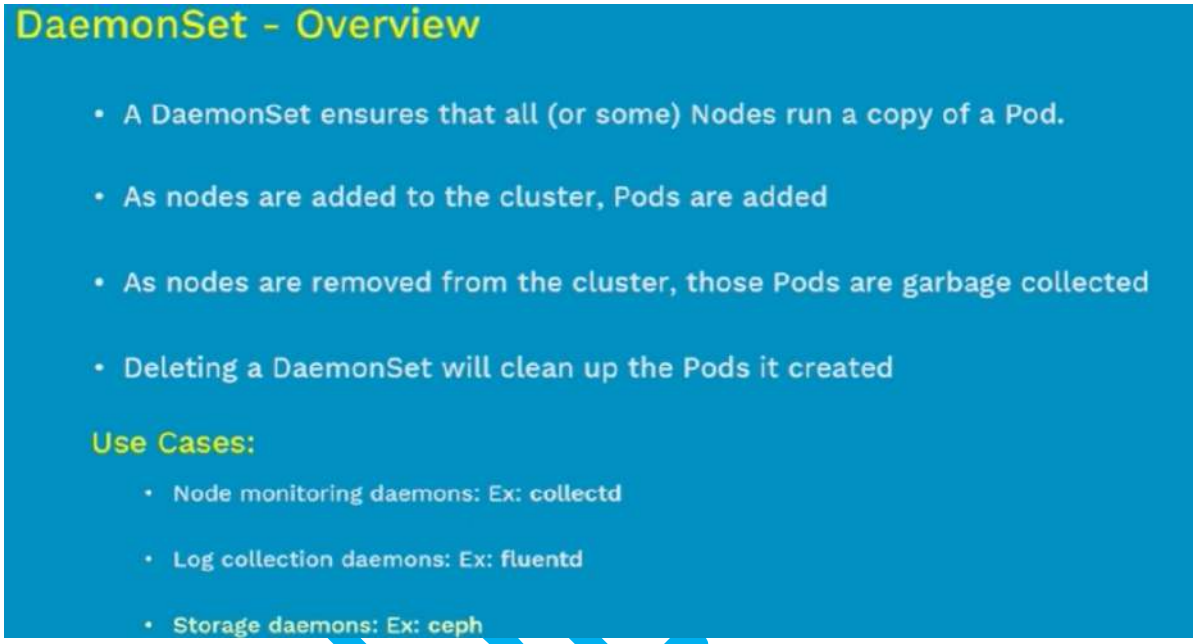
```
kubectl apply -f deploy-demo.yaml
kubectl get deployment
kubectl describe deployment mydeployments
kubectl get rs
kubectl get pods
kubectl get pods -o wide
kubectl logs -f mydeployments-9cf4bc565-tn9z2
kubectl rollout status deployment mydeployments
kubectl rollout history deployment mydeployments
kubectl rollout undo deploy/mydeployments --to-revision=1
kubectl get rs
kubectl get pods
kubectl get pods -o wide
kubectl logs -f mydeployments-84877d89c9-dwm26
kubectl get deployment
kubectl describe deployment mydeployments
kubectl delete -f deploy-demo.yaml
```

Prepared by PPNREDDY .... 😊 😊 😊

DaemonSet

A **DaemonSet** in Kubernetes ensures that a copy of a specific pod runs on all (or a subset of) nodes in the cluster. It is typically used for running system-level or background tasks on every node, such as logging agents, monitoring services, or network configuration utilities.

When a new node is added to the cluster, a Pod is added to it to match the rest of the nodes and when a node is removed from the cluster, the Pod is garbage collected. Deleting a DaemonSet will clean up the Pods it created.



Key Features of DaemonSets:

1. One Pod Per Node

DaemonSets ensure that exactly one pod runs on each node in the cluster. As new nodes are added to the cluster, a pod will automatically be scheduled on the new nodes.

2. Background Services

Common use cases include running:

- Monitoring Agents: Tools like Prometheus or Datadog that collect metrics from nodes.
- Logging Daemons: Fluentd, Filebeat, or Logstash to collect and ship logs from nodes.
- Network Proxies/Overlay Management: Tools that manage network configuration, proxies, or other networking needs at the node level.

3. Rolling Updates

DaemonSets support rolling updates, allowing you to incrementally update pods on each node without downtime. This is useful for gradually deploying updates to background services.

4. Scheduled on Specific Nodes (Node Selectors)

You can limit DaemonSets to run only on certain nodes using labels, node selectors, or affinity rules. For example, you might want your DaemonSet to run only on Linux-based nodes or nodes with specific hardware.

5. Guaranteed Availability

Because a DaemonSet runs one pod per node, the service it provides is guaranteed to be available on every node in the cluster. This is crucial for services that must operate on all nodes, like security tools or storage daemons.

6. Automatic Scaling

When new nodes are added to the cluster, DaemonSet automatically schedules a pod on those nodes without manual intervention. Similarly, when nodes are removed, the DaemonSet pod on that node is automatically cleaned up.

7. No Load Balancing

Unlike Deployments, DaemonSets don't provide load balancing. Each pod runs independently on its node, handling its own specific tasks or responsibilities.

Typical Use Cases:

Log Collection: Tools like Fluentd or Filebeat to collect logs from all nodes and forward them to a central storage or analysis service.

Node Monitoring: Metrics collection agents like Prometheus Node Exporter or Datadog Agent to monitor node health and performance.

Security Agents: Running security daemons or intrusion detection tools like Falco on every node for system-level monitoring.

Storage Daemons: Pods for storage systems like Ceph or GlusterFS that need to run on every node for managing distributed storage.

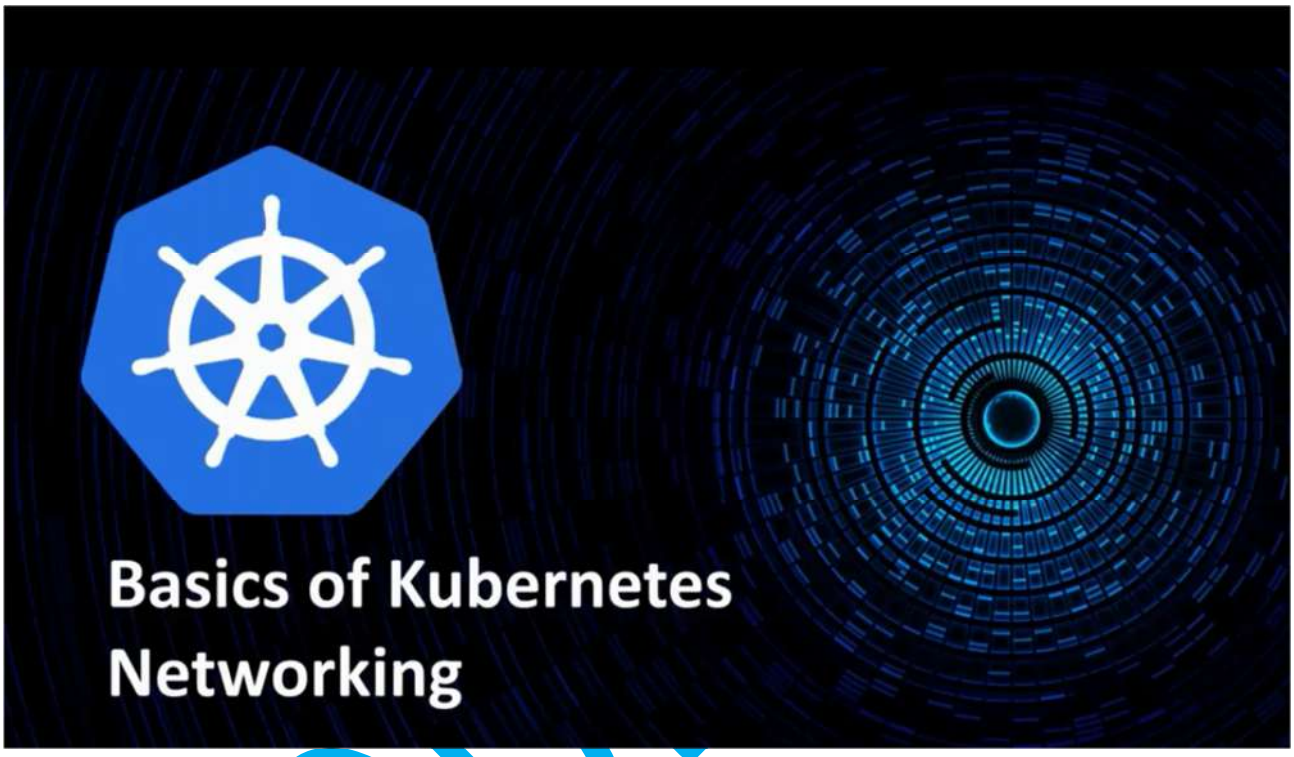
DaemonSets Demo

```
---
 apiVersion: apps/v1
 kind: DaemonSet
 metadata:
  name: nginx-ds
 spec:
  selector:
   matchLabels:
    app: nginx-app
```

```
  template:
    metadata:
      name: nginx-pod
      labels:
        app: nginx-app
    spec:
      containers:
        - name: nginx-container
          image: nginx:latest
          ports:
            - containerPort: 80
```

Commands
========

```
kubectl apply -f ds-demo.yaml

kubectl get ds
kubectl get rs
kubectl get pods
kubectl get pods -o wide
kubectl describe ds nginx-ds

kubectl delete -f  ds-demo.yaml
```

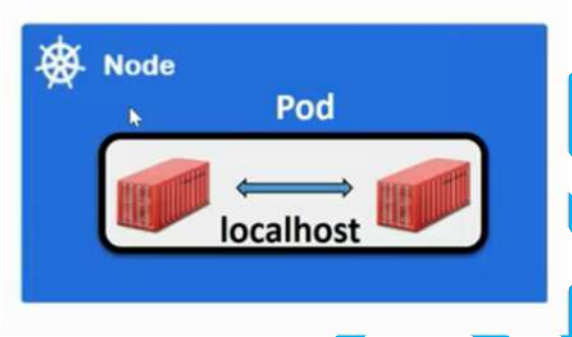


Prepared by PPNREDDY .... 😊 😊 😊

Kubernetes Networking

Kubernetes networking is essential for enabling communication between the various components (pods, services, nodes) in a Kubernetes cluster.

- Container to Container communication on same pod happens through localhost with in the container.



ctoc.yaml
=========

```
---
 kind: Pod
 apiVersion: v1
 metadata:
  name: testpod
 spec:
  containers:
   - name: uc
     image: ubuntu:latest
     command: ["/bin/bash", "-c", "while true; do echo Hello-Kubernetes; sleep 5 ; done"]
   - name: hc
     image: httpd:latest
     ports:
      - containerPort: 80
```

commands:
=========

```
kubectl apply -f ctoc.yaml
kubectl exec testpod -it -c uc -- /bin/bash
root@testpod:/# apt update && apt install curl
root@testpod:/# curl localhost:80
kubectl delete -f ctoc.yaml
```

- Pod to Pod communication on same worker happens through POD IP.
- By Default's Pod's IP will not accessible outside worker node.

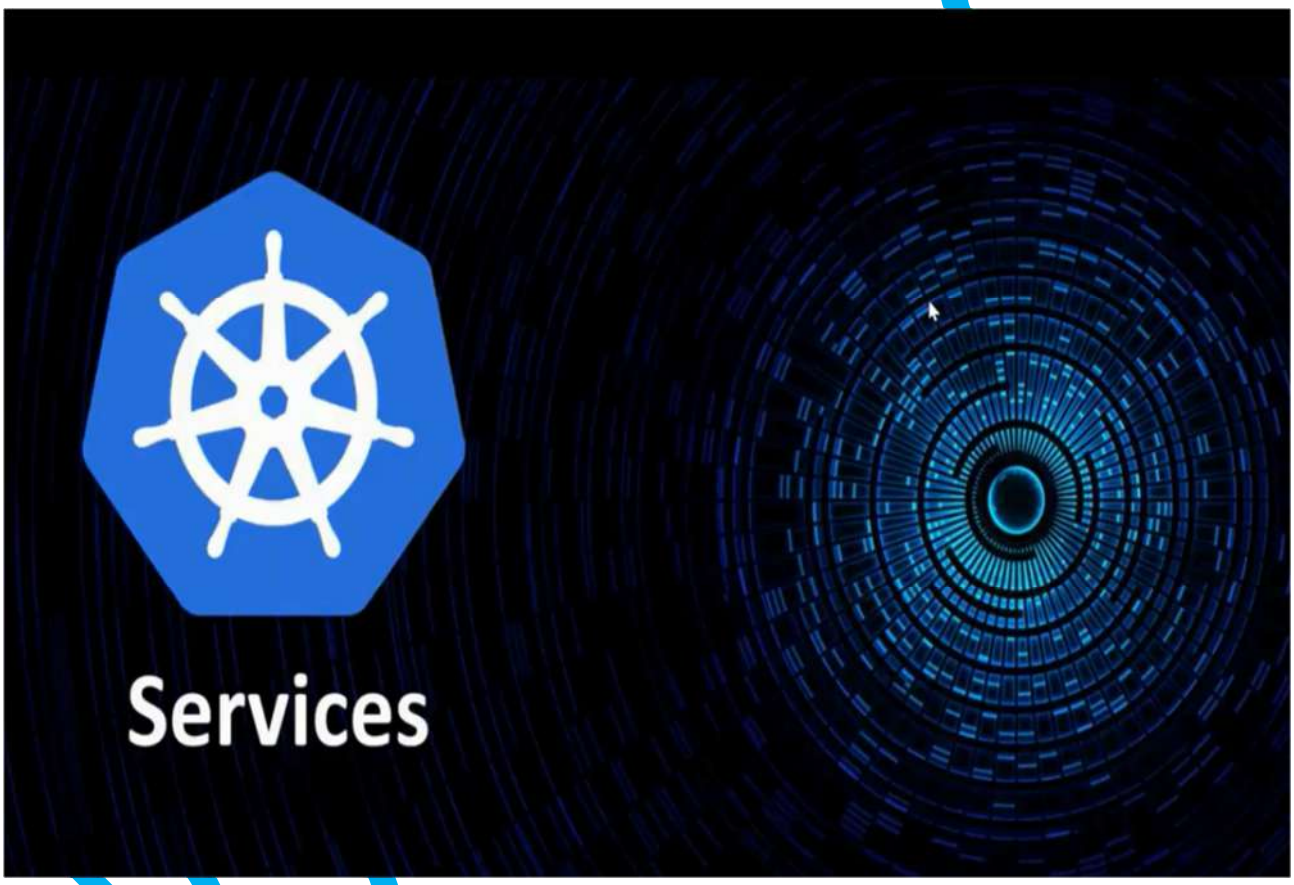
podnginx.yaml
==========

```
---
 kind: Pod
 apiVersion: v1
 metadata:
   name: testpodnginx
 spec:
   containers:
    - name: nc
      image: nginx:latest
      ports:
       - containerPort: 80
```

podnginx.yaml
==========

```
---
 kind: Pod
 apiVersion: v1
 metadata:
   name: testpodhttpd
 spec:
   containers:
    - name: hc
      image: httpd:latest
      ports:
       - containerPort: 80
```

commands:
=========
kubectl apply -f podnginx.yaml
kubectl apply -f podhttpd.yaml
kubectl get pods
kubectl get pods -o wide
curl 10.244.54.42:80
curl 10.244.54.41:80
kubectl delete -f podnginx.yaml
kubectl delete -f podhttpd.yaml



Prepared by PPNREDDY .... 😊 😊 😊

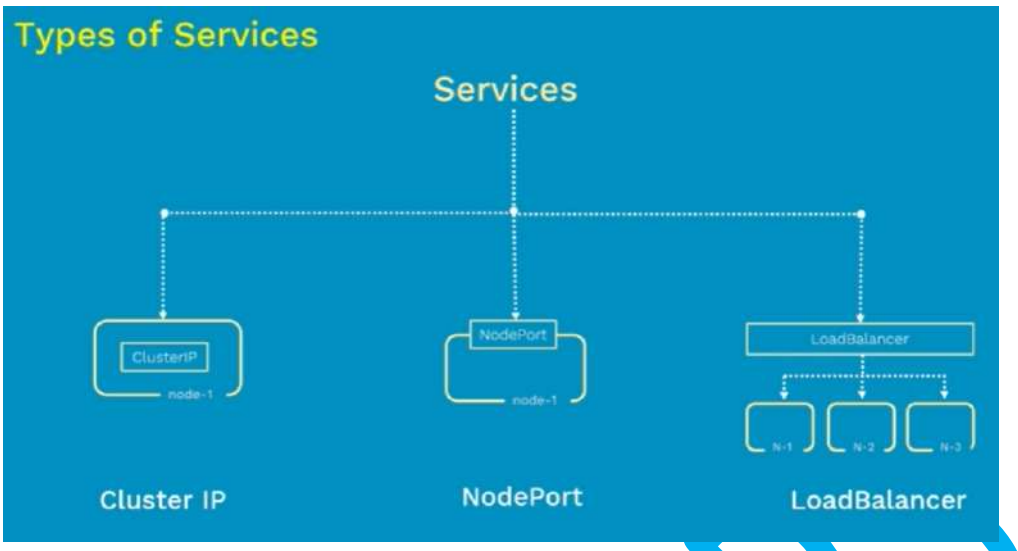
Services

In Kubernetes, **Services** provide a way to expose applications running on a set of Pods as a network service. Services allow Pods to communicate with each other or with external services without needing to know the underlying IP addresses of individual Pods. They ensure stable networking, as Pods themselves can be ephemeral and may change IP addresses as they are created or terminated. A service is responsible for making our Pods discoverable inside the network or exposing them to the internet. A Service identifies Pods by its LabelSelector.

- When using RC, pods are terminated and created during scaling or replication operations.
- When using Deployments, while updating the image version the pods are terminated & new pods take the place of older pods.
- Pods are very dynamic i.e. they come & go on the k8s cluster and on any of the available nodes & it would be difficult to access the pods as the pods IP changes once its recreated.
- Service Object is a logical bridge between pods & end-users, which provides Virtual IP (VIP) address.
- Service allows clients to reliably connect to the containers running in the pod using the VIP.
- The VIP is not an actual IP connected to a network interface, but its purpose is purely to forward traffic to one or more pods.
- kube-proxy is the one which Keeps the mapping between the VIP and the pods up-to-date, which queries the API server to learn about new services in the cluster.
- Although each Pod has a unique IP address, those IPs are not exposed outside the cluster.
- Services helps to expose the VIP mapped to the pods & allows applications to receive traffic.
- Labels are used to select which are the Pods to be put under a Service.
- Creating a Service will create an endpoint to access the pods/Application in it.
- By default service can run only between ports 30000-32767.

Types of Services:

- **ClusterIP** (default): Exposes the service on an internal IP within the cluster. This service is only accessible from within the Kubernetes cluster.
- **NodePort**: Exposes the service on each Node's IP at a static port. The service can be accessed externally via <NodeIP>:<NodePort>.
- **LoadBalancer**: Used in cloud environments. It provisions a load balancer for the service, which can be accessed externally.
- **ExternalName**: Maps a service to a DNS name, useful when directing traffic to services outside of the cluster.



Service Discovery:

Kubernetes offers service discovery mechanisms that allow Pods to find services by name. For example, if a service called my-service exists in a namespace, Pods can access it using the DNS name *my-service.<namespace>.svc.cluster.local*.

Service Selectors:

Services use **selectors** to determine which Pods they should route traffic to. For example, if a service has the selector app=my-app, it will forward traffic to all Pods that have the label app=my-app.

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

Endpoints:

A Service automatically creates **endpoints** to keep track of the Pods that match its selector. These are the IP addresses and ports of the Pods behind the service.

Headless Services:

A **headless service** is created when you do not need load balancing or a stable IP, but you still want service discovery. To create one, you set the clusterIP field to None.

```
apiVersion: v1
kind: Service
metadata:
  name: headless-service
spec:
  clusterIP: None
  selector:
    app: my-app
```

Use Cases:

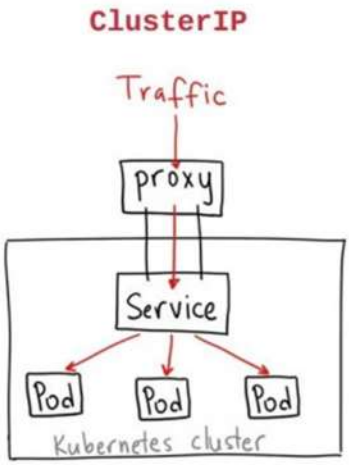
Internal Communication: Services facilitate communication between Pods without exposing them to the outside world.

External Access: Services like NodePort or LoadBalancer expose your applications to external users.

Load Balancing: By default, services route traffic evenly to all matching Pods.

ClusterIP

In Kubernetes, a **ClusterIP** is the default type of service that provides an internal IP address accessible only within the cluster. It is used for internal communication between different components of an application or between different applications running in the same Kubernetes cluster. The **ClusterIP** service ensures that Pods can discover and communicate with each other through a stable, internal virtual IP address, abstracting the dynamic nature of Pod IPs.



Features of ClusterIP:

Internal Access Only

ClusterIP services are only accessible within the cluster. This means external users or services outside the Kubernetes cluster cannot access this service directly.

Load Balancing

When multiple Pods match the service's selector, the service load-balances traffic among these Pods using round-robin or session affinity.

DNS-based Service Discovery

Kubernetes automatically assigns a DNS name to the service. For example, a service named my-service in the default namespace will be accessible at *my-service.default.svc.cluster.local*

Static Virtual IP

Even though Pods may come and go and their IPs may change, the ClusterIP remains stable, so services depending on it always have a consistent address to communicate with.

Example of a ClusterIP Service

Here's an example YAML configuration for creating a ClusterIP service that forwards traffic to Pods running on port 8080.

```
apiVersion: v1
kind: Service
metadata:
  name: my-clusterip-service
spec:
  type: ClusterIP
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80        # Port exposed by the service
      targetPort: 8080  # Port that the Pods listen on
```

In this example:

- port: 80: The service listens on port 80 for incoming traffic within the cluster.
- targetPort: 8080: The traffic received by the service on port 80 is forwarded to port 8080 on the selected Pods.
- selector: The service routes traffic to any Pods with the *label app=my-app*.

How it Works:

- The service gets a **ClusterIP** (a virtual IP) automatically assigned by Kubernetes.
- Any Pods or other services within the cluster can access the service using this IP address or its DNS name.
- If you have multiple replicas of a Pod (e.g., from a Deployment), the service will distribute traffic to those Pods automatically.

Use Cases

ClusterIP services are primarily used for:

- **Internal Microservice Communication**: Allows one microservice (Pod) to communicate with another inside the cluster.
- **Service Discovery**: Provides stable service discovery by name instead of needing to track individual Pod IPs.
- **Load Balancing**: Kubernetes automatically balances requests across all Pods that match the service's selector.
- **Stable IPs**: Even when Pods restart or scale up/down, the service's IP remains consistent.

Limitations

Since **ClusterIP** is restricted to the cluster, it cannot be used to expose services to external users or clients outside of the cluster. For external exposure, you would need to use other service types, like NodePort or LoadBalancer.

Demo

clusteripdeployment.yaml

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

---
 kind: Deployment
 apiVersion: apps/v1
 metadata:
  name: mydeployments
 spec:
  replicas: 1
  selector:     # tells the controller which pods to watch/belong to
    matchLabels:
      name: deployment
  template:
    metadata:
      name: testpod8
      labels:
        name: deployment
```

```
    spec:
      containers:
        - name: hc
          image: httpd
          ports:
            - containerPort: 80
```

clusteripservice.yaml

```
---
 kind: Service              # Defines to create Service type Object
 apiVersion: v1
 metadata:
   name: democipservice
 spec:
   ports:
     - port: 80             # Containers port exposed
       targetPort: 80       # Pods port
   selector:
     name: deployment       # Apply this service to any pods which has the specific label
   type: ClusterIP          # Specifies the service type i.e ClusterIP or NodePort
```

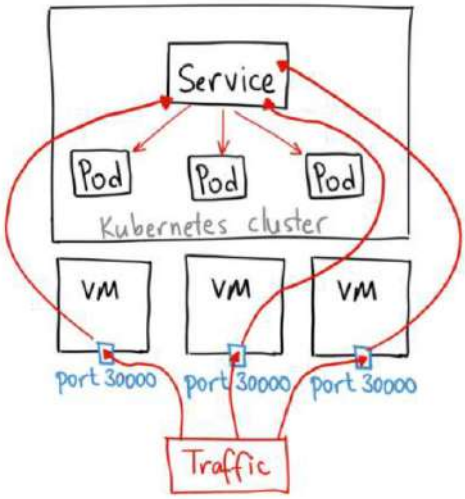
commands:

```
kubectl apply -f clusteripdeployment.yaml
kubectl apply -f clusteripservice.yaml
kubectl get pods
kubectl get svc
curl 10.96.78.101:80
kubectl delete pod mydeployments-779f4dbf96-rc89d
kubectl get pods
curl 10.96.78.101:80

kubectl delete -f clusteripdeployment.yaml
kubectl delete -f clusteripservice.yaml
```

NodePort

NodePort, as the name implies, opens a specific port on all the Nodes (the VMs). NodePort Exposes the Service on each Node's IP at a static port or A NodePort is an open port on every node of your cluster. any traffic that is sent to this port is forwarded to the service. is automatically created. You'll be able to contact the NodePort service, from outside the cluster, by using "<NodeIP>:<NodePort>".

- Kubernetes controller allocates a port from a range specified by (typically 30000–32767).
- If you don't specify this port, it will pick a random port range from (typically 30000–32767). If you want a specific port number, you can specify a value in the NodePort field.
- NodePort Services are easy to create but hard to secure since its open the same port in all the nodes to public. and standard ports such as 80, 443 or 8443 are cannot be used. We can use ports only in a range between 3000-32767 and if the ip of the vm/nodes changes then we have to deal this issue, so we don't recommend this node port service in production environments.



Features of NodePort:

Exposes the Service on All Nodes:

The service is exposed on a specific port (NodePort) across all nodes in the cluster. External clients can connect to any node's IP address using this port to access the service.

Fixed Port Range:

NodePort uses a port in the range 30000-32767 by default. This range is configurable but generally, Kubernetes will automatically assign an available port within this range.

Access via Node IP:

The service becomes accessible at http://<NodeIP>:<NodePort>. You can access the service via any node in the cluster, and Kubernetes will route the traffic to the appropriate Pod.

Internally Accessible:

A NodePort service is also accessible inside the cluster via the assigned ClusterIP (if desired), providing flexibility for both internal and external communication.

Example of a NodePort Service:

Here's an example YAML configuration for creating a NodePort service:

```
apiVersion: v1
kind: Service
metadata:
  name: my-nodeport-service
spec:
  type: NodePort
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80          # Port that the service exposes inside the cluster
      targetPort: 8080     # Port that the Pods are listening on
      nodePort: 30007          # Optional: Specify the NodePort (or let Kubernetes assign one automatically)
```

In this example:

- port: 80: The service listens on port 80 inside the cluster.
- targetPort: 8080: Traffic received by the service on port 80 is forwarded to the Pods on port 8080.
- nodePort: 30007: Kubernetes exposes the service on port 30007 on each node (optional—if not specified, Kubernetes will automatically assign a port within the 30000-32767 range).

How NodePort Works:

1. **External Access**: When the service is created, it becomes accessible from outside the cluster at <NodeIP>:<NodePort>. You can access the service using the IP address of any node in the cluster.
2. **Traffic Routing**: Kubernetes routes the external traffic arriving at the NodePort to the corresponding Pods, based on the service's selector.
3. **Load Balancing**: If multiple Pods are running that match the service's selector, Kubernetes will load balance the traffic across those Pods.

Use Case for NodePort:

- **Development and Testing**: NodePort services are often used in development or testing environments where you need quick external access to your services without setting up a cloud load balancer or other complex networking.
- **Bare-Metal Clusters**: For Kubernetes clusters running on bare metal or on-prem environments where cloud provider load balancers (e.g., AWS, GCP) are not available, NodePort is a straightforward way to expose services externally.
- **Direct External Access**: It is useful when you need to expose a service externally for direct access via a node's IP and port.

Example of Accessing a NodePort Service:

Assume you have a NodePort service with the following properties:

- NodeIP: 192.168.1.10 (IP of one of your Kubernetes nodes)
- NodePort: 30007

You can access the service externally via: http://192.168.1.10:30007

Advantages of NodePort:

- **Simple Setup**: No external cloud load balancer is required, and you can expose services quickly.
- **Cluster-Wide Availability**: The service is accessible via any node in the cluster, ensuring redundancy if a node goes down.

Limitations of NodePort:

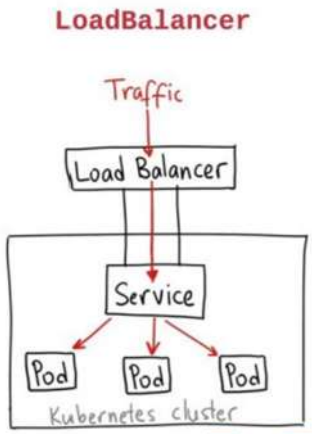
- **Limited Port Range**: NodePort services use a port from a specific range (30000-32767 by default), which may lead to port exhaustion in large-scale environments.
- **No Advanced Load Balancing**: While Kubernetes distributes traffic among Pods, NodePort doesn't offer advanced load balancing features (e.g., session persistence or health checks) that cloud load balancers provide.
- **Exposed to the Internet**: If the nodes have public IPs, the service is directly exposed to the internet, which might be a security concern if not properly secured.

Combining NodePort with Other Services:

- **Ingress**: In production environments, NodePort is often used in conjunction with an **Ingress** resource, where Ingress can act as a reverse proxy and load balancer, managing access to multiple services via a single external endpoint.
- **LoadBalancer**: In cloud environments, a **LoadBalancer** service type is often preferred for exposing services externally. Internally, it uses a NodePort under the hood but adds the additional benefits of cloud load balancer features.

LoadBalancer

In Kubernetes, a **LoadBalancer** service type provides a way to expose your application externally by automatically provisioning an external load balancer from a cloud provider (such as AWS, Google Cloud, or Azure). This service type enables users to route traffic to Pods from outside the Kubernetes cluster, leveraging the load balancing mechanisms provided by the cloud.



Features of LoadBalancer:

1. **External Access**:
   - The service is exposed via an external IP address (or DNS name) provided by the cloud provider's load balancer (e.g., AWS ELB, Google Cloud Load Balancer, Azure Load Balancer). This allows external traffic to access the service.
2. **Cloud Provider Integration**:
   - The LoadBalancer service type integrates with cloud providers, automatically creating and configuring a cloud load balancer when the service is created. The load balancer distributes traffic among the Pods that match the service's selector.
3. **Multi-Layer Load Balancing**:
   - LoadBalancer services combine two layers of load balancing:

1. **Cloud Provider Load Balancer**: Distributes traffic to nodes running in the Kubernetes cluster.
2. **Kubernetes NodePort**: Each node further distributes traffic to the appropriate Pods running on that node.

4. **Automatic Scaling**:
   - The cloud provider load balancer adjusts automatically as Pods scale up or down, ensuring that traffic is routed correctly to all available Pods.

Example of a LoadBalancer Service:

Here's an example YAML configuration for creating a LoadBalancer service:

```
---
 kind: Deployment
 apiVersion: apps/v1
 metadata:
  name: mydeployments
 spec:
  replicas: 1
  selector:      # tells the controller which pods to watch/belong to
   matchLabels:
    name: deployment
  template:
   metadata:
    name: testpod8
    labels:
     name: deployment
   spec:
    containers:
     - name: hc
       image: httpd
       ports:
        - containerPort: 80


apiVersion: v1
kind: Service
metadata:
 name: my-app-service
 annotations:
  service.beta.kubernetes.io/aws-load-balancer-type: "nlb"  # or "clb" for Classic Load Balancer
spec:
 selector:
  app: deployment
 ports:
  - protocol: TCP
    port: 80          # Expose on port 80
    targetPort: 8080     # Target container port
 type: LoadBalancer      # Create an ELB
```

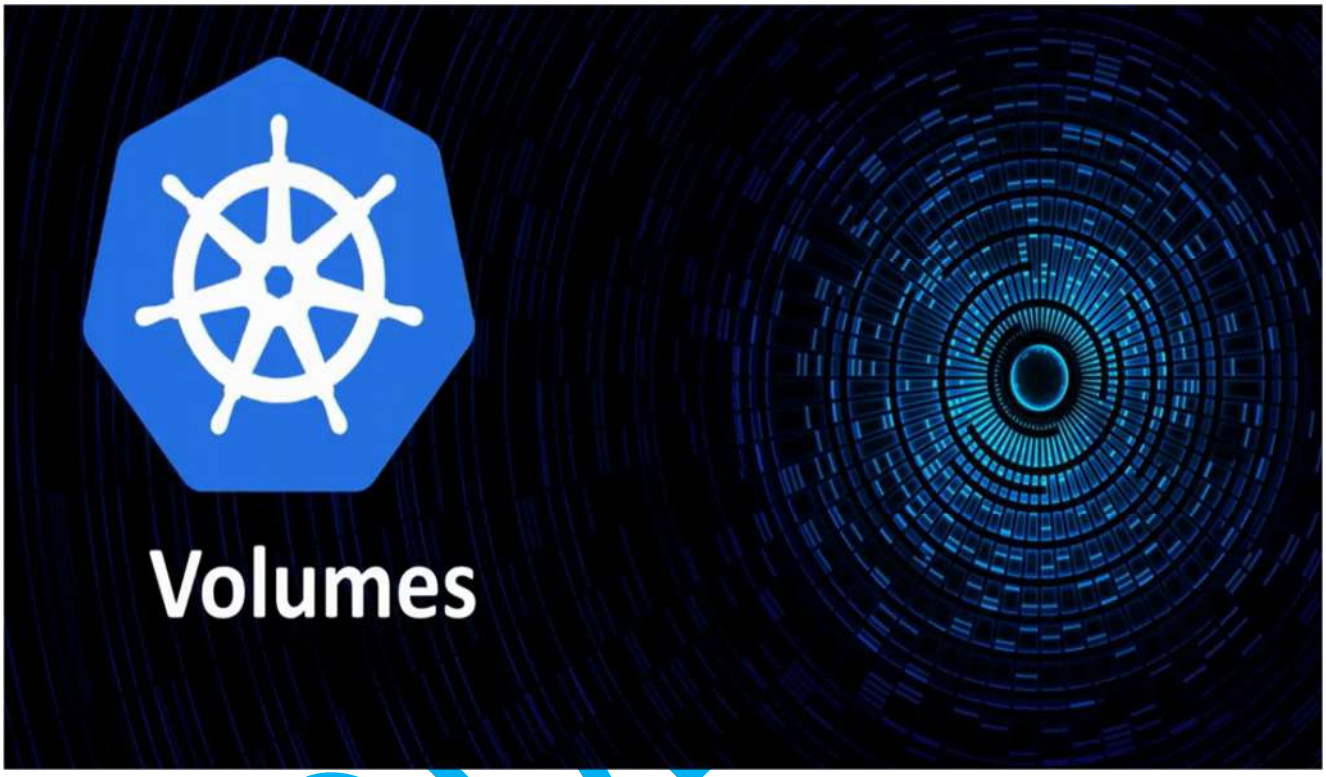
In this example:

- **type: LoadBalancer**: This tells Kubernetes to create an external load balancer.
- **port: 80**: The external load balancer will expose port 80.
- **targetPort: 8080**: Traffic received on port 80 is routed to port 8080 on the Pods.
- **selector**: The service forwards traffic to any Pods with the label app= deployment.

Cloud Provider-Specific Behaviour:

Each cloud provider may have different configurations and features for their load balancers:

- **AWS**: Creates an Elastic Load Balancer (ELB) or a Network Load Balancer (NLB) based on the Kubernetes configuration.
- **Google Cloud**: Creates a Google Cloud Load Balancer.
- **Azure**: Creates an Azure Load Balancer.



Prepared by PPNREDDY .... 😊 😊 😊

Volumes

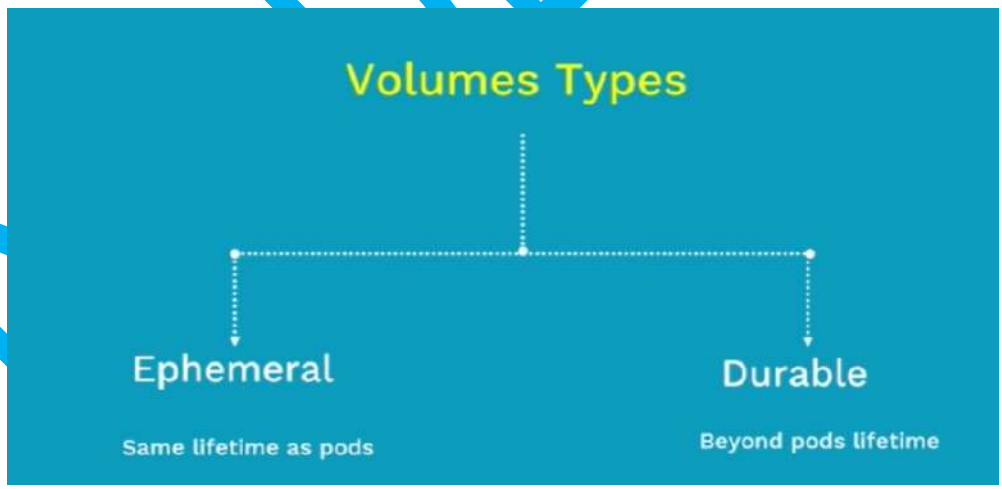
In Kubernetes, **volumes** are used to provide persistent storage for containers. Unlike container file systems, which are ephemeral and vanish when the container restarts or is destroyed, Kubernetes volumes allow data to persist beyond the container's lifecycle.

In Kubernetes, a volume can be thought of as a directory which is accessible to the containers in a pod. We have different types of volumes in Kubernetes and the type defines how the volume is created and its content.

The concept of volume was present with the Docker; however, the only issue was that the volume was very much limited to a particular pod. As soon as the life of a pod ended, the volume was also lost.

On the other hand, the volumes that are created through Kubernetes is not limited to any container. It supports any or all the containers deployed inside the pod of Kubernetes. A key advantage of Kubernetes volume is, it supports different kind of storage wherein the pod can use multiple of them at the same time.

- Containers are ephemeral (short lived in nature).
- All data stored inside a container is deleted if the container crashes. However, the kubelet will restart it with a clean state, which means that it will not have any of the old data
- To overcome this problem, Kubernetes uses Volumes. A Volume is essentially a directory backed by a storage medium. The storage medium and its content are determined by the Volume Type.
- In Kubernetes, a Volume is attached to a Pod and shared among the containers of that Pod.
- The Volume has the same life span as the Pod, and it outlives the containers of the Pod -this allows data to be preserved across container restarts.



Advantages of Volumes

Kubernetes volumes offer several key advantages, especially when managing containers in dynamic and distributed environments. Below are some of the main benefits:

2. **Persistent Storage**
   - **Ephemeral vs Persistent**: By default, data inside containers is ephemeral and lost when the container is terminated. Kubernetes volumes allow data to persist beyond the lifecycle of a container, ensuring critical data is not lost when a pod is restarted or rescheduled.
   - **PersistentVolume (PV) and PersistentVolumeClaim (PVC)** mechanisms make it easy to provision persistent storage that survives pod restarts and can be reused across containers.

3. **Decoupled Storage Management**
   - Kubernetes volumes decouple storage from the lifecycle of the container. You can destroy or recreate containers without losing the data stored in volumes.
   - This allows developers to focus on building applications without worrying about losing state, while DevOps or administrators can manage the underlying storage infrastructure.

4. **Flexibility in Storage Backend**
   - Kubernetes supports various storage backends including local storage, cloud providers (AWS EBS, GCE Persistent Disk, Azure Disks), NFS, Ceph, GlusterFS, and more.
   - The **Container Storage Interface (CSI)** standard allows third-party storage providers to integrate seamlessly, offering flexibility in choosing the best storage solution for your application needs.

5. **Data Sharing Across Containers**
   - Volumes can be used to share data between different containers within the same pod. This enables use cases like data aggregation, caching, and inter-container communication without needing external storage systems.

5. **Data Security and Configuration Management**
   - Kubernetes **secret** and **configMap** volumes allow sensitive data (e.g., passwords, tokens) and configuration files to be injected into pods securely and easily. These values can be mounted into containers without hardcoding them, enhancing security and maintainability.
   - Secrets can be mounted as encrypted volumes, ensuring sensitive data is managed safely.

6. **Dynamic Storage Provisioning**
   - Kubernetes supports **dynamic provisioning** of storage, meaning PersistentVolumes can be automatically created when a PersistentVolumeClaim is made by a pod. This eliminates the need for pre-provisioning storage manually.
   - This is particularly useful in cloud environments where storage needs can fluctuate dynamically and automation is key to handling this without human intervention.

7. **Cross-Node Data Access**
   - In cluster environments, where pods may be rescheduled to different nodes, Kubernetes volumes can ensure that data is accessible across nodes via cloud storage backends, distributed file systems (like Ceph or GlusterFS), or NFS.
   - This allows for a highly resilient and scalable infrastructure that can move workloads around without losing data.

8. **Multiple Access Modes**
   - Kubernetes allows defining access modes for volumes, such as **ReadWriteOnce** (can be mounted for reading and writing by a single node) or **ReadWriteMany** (can be mounted by multiple nodes). This gives flexibility in defining how storage is shared and used.
   - This is particularly useful for different workloads, like databases (which often require exclusive access) or logging systems that need shared access.

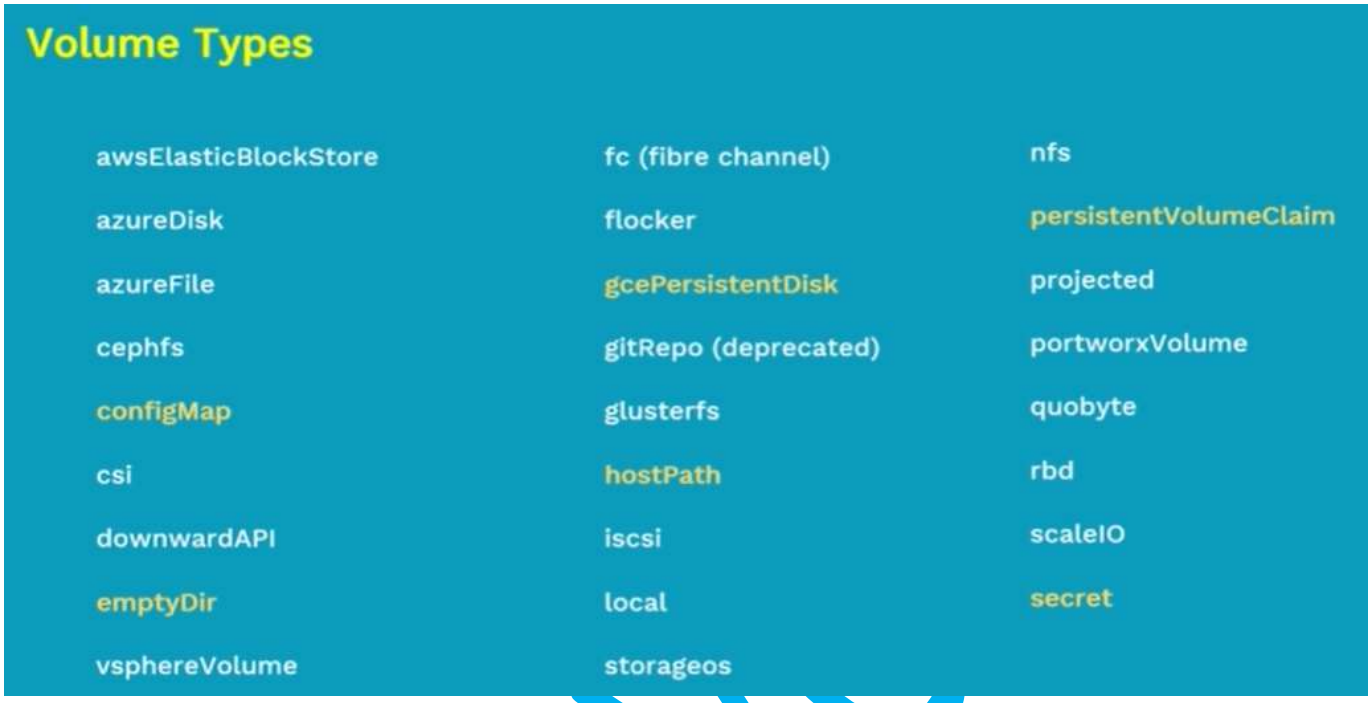
9. **Backup and Recovery**
   - Kubernetes volumes, especially PersistentVolumes, can be used in conjunction with storage systems that support snapshotting and backup functionality. This simplifies setting up disaster recovery plans.
   - Cloud-native storage backends often provide automatic snapshot and replication features, adding another layer of protection against data loss.

10. **Scalability**
    - Kubernetes volumes are ideal for scaling stateful applications. Using **StatefulSets** in Kubernetes, you can manage the scaling of stateful applications while ensuring each replica has its own persistent volume.
    - Persistent volumes help in maintaining data consistency and integrity even when scaling up or down dynamically.

11. **Simplified Application Deployment**
    - By abstracting the storage management layer, Kubernetes makes it easy to provision and manage storage without requiring specialized knowledge of the underlying infrastructure.
    - Developers can focus on deploying applications while the storage admin can manage storage resources independently.

12. **Data Localization**
    - Kubernetes supports **hostPath** volumes, which allows containers to directly access the local filesystem of the node. This can be useful when you need high-performance access to files on a specific machine.

Types of Kubernetes Volume



Here is a list of some popular Kubernetes Volumes

- **emptyDir** – It is a type of volume that is created when a Pod is first assigned to a Node. It remains active as long as the Pod is running on that node. The volume is initially empty and the containers in the pod can read and write the files in the emptyDir volume. Once the Pod is removed from the node, the data in the emptyDir is erased.
- **hostPath** – This type of volume mounts a file or directory from the host node's filesystem into your pod.
- **gcePersistentDisk** – This type of volume mounts a Google Compute Engine (GCE) Persistent Disk into your Pod. The data in a **gcePersistentDisk** remains intact when the Pod is removed from the node.
- **awsElasticBlockStore** – This type of volume mounts an Amazon Web Services (AWS) Elastic Block Store into your Pod. Just like **gcePersistentDisk**, the data in an **awsElasticBlockStore** remains intact when the Pod is removed from the node.
- **nfs** – A **nfs** volume allows an existing NFS (Network File System) to be mounted into your pod. The data in a **nfs** volume is not erased when the Pod is removed from the node. The volume is only unmounted.
- **iscsi** – An **iscsi** volume allows an existing iSCSI (SCSI over IP) volume to be mounted into your pod.
- **flocker** – It is an open-source clustered container data volume manager. It is used for managing data volumes. A **flocker** volume allows a Flocker dataset to be mounted into a pod. If the dataset does not exist in Flocker, then you first need to create it by using the Flocker API.
- **glusterfs** – Glusterfs is an open-source networked filesystem. A glusterfs volume allows a glusterfs volume to be mounted into your pod.

- **rbd** – RBD stands for Rados Block Device. A **rbd** volume allows a Rados Block Device volume to be mounted into your pod. Data remains preserved after the Pod is removed from the node.
- **cephfs** – A **cephfs** volume allows an existing CephFS volume to be mounted into your pod. Data remains intact after the Pod is removed from the node.
- **gitRepo** – A **gitRepo** volume mounts an empty directory and clones a **git** repository into it for your pod to use.
- **secret** – A **secret** volume is used to pass sensitive information, such as passwords, to pods.
- **persistentVolumeClaim** – A **persistentVolumeClaim** volume is used to mount a PersistentVolume into a pod. PersistentVolumes are a way for users to "claim" durable storage (such as a GCE PersistentDisk or an iSCSI volume) without knowing the details of the particular cloud environment.
- **downwardAPI** – A **downwardAPI** volume is used to make downward API data available to applications. It mounts a directory and writes the requested data in plain text files.
- **azureDiskVolume** – An **AzureDiskVolume** is used to mount a Microsoft Azure Data Disk into a Pod.

emptyDir volume

Kubernetes emptyDir is an empty directory in the pod and it will be created when the pod is created and it will be deleted after pod deleted. If the pod is stopped also emptyDir volume will be available, but if the pod deleted you will lose entire data in emptyDir. emptyDir that lasts for the life of the Pod, even if the Container in the pod terminates and restarts.

Initially the emptyDir volume is empty. Pod/container can write and read data into this emptyDir volume. And this storage is persisting if the node is running. If the node goes down, the contents of the emptydir will be erased. emptydir volume will take memory from storage volumes of the node machine like SSD, or it can use RAM for higher performance.

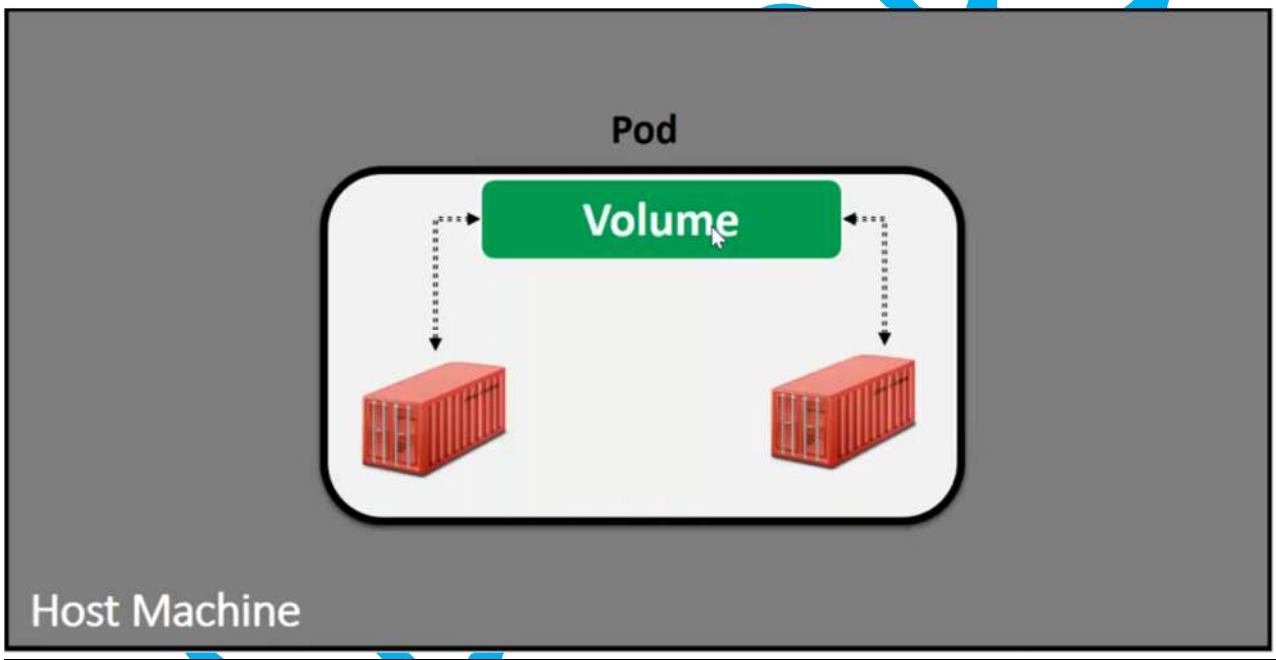
- Use this when we want to share contents between multiple containers on the same pod & not to the host machine
- An emptyDir volume is first created when a Pod is assigned to a Node, and exists as long as that Pod is running on that node.
- As the name says, it is initially empty.
- Containers in the Pod can all read and write the same files in the emptyDir volume, though that volume can be mounted at the same or different paths in each Container.
- When a Pod is removed from a node for any reason, the data in the emptyDir is deleted forever.
- A Container crashing does not remove a Pod from a node, so the data in an emptyDir volume is safe across Container crashes.

emptyDir

- Creates empty directory first created when a Pod is assigned to a Node,
- Stays as long as pod is running
- Once pod is removed from a node, emptyDir is deleted forever

Use cases:

- Temporary space



kubectl apply -f empty-dir-pods.yml

==========

```
---
 apiVersion: v1
 kind: Pod
 metadata:
   name: myvolemptydir
 spec:
   containers:
    - name: uc1
      image: ubuntu:latest
      command: ["/bin/bash", "-c", "sleep 10000"]
      volumeMounts:                    # Mount definition inside the container
```

```
      - name: xchange
        mountPath: "/tmp/xchange"        # Path inside the container to share
    - name: uc2
      image: ubuntu:latest
      command: ["/bin/bash", "-c", "sleep 10000"]
      volumeMounts:
       - name: xchange
         mountPath: "/tmp/data"
  volumes:                          # Definition for host
   - name: xchange
     emptyDir: {}
```

Commands:
=========

```
kubectl apply -f empty-dir-pods.yml
kubectl exec myvolemptydir -c uc1 -it -- /bin/bash
cd /tmp/xchange
touch uc1.txt

kubectl exec myvolemptydir -c uc2 -it -- /bin/bash
cd /tmp/data/
ls
touch uc2.txt
kubectl delete -f empty-dir-pods.yml
```


hostPath volume

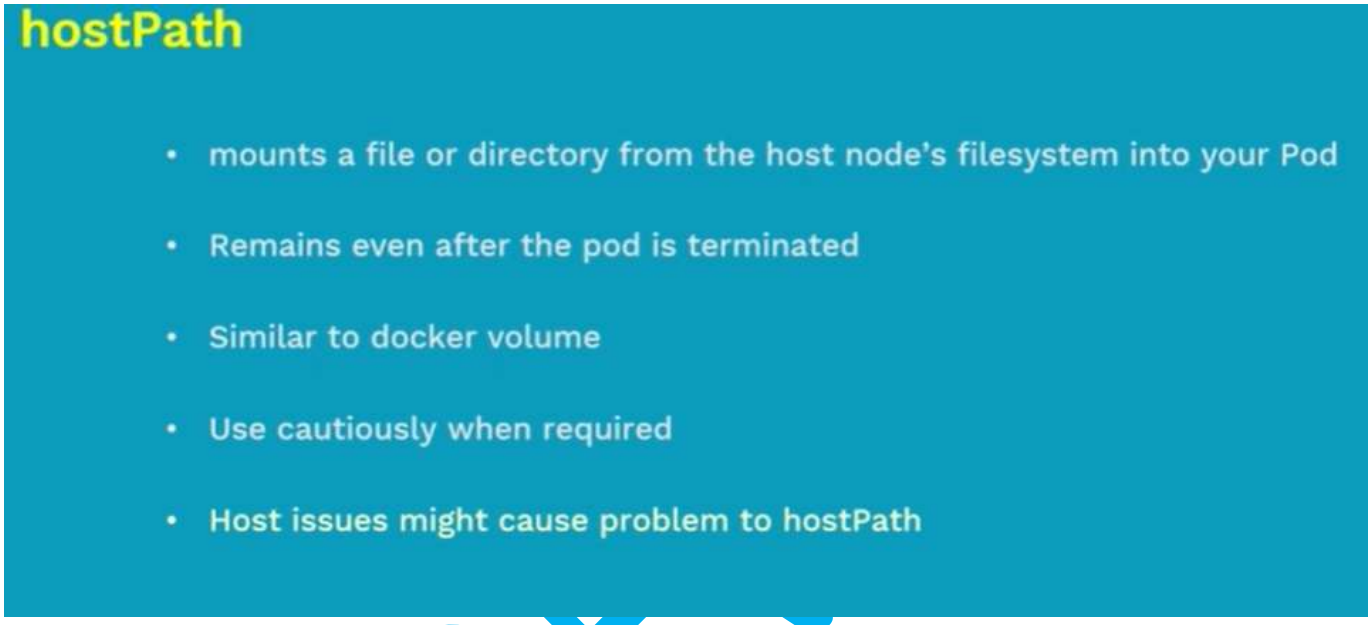
A **Kubernetes** hostPath volume mounts a file or directory from the host node's filesystem into your Pod. Kubernetes supports hostPath for development and testing on a single-node cluster. In a production cluster we would not use Kubernetes hostPath. Even if the pod dies, the data is persisted in the host machine. In order for hostPath to work, you will need to run a single node cluster. Kubernetes does not support host path for multimode cluster. Since There is no guarantee that your pod will end up on the correct node where the HostPath resides.

Some uses for a Kubernetes hostPath

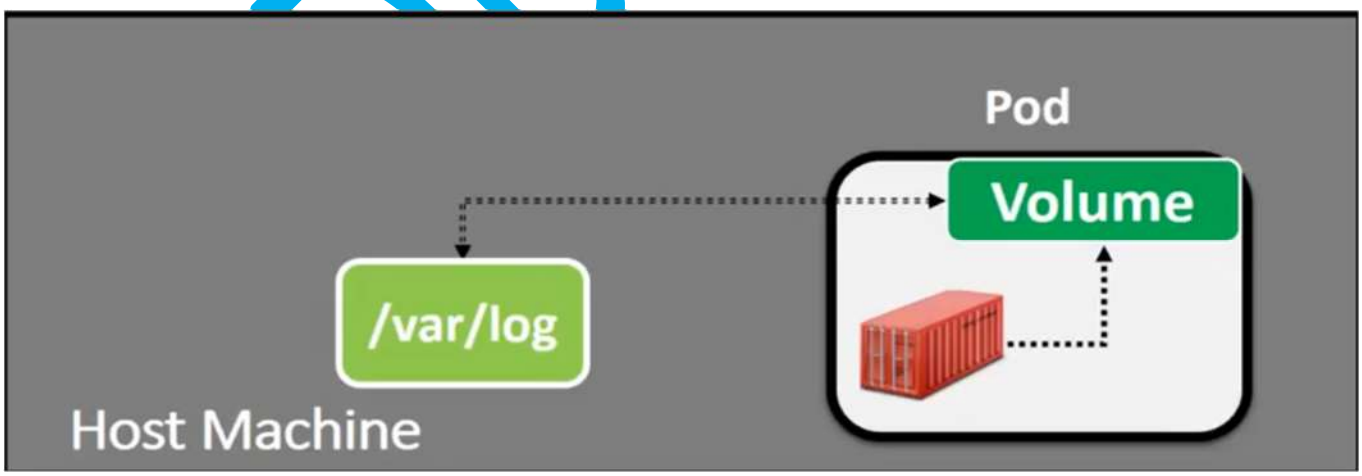
- Running a container that needs access to Docker internals; use a hostPath of /var/lib/docker
- Running cAdvisor in a container; use a hostPath of /dev/cgroups

How to use Hostpath in Multinode cluster

When the host path is going to create in the pod. Kubernetes scheduler will check on that node, is that host path is existed or not. But most of the times host path is not available in all nodes since it is a multinode cluster. So the Kubernetes will create an empty directory on the node. And it will mount this empty directory into container. So, if you want to mount any node directory into container use nodeselector. Using node selector, we can use hostPath in multinode cluster otherwise your empty directory will be mounted.



- Use this when we want to access the content of a pod/container from host machine.
- A hostPath volume mounts a file or directory from the host node's filesystem into your Pod.



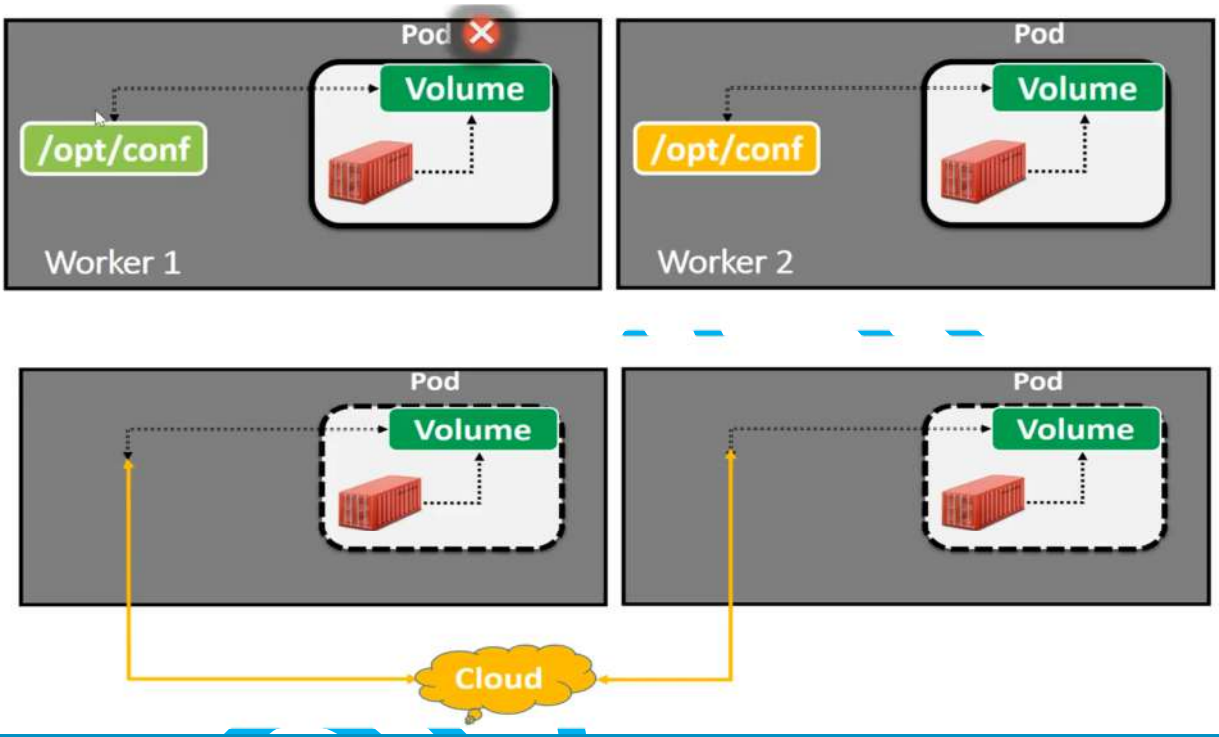
hostpath-pods.yml

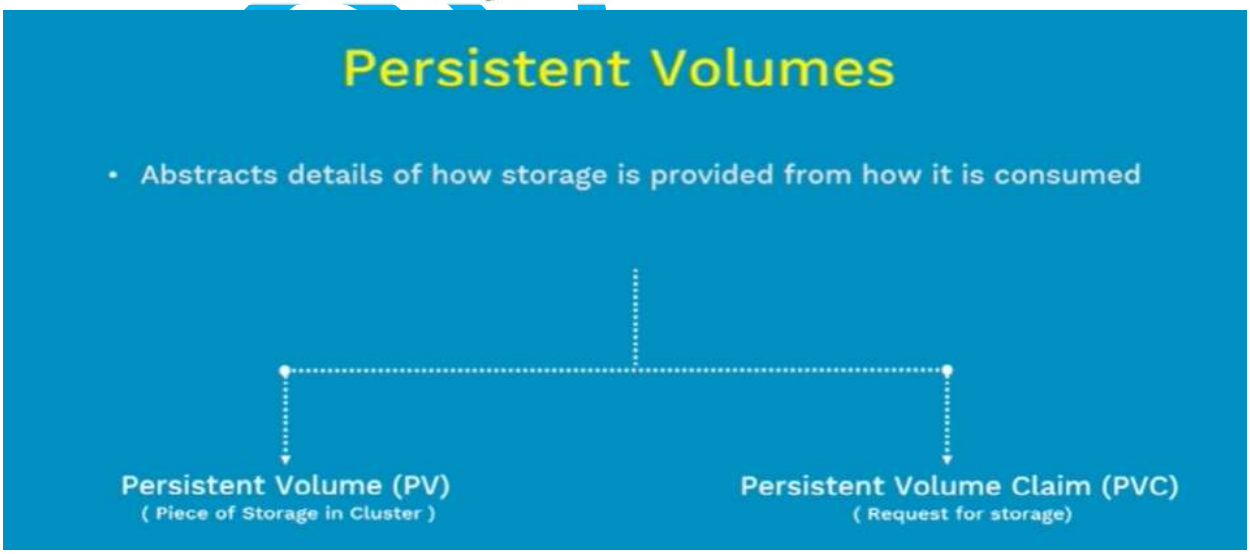
```
==================
---
 apiVersion: v1
 kind: Pod
 metadata:
  name: myhostpath
 spec:
  containers:
   - name: uc1
     image: ubuntu:latest
     command: ["/bin/bash", "-c", "sleep 10000"]
     volumeMounts:                  # Mount definition inside the container
      - mountPath: /tmp/hostpath
        name: testvolume
  volumes:                          # Definition for host
   - name: testvolume
     hostPath:
       path: /tmp/data
       type: DirectoryOrCreate

Commands:
=========
kubectl apply -f hostpath-pods.yml
kubectl get pods -o wide

verify the host where it's created
kubectl delete -f hostpath-pods.yml
```

PersistentVolumes (PV) && PersistentVolumeClaim (PVC)

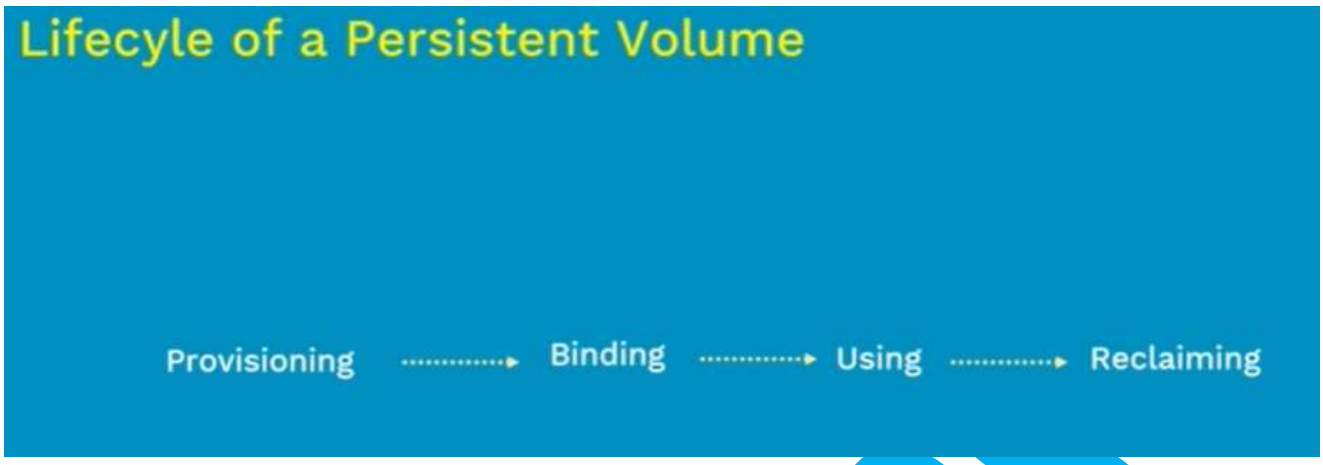
A PersistentVolume (PV) is a piece of pre-provision storage inside the cluster in short this are called as PV. This is typically provided by administrator. As it names says these volumes are persistent meaning the data inside volume exist beyond the lifecycle of individual POD that uses this PV.

A PersistentVolumeClaim (PVC) is a storage request by a user typically this user is a developer in short PersistentVolumeClaim called as PVC. So, developers are requested for storage some capacity along with access modes such as ReadWrite OR ReadOnly





Lifecycle of a PersistentVolume



Provisioning:

In Provisioning stage typically administrator create the storage chunks which is called as PersistentVolume.

Binding:

Developer requests the storage request using the PersistentVolumeClaim, Once the storage request finds the suitable size of a PersistentVolume then it gets bounded.

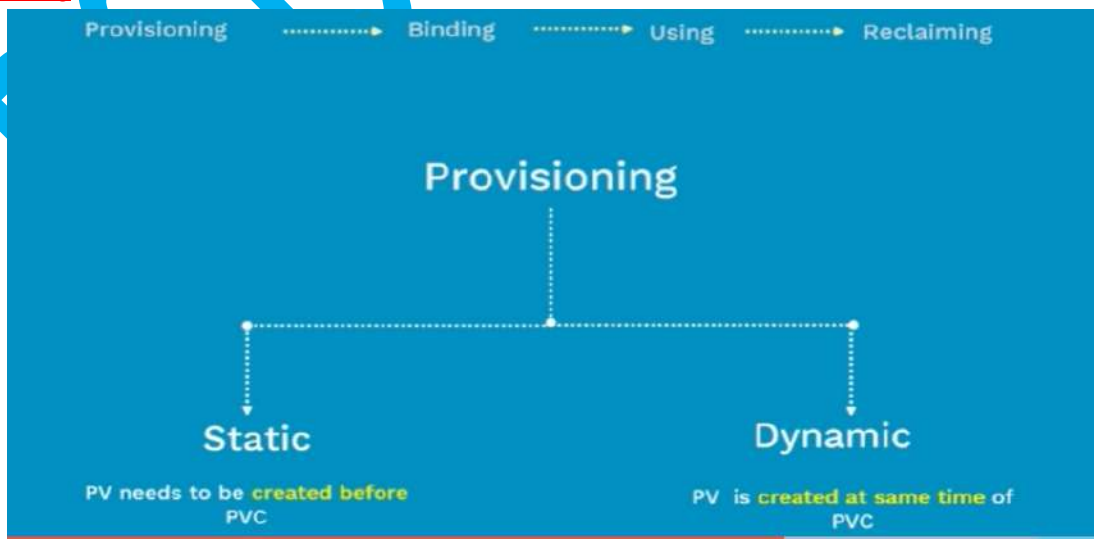
Using:

Developers can use this volume inside the POD to mount the volume on a mount point and using.

Reclaiming:

Finally, a user done in this volume they can delete the PersistentVolumeClaim objects from the Kubernetes so which allows reclaiming the resources.

Provisioning

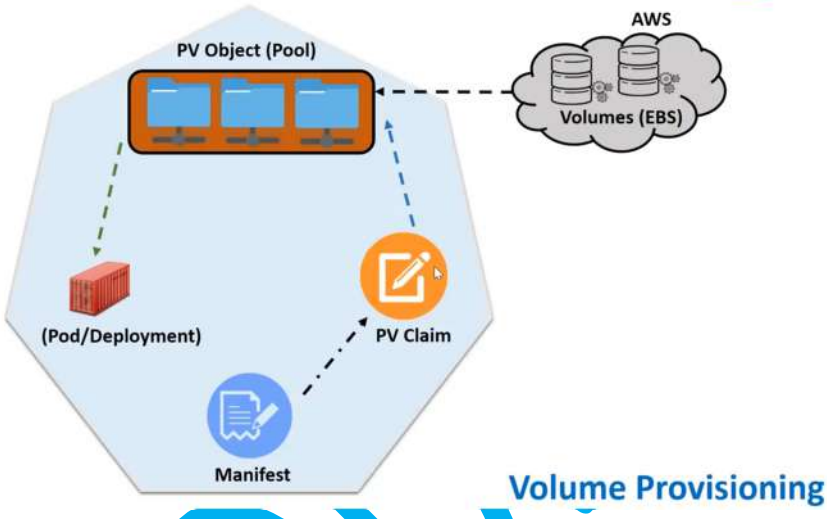


In Kubernetes, **static** and **dynamic provisioning** are two methods used for managing storage resources, particularly when dealing with **PersistentVolumes (PVs)** and **PersistentVolumeClaims (PVCs)**. Each has its own advantages, and understanding the benefits of both is crucial when designing a robust storage architecture.

1. Static Provisioning (PV needs to be created before PVC)
2. Dynamic provisioning (PV is created at same time of PVC)

Static Provisioning

In static provisioning, an administrator manually creates **PersistentVolumes (PVs)** that can later be claimed by users through **PersistentVolumeClaims (PVCs)**. The storage is pre-provisioned, and users bind to these pre-created volumes.



Volume Provisioning

Advantages of Static Provisioning:

1. Control Over Storage Configuration

Administrators have full control over how the storage is provisioned and configured. This includes specific performance, replication, backup, and resiliency settings tailored to the needs of each application or workload.

Ideal for environments where compliance or security requires strict control over how storage is allocated.

2. Predefined Storage Allocation

Since storage volumes are provisioned ahead of time, there is no risk of storage being dynamically misconfigured. Administrators can ensure that storage is allocated from specific storage backends or configurations that meet organizational policies or performance criteria.

3. Predictable Storage Performance

Administrators can assign specific storage resources for different applications, ensuring predictable and consistent performance. For example, high-performance storage can be allocated for critical databases, while lower-cost storage can be used for other workloads.

4. No Over-provisioning Risks

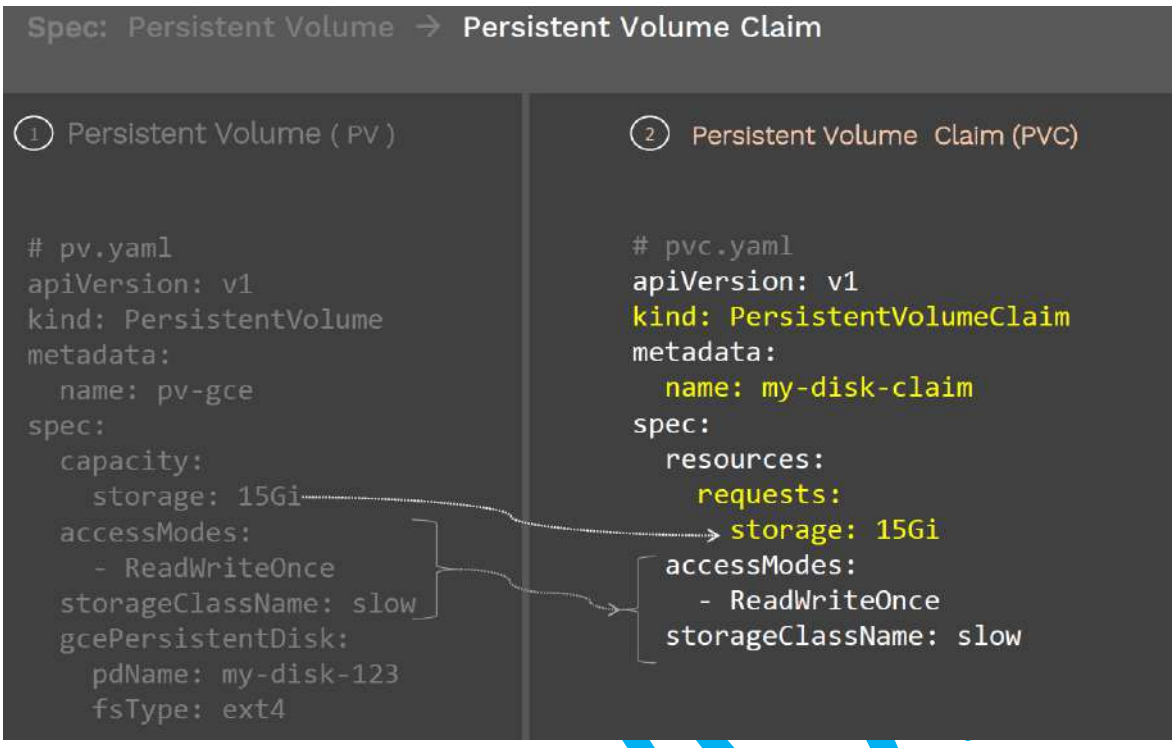
Because the volumes are manually created and sized, there's no risk of unintentional over-provisioning of storage, which can happen if resources are dynamically allocated in response to PVC requests without careful control.

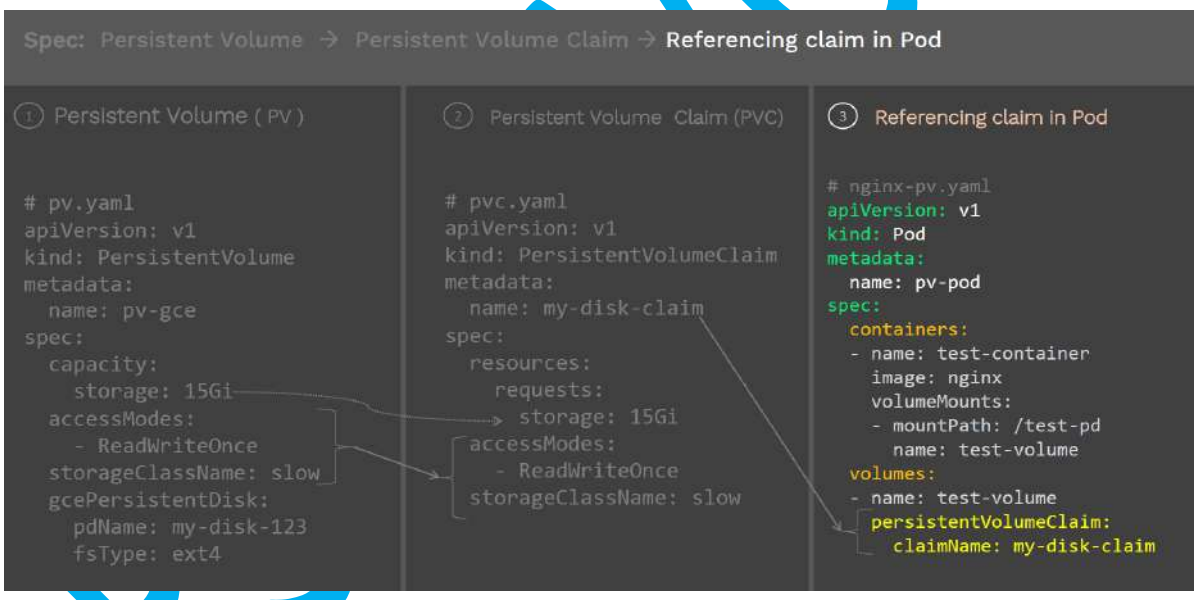
5. Legacy Systems and External Storage

In environments with legacy storage systems or specific external storage arrays, static provisioning allows administrators to integrate Kubernetes with those existing systems. This is especially useful when using external network storage, like NFS or SAN systems, that require careful manual configuration.

6. Guaranteed Resource Availability

Pre-provisioned volumes ensure that storage resources are always available when a pod requests them, eliminating the uncertainty that might arise in dynamic environments where storage resources could be temporarily unavailable.





Demo

kubectl apply -k "github.com/kubernetes-sigs/aws-ebs-csi-driver/deploy/kubernetes/overlays/stable/ecr/?ref=release-1.22" kubectl apply -k "github.com/kubernetes-sigs/aws-efs-csi-driver/deploy/kubernetes/overlays/stable/ecr/?ref=release-1.6"

Prepared by PPNREDDY .... 😊 😊 😊

pv.yaml

```
=======
apiVersion: v1
kind: PersistentVolume
metadata:
  name: efs-pv
spec:
  capacity:
    storage: 5Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Retain
  storageClassName: efs-sc
  csi:
    driver: efs.csi.aws.com
    volumeHandle: fs-012f3739be582185a # Replace with your EFS File System ID
```

pvc.yaml

```
========
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: efs-pvc
spec:
  accessModes:
    - ReadWriteMany
  storageClassName: efs-sc
  resources:
    requests:
      storage: 5Gi
```

deployment.yaml

```
===============
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
```

```
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:latest
          volumeMounts:
            - name: shared-storage
              mountPath: /usr/share/nginx/html
      volumes:
        - name: shared-storage
          persistentVolumeClaim:
            claimName: efs-pvc
```

Commands:
=========

kubectl apply -f .

Verify the pod activities

kubectl delete -f .

Dynamic provisioning

Dynamic provisioning automates the creation of PersistentVolumes when a **PersistentVolumeClaim** is made. Instead of manually creating PVs, Kubernetes interacts with a **StorageClass** to provision storage on demand based on the PVC's requirements.

Advantages of Dynamic Provisioning:

1. Simplified Storage Management

Dynamic provisioning reduces the administrative burden by automating the creation of storage volumes when needed. There's no need to pre-create and manage volumes manually, making it easier for teams to manage large clusters with many workloads.

2. Scalability

Dynamic provisioning scales effortlessly with the growth of your applications. As new applications or workloads are deployed, storage is automatically provisioned based on the PVC's specifications, supporting rapid and dynamic scaling of storage.

3. Optimized Resource Utilization

Since storage is only created when a PVC is requested, dynamic provisioning reduces the risk of wasting unused storage resources. This leads to more efficient storage utilization and cost savings, particularly in cloud environments where storage is billed based on actual usage.

4. Fast Deployment

Developers can easily claim storage without waiting for an administrator to provision resources. This accelerates the deployment and iteration cycles, especially in DevOps environments, where automation and speed are critical.

5. Consistency Through Storage Classes

StorageClass objects allow administrators to define specific types of storage based on performance, replication, or cost. Dynamic provisioning uses these classes to automatically provide the right type of storage to applications without manual intervention.

This ensures consistency across deployments, where each application gets the storage type and configuration it needs without administrator involvement.

6. Cloud-Native Integration

Dynamic provisioning is ideal for cloud-native environments where resources can be automatically provisioned through cloud providers (e.g., AWS EBS, Google Persistent Disk, Azure Disks). Cloud platforms provide APIs that Kubernetes interacts with to dynamically allocate the storage.

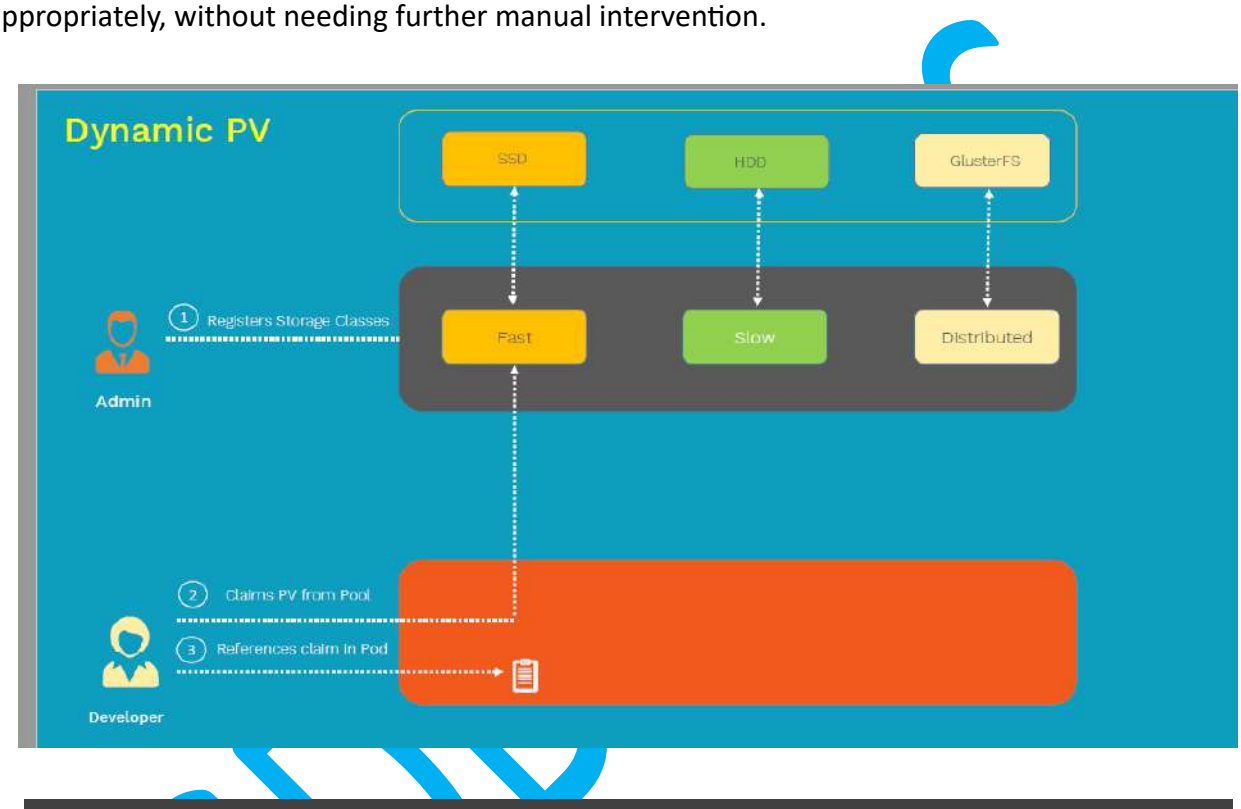
It's especially useful in autoscaling environments, where applications or clusters scale dynamically, and additional storage resources are needed in real time.

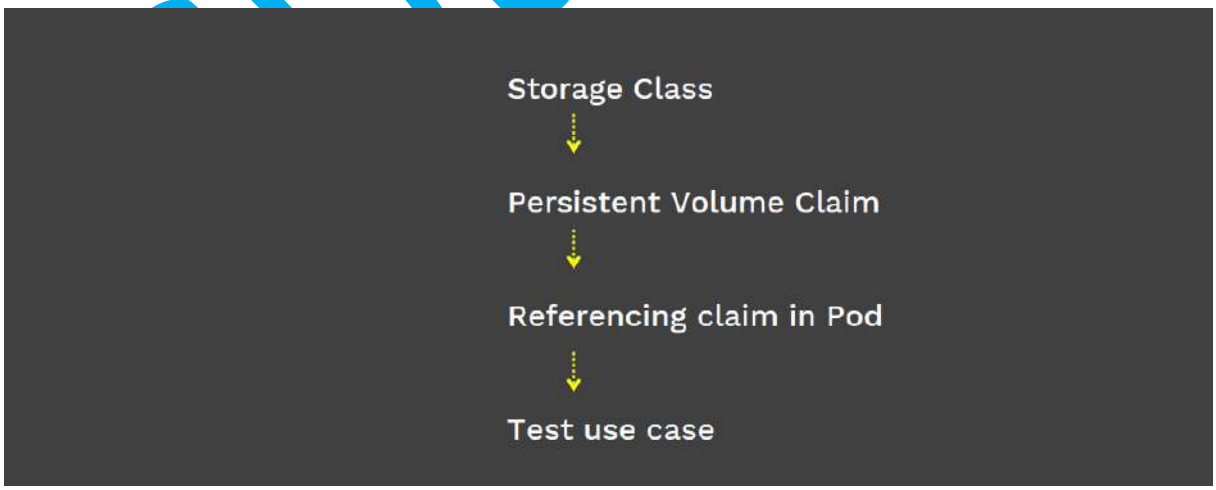
7. Fewer Storage Admins Required

Dynamic provisioning reduces the need for dedicated storage administrators by automating much of the resource allocation process. This is particularly beneficial in organizations with small teams or those focused on maximizing operational efficiency.

8. Reduced Storage Complexity

The ability to use predefined storage classes reduces the complexity of managing various types of storage. Administrators define these classes once, and Kubernetes ensures they are used appropriately, without needing further manual intervention.





StorageClass – Manifest file

① Storage Class

```yaml
# sc.yaml
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: fast
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-ssd
```

StorageClass – Create & Display

```
schalla@master:$ kubectl create -f sc.yaml
storageclass "fast" created
```

```
schalla@master:$ kubectl get storageclass
NAME                 PROVISIONER            AGE
fast                 kubernetes.io/gce-pd   37s
standard (default)   kubernetes.io/gce-pd   4d
```

```
schalla@master:$ kubectl describe storageclass fast
Name:            fast
IsDefaultClass:  No
Annotations:     <none>
Provisioner:     kubernetes.io/gce-pd
Parameters:      type=pd-ssd
ReclaimPolicy:   Delete
Events:          <none>
```

Persistent Volume Claim(PVC) - Config

① Storage Class

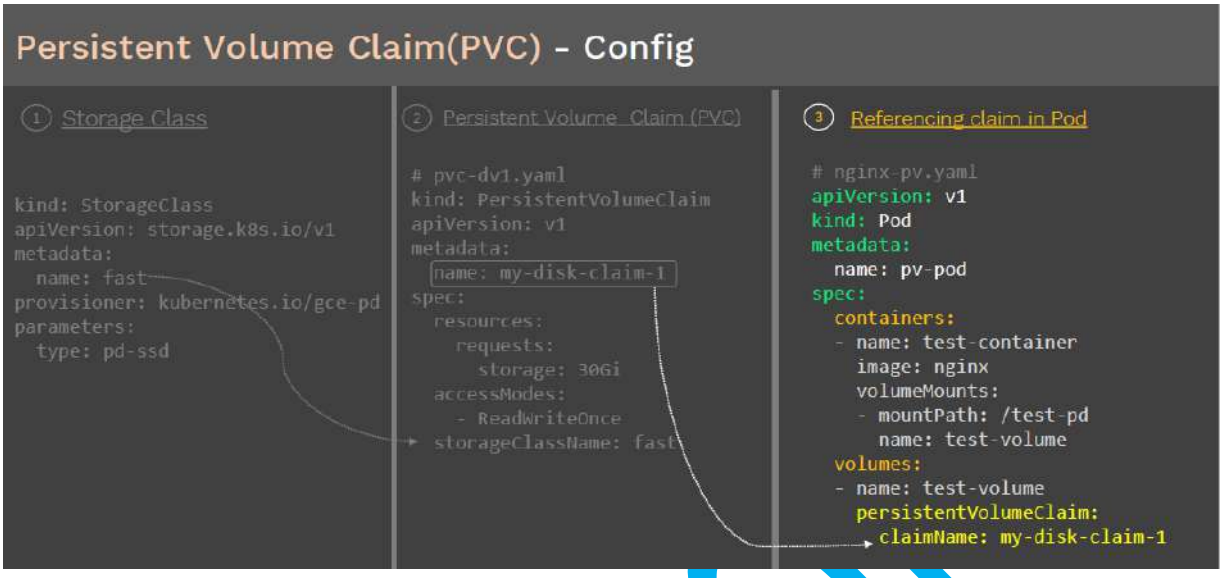
```yaml
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: fast
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-ssd
```

② Persistent Volume Claim (PVC)

```yaml
# pvc-dv1.yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: my-disk-claim-1
spec:
  resources:
    requests:
      storage: 30Gi
  accessModes:
    - ReadWriteOnce
  storageClassName: fast
```

```yaml
# pvc-dv2.yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: my-disk-claim-2
spec:
  resources:
    requests:
      storage: 40Gi
  accessModes:
    - ReadWriteOnce
  storageClassName: fast
```

Prepared by PPNREDDY .... ☺ ☺ ☺



Demo

storageclass.yaml

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: efs-sc
provisioner: efs.csi.aws.com
parameters:
  provisioningMode: efs-ap # Automatic creation of access points
  fileSystemId: fs-012f3739be582185a # Replace with your EFS File System ID
  directoryPerms: "777" # Default directory permissions
  gidRangeStart: "1000" # Optional group ID range
  gidRangeEnd: "2000" # Optional group ID range
```

pvc.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: efs-pvc
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 5Gi
  storageClassName: efs-sc
```

deloyment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:latest
          ports:
            - containerPort: 80
          volumeMounts:
            - name: shared-storage
              mountPath: /usr/share/nginx/html
      volumes:
        - name: shared-storage
          persistentVolumeClaim:
            claimName: efs-pvc
```

Commands

========

kubectl apply -f .

perform pod activities

kubectl delete -f .

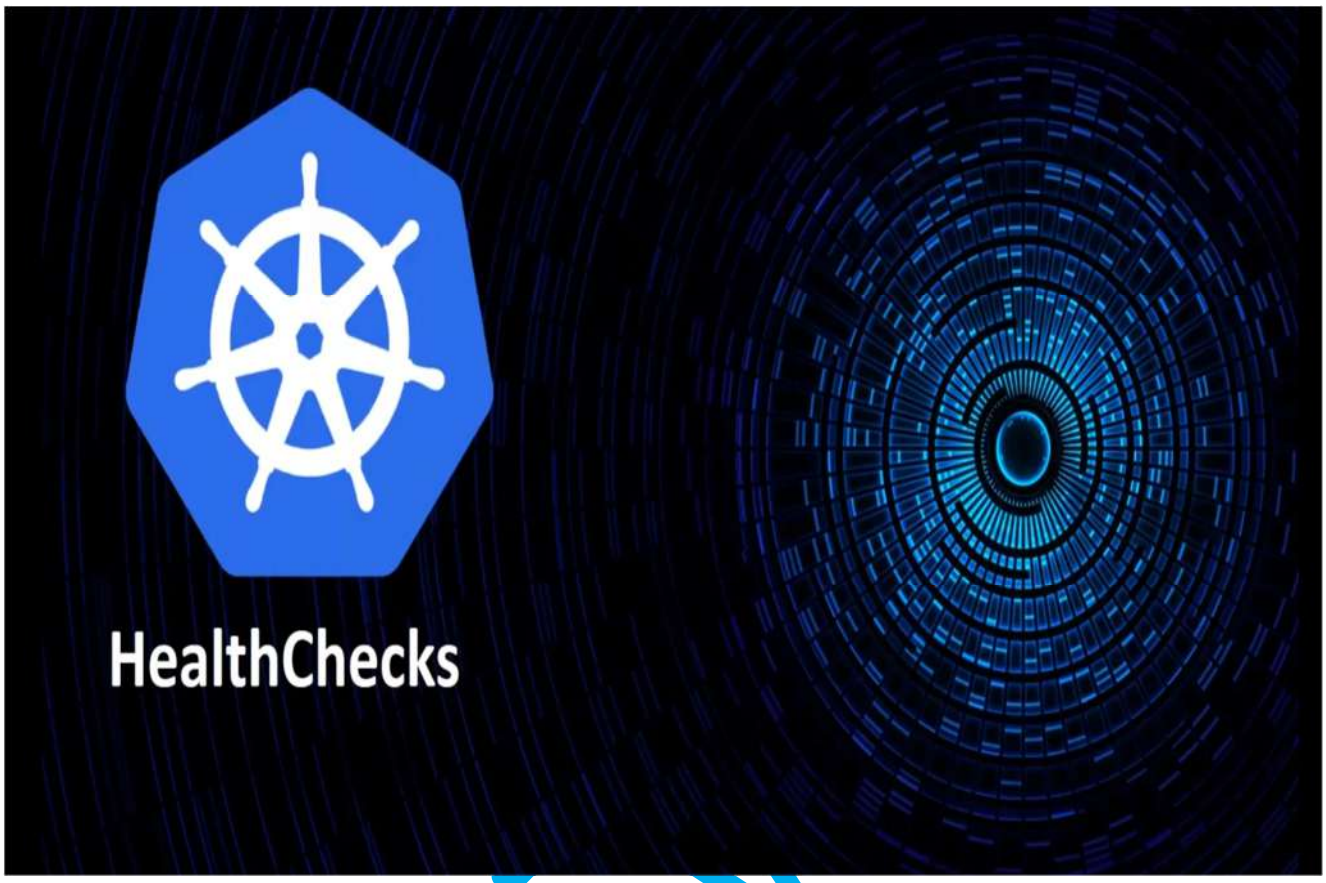
Comparison of Static vs Dynamic Provisioning

Aspect	Static Provisioning	Dynamic Provisioning
Control	Full control over how volumes are created and managed	Storage is provisioned automatically based on PVC requests
Administrative Burden	High, as storage must be pre-created and managed manually	Low, as storage is provisioned on-demand without manual work
Scalability	Limited to pre-provisioned storage	Scales dynamically based on needs
Resource Utilization	May lead to unused or underutilized storage	Optimized for on-demand resource utilization
Speed	Slower, as manual provisioning is required	Faster, as storage is automatically created when needed
Storage Type Flexibility	Can integrate with legacy or specific storage systems	Uses cloud-native or modern storage backends
Configuration Consistency	Fully customizable per volume	Consistency enforced through StorageClass definitions

Note:

Static provisioning is best suited for environments where storage configurations need to be tightly controlled, predictable, or integrated with existing systems.

Dynamic provisioning excels in highly scalable, cloud-native, and automated environments where ease of use, flexibility, and resource optimization are critical. It significantly reduces the overhead involved in managing storage, making it ideal for DevOps practices and rapid application development.



Prepared by PPNREDDY .... 😊 😊 😊

Liveness Probe

In Kubernetes, a *Liveness Probe* is a mechanism used to determine whether a container with in a pod is still running correctly. If the liveness probe fails, Kubernetes will automatically restart the container to try to recover it. This ensures that unhealthy containers do not continue to run and affect the application's availability.

Why Use Liveness Probes?

Containers may sometimes enter a non-responsive state (e.g., due to deadlocks, memory issues, or application bugs) while still being technically "alive." A liveness probe helps detect and recover from such situations by restarting the problematic container without needing to restart the entire pod or service manually.

Types of Liveness Probes

Kubernetes supports several methods for probing the health of a container:

1. HTTP GET Probe:
   - Kubernetes sends an HTTP GET request to the container. If the response has a status code between 200 and 399, the container is considered healthy.
   - Common for applications with an HTTP interface, such as web servers or REST APIs.

2. TCP Socket Probe:
   - Kubernetes attempts to open a TCP connection to a specified port on the container. If the connection is successful, the container is deemed healthy.
   - Useful for applications that listen on a specific port.

3. Exec Command Probe:
   - Kubernetes runs a command inside the container. If the command returns a 0-exit code, the container is considered healthy.
   - Suitable for checking specific internal conditions, such as a process status, file existence, or resource usage.

Example Configuration

Here's an example of configuring a liveness probe in a pod's spec:

HTTP GET Probe Example

This liveness probe checks an endpoint /health on port 8080:

```
apiVersion: v1
kind: Pod
metadata:
  name: liveness-http-example
spec:
  containers:
```

```
  - name: my-app
    image: my-app-image
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
      initialDelaySeconds: 10
      periodSeconds: 5
```

initialDelaySeconds: The number of seconds to wait after the container starts before performing the first probe.
periodSeconds: How often (in seconds) to perform the probe.

TCP Socket Probe Example

This liveness probe checks if a TCP connection can be established on port 3306

```
apiVersion: v1
kind: Pod
metadata:
  name: liveness-tcp-example
spec:
  containers:
  - name: my-db
    image: my-db-image
    livenessProbe:
      tcpSocket:
        port: 3306
      initialDelaySeconds: 15
      periodSeconds: 10
```

Exec Command Probe Example

This liveness probe runs a command to check if the file /tmp/healthy exists. If it does, the container is healthy:

```
apiVersion: v1
kind: Pod
metadata:
  name: liveness-exec-example
spec:
  containers:
  - name: my-app
    image: my-app-image
```

```
  livenessProbe:
    exec:
      command:
      - cat
      - /tmp/healthy
    initialDelaySeconds: 5
    periodSeconds: 5
```

Important Probe Parameters

initialDelaySeconds: Time to wait after the container starts before performing the probe. This is useful for applications that take some time to start up.
periodSeconds: How often the probe runs.
timeoutSeconds: How long to wait for the probe to complete before considering it a failure.
failureThreshold: How many consecutive probe failures Kubernetes tolerates before restarting the container.
successThreshold: The number of consecutive successes required to determine that the container is healthy (mostly used in readiness probes).

Use Cases for Liveness Probes

Deadlock detection: If an application gets stuck in an unrecoverable state, a liveness probe can detect the issue and restart the container.
Service health monitoring: By periodically checking endpoints or internal conditions, the probe can ensure that unhealthy containers do not continue running.
Graceful degradation: Sometimes a container might be partially functioning but unable to serve requests. Liveness probes can help Kubernetes detect and address these cases efficiently.

liveness-probe.yaml

```
==================
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: mylivenessprobe
spec:
  containers:
  - name: liveness
    image: ubuntu
    args:
    - /bin/sh
```

```
  - -c
  - touch /tmp/healthy; sleep 1000
  livenessProbe:                    # define the health check
    exec:
      command:                      # command to run periodically
      - cat
      - /tmp/healthy
    initialDelaySeconds: 5      # Wait for the specified time before it runs the first probe
    periodSeconds: 5          # Run the above command every 5 sec
    timeoutSeconds: 30
```

Commands

========

```
kubectl apply -f liveness-probe.yaml
kubectl exec -it mylivenessprobe -- /bin/bash

cat /tmp/healthy
echo $?
rm /tmp/healthy

kubectl describe pod mylivenessprobe
kubectl delete -f liveness-probe.yaml
```

Readiness Probes:

Readiness probe used to check if the application is ready to serve traffic. If a readiness probe fails, Kubernetes temporarily removes the pod from the service's endpoints until it becomes ready again. This ensures that only healthy and ready pods receive traffic.

Readiness Probe Types

There are three ways to define a readiness probe:

1. **HTTP GET Probe**: Sends an HTTP GET request to a specified path and port in the container. If the response is a 2xx or 3xx status code, the container is considered ready.

```
readinessProbe:
  httpGet:
    path: /ready
    port: 8080
  initialDelaySeconds: 5   # Wait 5 seconds before performing the first check
  periodSeconds: 10        # Perform the check every 10 seconds
  timeoutSeconds: 1        # Timeout after 1 second if no response
  successThreshold: 1      # A single success is enough to mark the container as ready
  failureThreshold: 3      # After 3 failures, the pod will be marked as not ready
```

2. **TCP Socket Probe**: Attempts to open a TCP connection on the specified port. If the connection is successful, the container is considered ready.

```
readinessProbe:
  tcpSocket:
    port: 3306
  initialDelaySeconds: 10   # Wait 10 seconds before the first check
  periodSeconds: 5          # Check every 5 seconds
  timeoutSeconds: 1         # Timeout if the connection takes longer than 1 second
  successThreshold: 1
  failureThreshold: 3
```

3. **Exec Probe**: Runs a command inside the container. If the command returns exit code 0, the container is considered ready.

```
readinessProbe:
  exec:
    command: ["cat", "/tmp/ready"]
  initialDelaySeconds: 5    # Wait 5 seconds before performing the first check
  periodSeconds: 10         # Check every 10 seconds
  timeoutSeconds: 1         # Command must finish within 1 second
  successThreshold: 1
  failureThreshold: 3
```

```
---
 kind: Deployment
 apiVersion: apps/v1
 metadata:
  name: mydeployments
 spec:
  replicas: 1
  selector:     # tells the controller which pods to watch/belong to
   matchLabels:
    name: deployment
  template:
   metadata:
    name: testpod8
    labels:
     name: deployment
   spec:
    containers:
     - name: hc
       image: httpd
       ports:
        - containerPort: 80
       args:
        - /bin/sh
        - -c
        - touch /tmp/healthy; sleep 1000
       readinessProbe:
        exec:
         command:
          - cat
          - /tmp/ready
        initialDelaySeconds: 15     # Wait 15 seconds before performing the first check
        periodSeconds: 10        # Check every 10 seconds
        timeoutSeconds: 30         # Command must finish within 30 seconds
        successThreshold: 1    # A single successful check marks the container as ready
        failureThreshold: 3    # 3 consecutive failures will mark the pod as not ready
clusteripservice.yaml
====================

---
 kind: Service              # Defines to create Service type Object
 apiVersion: v1
```

```
metadata:
  name: democipservice
spec:
  ports:
    - port: 80              # Containers port exposed
      targetPort: 80        # Pods port
  selector:
    name: deployment        # Apply this service to any pods which has the specific label
  type: ClusterIP           # Specifies the service type i.e ClusterIP or NodePort
```

Startup Probe

A **Startup Probe** is a type of probe used in Kubernetes to check whether an application inside a container has started successfully. It's specifically designed for containers that may take a long time to start up, such as those running legacy applications or databases, and avoids prematurely restarting them while they are still initializing.

Startup probes support the same types of checks as liveness and readiness probes:

- **HTTP GET Probe**: Makes an HTTP GET request to a specified endpoint.

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
  containers:
  - name: example-container
    image: myapp:latest
    startupProbe:
      httpGet:
        path: /startup
        port: 8080
      initialDelaySeconds: 10    # Time to wait before starting the first probe
      periodSeconds: 5          # Time between probe attempts
      failureThreshold: 30      # After 30 failures, the pod will be killed and restarted
```

- **TCP Socket Probe**: Tries to open a TCP connection.

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
```

```
containers:
- name: example-container
  image: myapp:latest
  startupProbe:
    tcpSocket:
      port: 3306            # Check if the container is accepting connections on port 3306
    initialDelaySeconds: 15     # Wait 15 seconds before starting the probe
    periodSeconds: 5          # Try every 5 seconds
    failureThreshold: 20       # After 20 failed attempts, restart the pod
```

- **Exec Probe**: Runs a command inside the container.

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
  containers:
  - name: example-container
    image: myapp:latest
    startupProbe:
      exec:
        command:
          - cat
          - /app/startup-complete   # Check if a specific file exists to signal readiness
      initialDelaySeconds: 10       # Wait 10 seconds before starting the first probe
      periodSeconds: 10             # Run the probe every 10 seconds
      failureThreshold: 60          # After 60 failed attempts, restart the pod
```


How Startup Probes Work with Liveness and Readiness Probes

- **While the startup probe is running**: Kubernetes does not run liveness or readiness probes.
- **Once the startup probe succeeds**: Kubernetes starts running the configured liveness and readiness probes.
- **If the startup probe fails**: Kubernetes restarts the container.

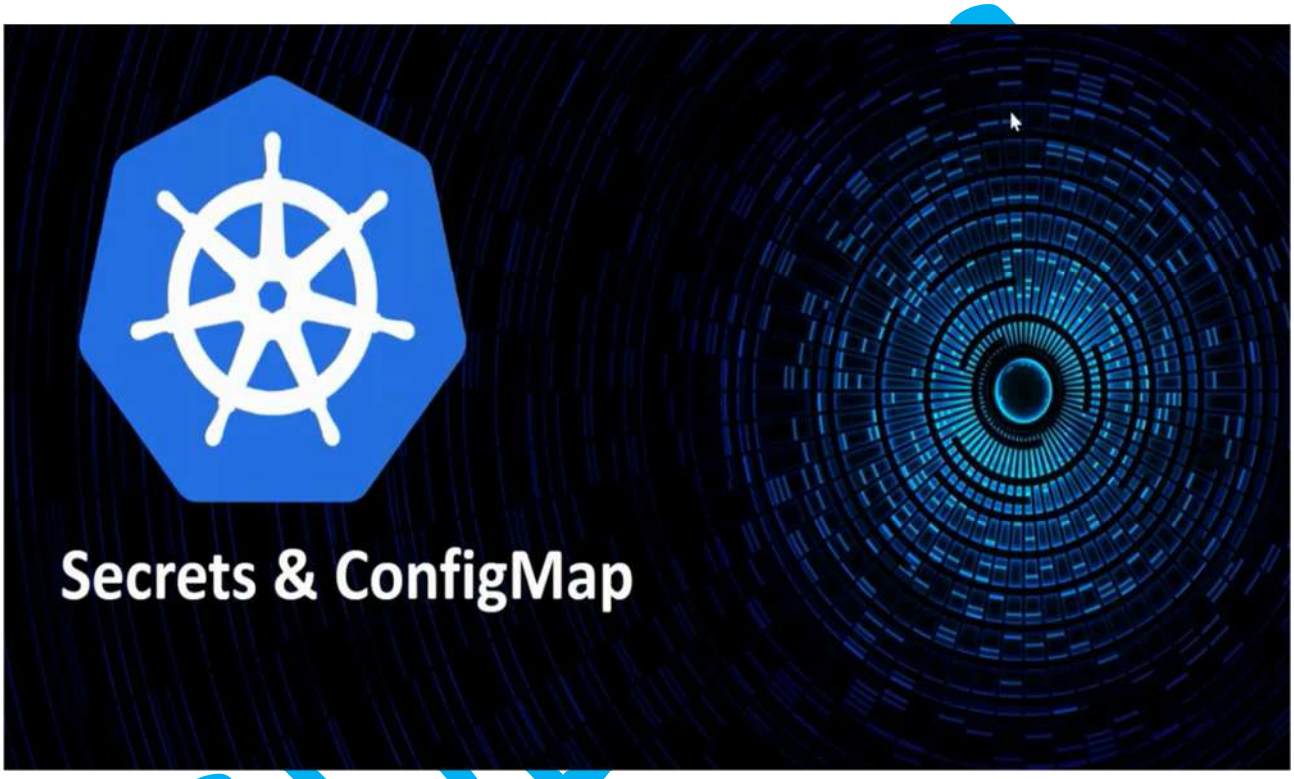
Demo

startup-probe.yaml

```
==================
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: startup
  name: mystartupprobe
spec:
  restartPolicy: Always
  containers:
  - name: startup
    image: ubuntu:latest
    args:
      - /bin/sh
      - -c
      - touch /tmp/healthy; sleep 1000
    startupProbe:                    # define the health check
      exec:
        command:                     # command to run periodically
          - cat
          - /tmp/ready
      initialDelaySeconds: 5     # Wait for the specified time before it runs the first probe
      periodSeconds: 5          # Run the above command every 5 sec
      timeoutSeconds: 30         # Command must finish within 30 seconds
```

Key Parameters

1. **initialDelaySeconds**: This defines how long Kubernetes will wait after the container has started before performing the first startup probe. This delay gives time for the application to initialize before checks are made.
2. **periodSeconds**: How often (in seconds) the startup probe will be performed. For example, if set to 5 seconds, Kubernetes will check the startup state every 5 seconds.
3. **timeoutSeconds**: The amount of time Kubernetes will wait for the startup probe to complete before it's considered a failure. If the probe doesn't respond within this timeout period, it fails.
4. **failureThreshold**: The number of consecutive probe failures that are allowed before the container is considered to have failed startup and is restarted. A higher value is typically used for applications that take a long time to start.
5. **successThreshold**: This is typically set to 1 for startup probes since the application only needs to pass the probe once to be considered started.

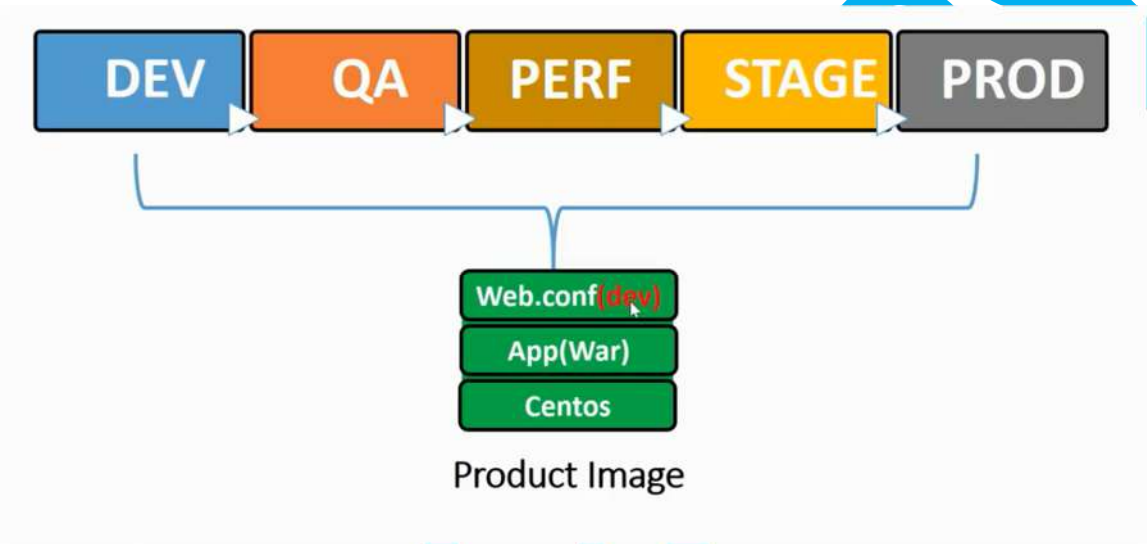


Prepared by PPNREDDY .... 😊 😊 😊

Configuration

Configuring Containerized Applications

- Container images are built to be portable.
- Containers expect configurations from.
  - Configuration files
  - Command line arguments
  - Environment variables
- Containers able to read the INI, JSON and XML formats will read.



ConfigMaps

ConfigMaps is a Kubernetes Object which allows you to separate configuration from PODS and components. It's become our application container become portable and makes the configuration easier to change and manage and prevent the hardcode configurations.

- A ConfigMaps is a dictionary of key-value pairs that store configuration settings for your application. We can mount this ConfigMaps in container as files or volumes or environment variables. Using ConfigMaps we store configuration files in a ConfigMaps and we can mount this configuration files into the container.
- ConfigMaps and secrets both are similar, both work in the similar way. Difference between secrets and ConfigMaps is, we use secrets for some sensitive data and we use ConfigMaps for non-sensitive like configuration files and environment variables.
- Stores configuration data as key-value pairs.
  - Configuration files
  - Command line arguments
  - Environment variables
- We must create a ConfigMaps before referencing it in a POD spec.

Advantages of ConfigMaps

1. Separation of Configuration and Code

ConfigMaps allow you to decouple configuration data from application images, enabling you to update the configuration without rebuilding the container image. This follows the 12-factor app principle of separating config from code.

2. Dynamic Configuration Changes

With *ConfigMaps,* you can update configuration values dynamically, allowing your application to react to changes without redeploying the application. If the application is designed to re-read the configuration (or if it watches for changes), the configuration can be updated on-the-fly.

3. Reuse Across Environments

You can easily reuse the same container image in different environments (development, testing, production) with different configurations. This makes it easier to promote applications across environments without altering the application itself.

4. Centralized Configuration Management

Instead of hardcoding configurations in your application or managing them in individual environment variables, *ConfigMaps* provide a centralized way to manage configuration data in one place, improving traceability and consistency.

5. Fine-Grained Control of Configuration Data

ConfigMaps support managing different types of configuration data, such as key-value pairs, entire configuration files, or command-line arguments. This gives developers flexibility in how they structure their configuration and allows for easier fine-tuning of specific configuration parts.

6. Environmental Adaptation

ConfigMaps help manage configurations specific to the cluster environment (e.g., cloud provider settings, external services, etc.), making it simpler to adapt applications when moving across clusters or cloud providers.

7. Version Control and Auditing

When stored in Git (infrastructure-as-code approach), *ConfigMaps* can be version-controlled alongside your application. This allows for easy rollback to a previous configuration in case of issues and helps in auditing configuration changes.

8. Simplified Secrets Management (with limitations)

While *ConfigMaps* are not intended for managing sensitive data (that's what *Secrets* are for), they can manage less-sensitive application configuration data separately from secrets, maintaining clarity and security.

9. Pod Lifecycle Flexibility

ConfigMaps can be injected into Pods at runtime via environment variables or mounted as volumes. This ensures that the Pod has access to the necessary configuration data when it starts, giving more flexibility in how and when you apply configuration.

10. Easier Scaling and Rollouts

When scaling applications horizontally, using *ConfigMaps* ensures that each replica of your application has consistent configuration without having to manually set environment variables or modify code for each instance.

11. Lightweight and Read-Only by Default

ConfigMaps are lightweight and consumed in a read-only manner by containers. This avoids any accidental modifications to configuration data by the applications running inside the containers.

12. Integration with Kubernetes Resources

ConfigMaps can be integrated with other Kubernetes resources, such as *Deployments, StatefulSets, DaemonSets,* etc., allowing you to pass configuration seamlessly into your workloads.

Use Cases for ConfigMaps:

- Managing configuration files.
- Injecting environment variables.
- Managing non-sensitive credentials or connection strings.
- Sharing configuration across multiple microservices or pods.

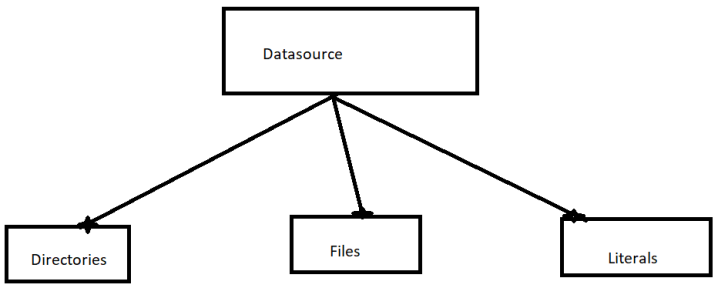
Syntax

kubectl create configMap <map-name> <data-source>

<data-source>→ Path to dir/file : --from-file

➔ Key-Value pair: --from-literal

DataSource



configmap.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: example-config
data:
  # Property-like keys
  config.properties: |
    setting1=psd
    setting2=devops
  # Single key-value pair
  log_level: "INFO"
  # JSON file
  data.json: |
    {
      "key1": "aws",
      "key2": "kubernetes"
    }
```

configmap-env.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: app-pod-env
spec:
  containers:
  - name: app-env-container
    image: nginx
    env:
    - name: LOG_LEVEL
      valueFrom:
        configMapKeyRef:
          name: example-config
          key: log_level
```

configmap-vol.yaml

```
apiVersion: v1
kind: Pod
```

```
metadata:
  name: app-pod-vol
spec:
  containers:
  - name: app-vol-container
    image: nginx
    volumeMounts:
    - name: config-volume
      mountPath: /etc/config
  volumes:
  - name: config-volume
    configMap:
      name: example-config
```

Commands:

```
kubectl apply -f configmap.yaml
kubectl apply -f configmap-env.yaml
kubectl apply -f configmap-vol.yaml

kubectl get configmaps
kubectl describe configmap example-config

kubectl exec -it  app-pod-env -- /bin/bash
echo $LOG_LEVEL

kubectl exec -it  app-pod-vol -- /bin/bash
cd /etc/config/

cat config.properties
cat data.json
cat log_level

kubectl delete -f configmap.yaml
kubectl delete -f configmap-env.yaml
kubectl delete -f configmap-vol.yaml
```

Secrets

Secrets is a Kubernetes object to handle small amount of sensitive data which includes username, password, tokens and ssh keys. We must protect this data and at the same time we have to use this data to run applications. Using Kubernetes secrets we can encrypt the sensitive data. The secrets will end up as environment variables within the pod.

- Secrets Reduces risk of exposing sensitive data.
- Secrets Created outside of Pods.
- Secrets Stored inside ETCD database on Kubernetes Master.
- Secrets size is not more than 1MB.
- Secrets used in two ways- Volumes or Env variables.
- Secrets Sent only to the target nodes.

Advantages of Secrets

1. Secure Management of Sensitive Data

Secrets are designed to store and manage sensitive data securely. By using *Secrets,* sensitive information (like database credentials or encryption keys) is not hardcoded into container images or configuration files, reducing the risk of accidental exposure.

2. Encrypted Storage

By default, Kubernetes stores *Secrets* in etcd, which can be encrypted at rest. This ensures that even if someone gains access to etcd, the secrets will be encrypted, providing an additional layer of security. Kubernetes can be configured to encrypt *Secrets* at rest, making it much safer than storing sensitive data in plaintext.

3. Control Access with RBAC

Secrets can be accessed using Kubernetes' Role-Based Access Control (RBAC). This allows for fine-grained control over who can create, update, or view *Secrets* within the cluster, ensuring that only authorized users and applications have access to sensitive data.

4. Secure Injection into Pods

Secrets can be securely injected into pods as environment variables or mounted as volumes. They are only available to the specific containers that need them, reducing the surface area of exposure within the cluster. Kubernetes ensures the secret is only accessible by the pods that need it.

5. Minimize Exposure with Automatic Cleanup

When a pod that uses a secret is terminated, Kubernetes ensures that the associated secret data is cleaned up, preventing sensitive information from being exposed after the pod is no longer running.

6. Separation of Concerns

By decoupling secrets from the application code and configuration, *Secrets* follow the principle of least privilege, where sensitive information is managed separately and only shared with applications that need it. This simplifies managing access to sensitive data across different applications and environments.

7. Easier Rotation of Secrets

Kubernetes *Secrets* make it easier to rotate sensitive data (like rotating passwords or tokens) without disrupting the running application. By simply updating the secret, the changes can be automatically reflected in the pods that use it (depending on how the application handles updates).

8. Protection in Transit

Kubernetes *Secrets* are transmitted securely between the API server and nodes using Transport Layer Security (TLS), ensuring that sensitive data is protected from interception while in transit.

9. Support for Different Data Formats

Kubernetes *Secrets* support various types of sensitive data, from simple key-value pairs (like passwords) to binary data (such as SSL certificates). This flexibility makes them useful for a wide range of use cases.

10. Memory-Only Storage Option in Pods

When *Secrets* are mounted as volumes in a pod, Kubernetes can store them in a temporary memory filesystem (tmpfs). This ensures that sensitive information is never written to disk, preventing potential leakage through persistent storage or logs.

11. Integration with External Secret Management Systems

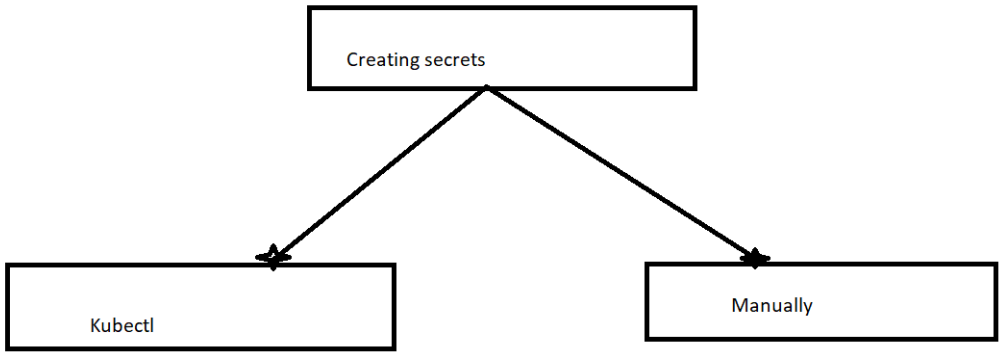
Kubernetes *Secrets* can integrate with external secret management systems, such as HashiCorp Vault or AWS Secrets Manager, to pull and manage sensitive data securely. This allows Kubernetes to work seamlessly with centralized secret management systems while maintaining Kubernetes-native security.

12. Reducing Risks of Configuration Errors

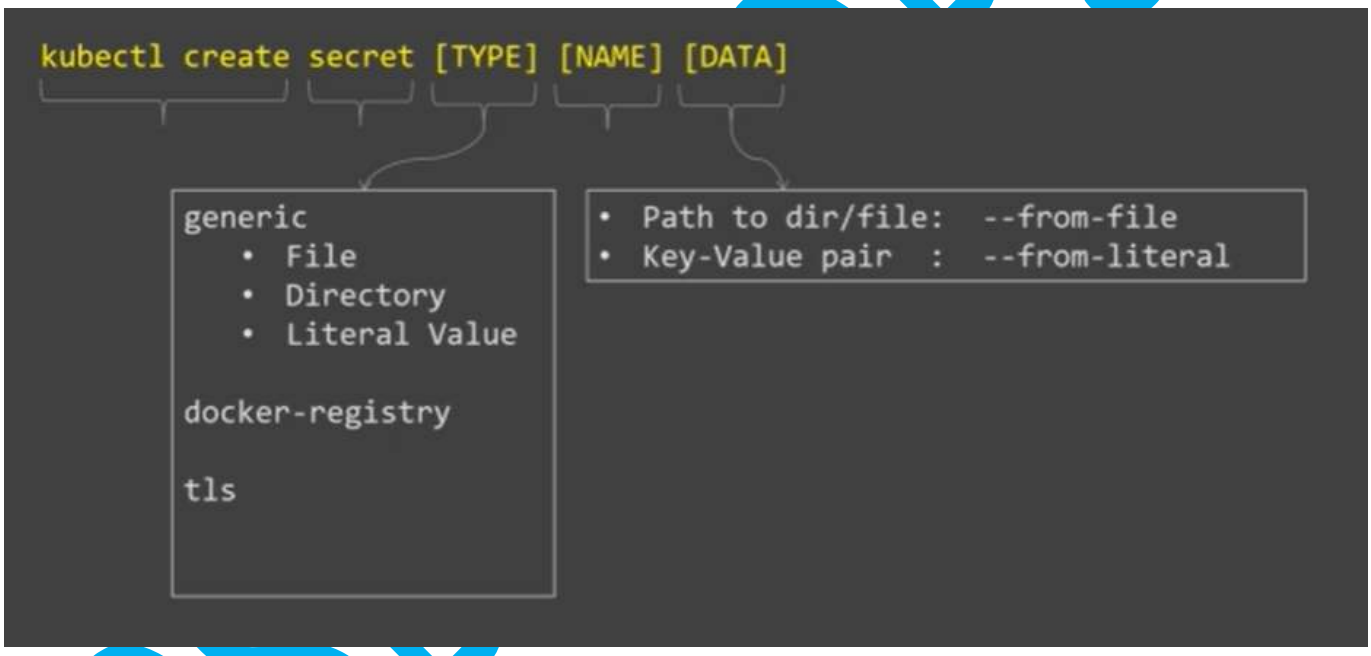
By keeping sensitive data in *Secrets*, there's a lower risk of accidentally leaking sensitive information into logs or configuration files, reducing the chances of human error in handling confidential data.

13. Minimal Resource Overhead

Kubernetes *Secrets* are lightweight and efficiently handled by the control plane and nodes. The overhead of using *Secrets* to manage sensitive data securely is minimal compared to alternatives like manually injecting sensitive information into pods or hardcoding it.



Using kubectl syntax



Creating Secret: Manually

```
srinath@master:$ echo -n 'admin' | base64
YWRtaW4=
srinath@master:$ echo -n '1f2d1e2e67df' | base64
MWYyZDFlMmU2N2Rm
```

```
# mysecret.yaml
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRtaW4=
  password: MWYyZDFlMmU2N2Rm
```

```
srinath@master:$ kubectl create -f mysecret.yaml
secret/mysecret created
```

Decoding Secrets

```
srinath@master:$ kubectl get secrets mysecret -o yaml
apiVersion: v1
data:
  password: MWYyZDFlMmU2N2Rm
  username: YWRtaW4=
kind: Secret
metadata:
  creationTimestamp: 2018-09-01T12:46:17Z
  name: mysecret
  namespace: default
  resourceVersion: "616565"
  selfLink:
/api/v1/namespaces/default/secrets/mysecret
  uid: 051e61ae-ade5-11e8-8d64-42010a800003
type: Opaque
```

```
srinath@master:$ echo 'YWRtaW4=' | base64 --decode
admin
srinath@master:$ echo 'MWYyZDFlMmU2N2Rm' | base64 --decode
1f2d1e2e67df
```

Consuming “Secrets” from volume

```
# mysecret-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
      readOnly: true
  volumes:
  - name: foo
    secret:
      secretName: mysecret
```

```
srinath@master:$ kubectl create -f mysecret-pod.yaml
secret/mysecret-pod created

srinath@master:$ kubectl get po
NAME      READY     STATUS              RESTARTS   AGE
mypod     1/1       Running             0          22m

srinath@master:$ kubectl exec mypod ls /etc/foo
password
username

srinath@master:$ kubectl exec mypod cat /etc/foo/passwd
1f2d1e2e67df

srinath@master:$ kubectl exec mypod cat /etc/foo/username
admin
```

Consuming “Secrets” from “Environment Variables”

```
# mysecret-env-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: secret-env-pod
spec:
  containers:
  - name: mycontainer
    image: redis
    env:
      - name: SECRET_USERNAME
        valueFrom:
          secretKeyRef:
            name: mysecret
            key: username

      - name: SECRET_PASSWORD
        valueFrom:
          secretKeyRef:
            name: mysecret
            key: password
  restartPolicy: Never
```

```
srinath@master:$ kubectl create -f mysecret-pod-env.yaml
secret/mysecret-pod-env created

srinath@master:$ kubectl get po
NAME             READY     STATUS    RESTARTS   AGE
secret-env-pod   1/1       Running   0          7s

srinath@master:$ kubectl exec secret-env-pod env | grep SECRET
SECRET_PASSWORD=1f2d1e2e67df
SECRET_USERNAME=admin
```

Create a Secret

kubectl create secret generic my-secret \
  --from-literal=username=ppreddy \
  --from-literal=password=India@123

kubectl get secrets my-secret -o yaml
echo -n 'ppreddy' | base64
echo -n 'India@123' | base64

ubuntu@ip-10-1-1-187:~$ echo -n 'ppreddy' | base64
cHByZWRkeQ==

```
ubuntu@ip-10-1-1-187:~$ echo -n 'India@123' | base64
SW5kaWFAMTIz
ubuntu@ip-10-1-1-187:~$
```

secret.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
type: Opaque
data:
  username: cHByZWRkeQ==   # 'myuser' base64-encoded
  password: SW5kaWFAMTIz  # 'mypassword' base64-encoded
```

secrets-env.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-env-pod
spec:
  containers:
    - name: my-container
      image: nginx
      env:
        - name: USERNAME
          valueFrom:
            secretKeyRef:
              name: my-secret
              key: username
        - name: PASSWORD
          valueFrom:
            secretKeyRef:
              name: my-secret
              key: password
```

secrets-vol.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-vol-pod
spec:
  containers:
    - name: my-container
      image: nginx
      volumeMounts:
        - name: secret-volume
          mountPath: /etc/secret
          readOnly: true
  volumes:
    - name: secret-volume
      secret:
        secretName: my-secret
```

commands:

```
kubectl apply -f secret.yaml
kubectl apply -f secrets-env.yaml
kubectl apply -f secrets-vol.yaml

kubectl get secrets
kubectl describe secret my-secret

kubectl get pods
kubectl exec -it secret-env-pod -- /bin/bash
echo $USERNAME
echo $PASSWORD

kubectl exec -it secret-vol-pod -- /bin/bash
cd /etc/secret
cat username
cat password

kubectl delete -f secret.yaml
kubectl delete -f secrets-env.yaml
kubectl delete -f secrets-vol.yaml
```

Namespace

In Kubernetes, **Namespaces** provide a way to organize and isolate resources within a cluster. They are essentially virtual clusters backed by the same physical cluster. Namespaces are helpful when multiple teams or applications are using the same Kubernetes cluster, or when you want to logically separate environments like development, staging, and production.

Key Features of Kubernetes Namespaces:

1. **Resource Isolation**: Each namespace provides its own environment, isolating resources like Pods, Services, and ConfigMaps. Resources within one namespace cannot directly interact with those in another unless explicitly configured.
2. **Resource Quotas**: You can set resource quotas (like CPU, memory) on namespaces to control resource consumption by workloads within that namespace.
3. **Access Control**: You can configure different levels of access for different namespaces using Kubernetes' Role-Based Access Control (RBAC).
4. **Default Namespace**: If a resource does not specify a namespace, it is created in the default namespace.

Use Cases for Namespaces

1. **Environment Separation**: Create namespaces for different environments like dev, stage, and prod to isolate development and production workloads.
2. **Multi-Tenant Clusters**: In a shared cluster, namespaces allow multiple teams to operate independently.
3. **Resource Quotas**: Enforce resource constraints within each namespace to prevent one team or application from exhausting cluster resources.

Listing Namespaces

To view the existing namespaces in a cluster:

kubectl get namespaces

```
ubuntu@ip-10-1-1-187:~$ kubectl get namespaces
NAME              STATUS   AGE
default           Active   36d
kube-node-lease   Active   36d
kube-public       Active   36d
kube-system       Active   36d
```

default: The default namespace for resources with no specified namespace.
kube-system: Contains resources for the Kubernetes control plane.
kube-public: A public namespace where resources are readable by all users (including those without authentication).
kube-node-lease: Used for node heartbeat data.

Creating namespace:

kubectl create namespace dev

kubectl get namespaces

```
apiVersion: v1
kind: Namespace
metadata:
  name: qa
```

kubectl apply -f create-ns.yaml

Deleting namespace

kubectl delete namespace dev

kubectl get namespaces

kubectl delete -f create-ns.yaml

Switch to the namespace

kubectl create namespace dev

kubectl create namespace qa

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

kubectl config view --minify --output 'jsonpath={..namespace}'

Create a pod in dev namespace

kubectl apply -f basic-pod.yaml -n dev

kubectl get pods -n dev

kubectl delete -f basic-pod.yaml -n dev

Resource Quotas and Limit Ranges in Namespaces

Namespaces are also useful for controlling resource usage within a Kubernetes cluster. You can set **Resource Quotas** to limit the total amount of CPU, memory, or other resources a namespace can consume.

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-quota
  namespace: my-namespace
spec:
  hard:
    pods: "10"
    requests.cpu: "4"
    requests.memory: 8Gi
    limits.cpu: "8"
    limits.memory: 16Gi
```

This ensures that the namespace my-namespace can only run 10 Pods, and the CPU and memory consumption are capped at specific limits.

LimitRange

You can also define default resource limits for Pods or Containers in a namespace using a

```
apiVersion: v1
kind: LimitRange
metadata:
  name: limits
  namespace: my-namespace
spec:
  limits:
    - default:
        cpu: "500m"
        memory: "512Mi"
      defaultRequest:
        cpu: "250m"
        memory: "256Mi"
      type: Container
```

Managing Compute Resources for Containers

- A pod in Kubernetes will run with no limits on CPU and memory
- You can optionally specify how much CPU and memory (RAM) each Container needs.
- Scheduler decides about which nodes to place Pods, only if the Node has enough CPU resources available to satisfy the Pod CPU request.
- CPU is specified in units of cores, and memory is specified in units of bytes.
- Two types of constraints can be set for each resource type: requests and limits
  1. A request is the amount of that resources that the system will guarantee for the container, and Kubernetes will use this value to decide on which node to place the pod.
  2. A limit is the maximum number of resources that Kubernetes will allow the container to use. In the case that request is not set for a container, it defaults to limit. If limit is not set, then if defaults to 0 (unbounded).
- CPU values are specified in "millicpu" and memory in MiB

resources.yaml

```
==============
apiVersion: v1
kind: Pod
metadata:
  name: resources
spec:
  containers:
  - name: resource
    image: centos
    command: ["/bin/bash", "-c", "while true; do echo Hello-Kubernetes; sleep 5 ; done"]
    resources:                # Describes the type of resources to be used
      requests:
        memory: "64Mi"
        cpu: "100m"
      limits:
        memory: "128Mi"
        cpu: "200m"
```

Commands:

```
=========
kubectl apply -f resources.yaml
kubectl get pods
kubectl describe pod resources
kubectl delete -f resources.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: resources
spec:
  containers:
  - name: resource
    image: centos
    command: ["/bin/bash", "-c", "while true; do echo Hello-Kubernetes; sleep 5 ; done"]
    resources:                  # Describes the type of resources to be used
      requests:
        memory: "6400Mi"
        cpu: "10000m"
      limits:
        memory: "12800Mi"
        cpu: "20000m"

Commands:
=========
kubectl apply -f resources.yaml
kubectl get pods
kubectl describe pod resources
kubectl delete -f resources.yaml
```

Resource Quotas

In Kubernetes, **Resource Quotas** are a way to manage and limit the resource usage (such as CPU, memory, or the number of objects) within a namespace. This ensures fair distribution of resources in a multi-tenant environment, preventing any single namespace or workload from consuming more than its allocated share of the cluster's resources.

Key Features of Resource Quotas

1. **Limit Resource Consumption**: You can restrict the total amount of compute resources (like CPU and memory) a namespace can use.
2. **Limit Object Counts**: You can limit the number of Kubernetes objects (like Pods, Services, ConfigMaps, etc.) that can be created in a namespace.
3. **Ensures Fairness**: Quotas prevent one namespace or team from consuming an unfair share of resources in a multi-tenant cluster.
4. **Enforced at Namespace Level**: Resource quotas apply to all resources within a namespace, and each namespace can have its own quota.

Types of Quotas

- **Compute Resource Quotas**: These quotas limit CPU and memory usage.
- **Storage Quotas**: These control the number of persistent volumes or the amount of storage that can be consumed.
- **Object Count Quotas**: These limit the number of objects (like Pods, Services, or ConfigMaps) that can be created.

Setting a Resource Quota

Here's a YAML configuration for creating a **ResourceQuota** in a namespace.

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-quota
  namespace: my-namespace
spec:
  hard:
    pods: "10"                  # Limit to 10 Pods in the namespace
    requests.cpu: "4"           # Total requested CPU cannot exceed 4 cores
    requests.memory: "8Gi"      # Total requested memory cannot exceed 8GiB
    limits.cpu: "8"             # Total CPU limit across all Pods is 8 cores
    limits.memory: "16Gi"       # Total memory limit across all Pods is 16GiB
```

pods: Limits the number of Pods to 10.
requests.cpu: Limits the total CPU requested by all Pods in the namespace to 4 cores.
requests.memory: Limits the total memory requested to 8GiB.
limits.cpu: Limits the total CPU limit across all Pods to 8 cores.
limits.memory: Limits the total memory limit across all Pods to 16GiB.

Limiting Object Counts

In addition to compute resources, you can limit the number of Kubernetes objects (e.g., Pods, Services) in a namespace:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: object-counts
  namespace: my-namespace
spec:
  hard:
    configmaps: "20"       # Limit to 20 ConfigMaps in the namespace
    persistentvolumeclaims: "5"   # Limit to 5 PersistentVolumeClaims
    secrets: "10"          # Limit to 10 Secrets
    services: "5"          # Limit to 5 Services
    replicationcontrollers: "10" # Limit to 10 Replication Controllers
```

Behaviour of Resource Quotas

1. **Pod Creation**: If a resource request or limit for a Pod exceeds the quota, the Pod will not be scheduled. For example, if a namespace is already using 4GiB of memory and the quota is 8GiB, a Pod requesting 5GiB of memory will be rejected.
2. **Object Creation**: If the quota for objects (like ConfigMaps or Services) is exceeded, no more objects of that type can be created in the namespace until the number drops below the quota.
3. **Requests and Limits**: If a container does not specify resource requests and limits, Kubernetes may refuse to create Pods if the sum of default values exceeds the quota.

Setting Quotas for Persistent Volumes

You can also set quotas for Persistent Volume Claims (PVCs), ensuring that only a limited amount of storage is used in a namespace.

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: storage-quota
  namespace: my-namespace
```

```
spec:
  hard:
    persistentvolumeclaims: "10"   # Limit to 10 PersistentVolumeClaims
    requests.storage: "500Gi"      # Limit total requested storage to 500GiB
```

Best Practices for Resource Quotas

1. **Define Quotas Early**: Set up resource quotas during the early stages of cluster setup, especially in shared environments, to prevent resource contention later.
2. **Use Quotas with LimitRanges**: Combine **Resource Quotas** with **LimitRanges** to enforce default limits on resource usage for Pods and Containers.
3. **Monitor Quota Usage**: Regularly monitor resource usage in namespaces to ensure that the quotas are being respected and adjusted as needed.
4. **Separate Environments**: Use namespaces along with quotas to separate and manage environments (e.g., dev, test, prod) in multi-tenant clusters.

Creating resource quota's default

resourcequota.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: myquota
spec:
  hard:
    limits.cpu: "400m"
    limits.memory: "400Mi"
    requests.cpu: "200m"
    requests.memory: "200Mi"
```

commands:

```
kubectl apply -f resourcequota.yaml
kubectl get resourcequota
kubectl describe resourcequota myquota
```

rqpod.yaml

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: deployments
spec:
```

```
  replicas: 3
  selector:
    matchLabels:
     objtype: deployment
  template:
    metadata:
      name: testpod8
      labels:
        objtype: deployment
    spec:
     containers:
      - name: c00
        image: ubuntu
        command: ["/bin/bash", "-c", "while true; do echo Hello-kubernetes; sleep 5 ; done"]
        resources:
          requests:
            cpu: "200m"
            memory: "128Mi"
          limits:
            cpu: "400m"
            memory: "256Mi"
```

commands

```
kubectl apply -f rqpod.yaml
kubectl get deployments
kubectl get rs
kubectl get pods
kubectl delete -f rqpod.yaml
```

Change the replicas to 1 and test (replicas: 3) above manifest file

```
kubectl apply -f rqpod.yaml
kubectl get deployments
kubectl get rs
kubectl get pods
kubectl delete -f rqpod.yaml
delete the resoucequota
kubectl delete -f resourcequota.yaml
```

LimitRange

A **LimitRange** in Kubernetes is a resource type that helps enforce minimum and maximum resource limits on containers within a namespace. It ensures that resource constraints such as CPU and memory are properly allocated and managed, preventing resource overconsumption by individual pods or containers.

Here's how it works:

1. **Minimum Resource Limits**: Set minimum CPU/memory that each container must request.
2. **Maximum Resource Limits**: Set maximum CPU/memory that a container can use.
3. **Default Requests and Limits**: Specify default resource requests/limits if none are specified by the user.
4. **Resource Usage Ratio**: Ensure that the resources requested are within a reasonable ratio of what is limited.

Example:

```
apiVersion: v1
kind: LimitRange
metadata:
  name: limits
  namespace: my-namespace
spec:
  limits:
  - max:
      cpu: "2"
      memory: "1Gi"
    min:
      cpu: "100m"
      memory: "128Mi"
    default:
      cpu: "500m"
      memory: "256Mi"
    defaultRequest:
      cpu: "200m"
      memory: "256Mi"
    type: Container
```


Key Components:

- **max**: Specifies the maximum resources a container can use.
- **min**: Specifies the minimum resources a container must request.
- **default**: Sets default resource limits if none are provided by the container.
- **defaultRequest**: Sets default resource requests if none are provided by the container.
- **type**: Can be Container (for individual containers) or Pod (for the entire pod).

cpulr.yaml
==========

```
apiVersion: v1
kind: LimitRange
metadata:
  name: cpu-limit-range
spec:
  limits:
  - default:
      cpu: 0.5
    defaultRequest:
      cpu: 0.2
    type: Container
```

commands:
=========

```
kubectl apply -f cpulr.yaml
kubectl get limitrange
kubectl describe limitrange cpu-limit-range
```

cpu-limit-pod.yaml
==================

```
apiVersion: v1
kind: Pod
metadata:
  name: default-cpu-demo-2
spec:
  containers:
  - name: default-cpu-demo-2-ctr
    image: nginx
    resources:
      limits:
        cpu: "0.5"
```

commands:
=========

```
kubectl apply -f cpu-limit-pod.yaml
kubectl describe pod default-cpu-demo-2
```

cpu-request-pod.yaml
====================

```
apiVersion: v1
kind: Pod
metadata:
  name: default-cpu-demo-3
spec:
  containers:
  - name: default-cpu-demo-3-ctr
    image: nginx
    resources:
      requests:
        cpu: "0.3"
```

commands:
=========

```
kubectl apply -f cpu-request-pod.yaml
kubectl describe pod default-cpu-demo-3
kubectl delete -f cpu-request-pod.yaml
```

memorylr.yaml
=============

```
apiVersion: v1
kind: LimitRange
metadata:
  name: mem-min-max-demo-lr
spec:
  limits:
  - max:
      memory: 500Mi
    min:
      memory: 250Mi
    type: Container
```

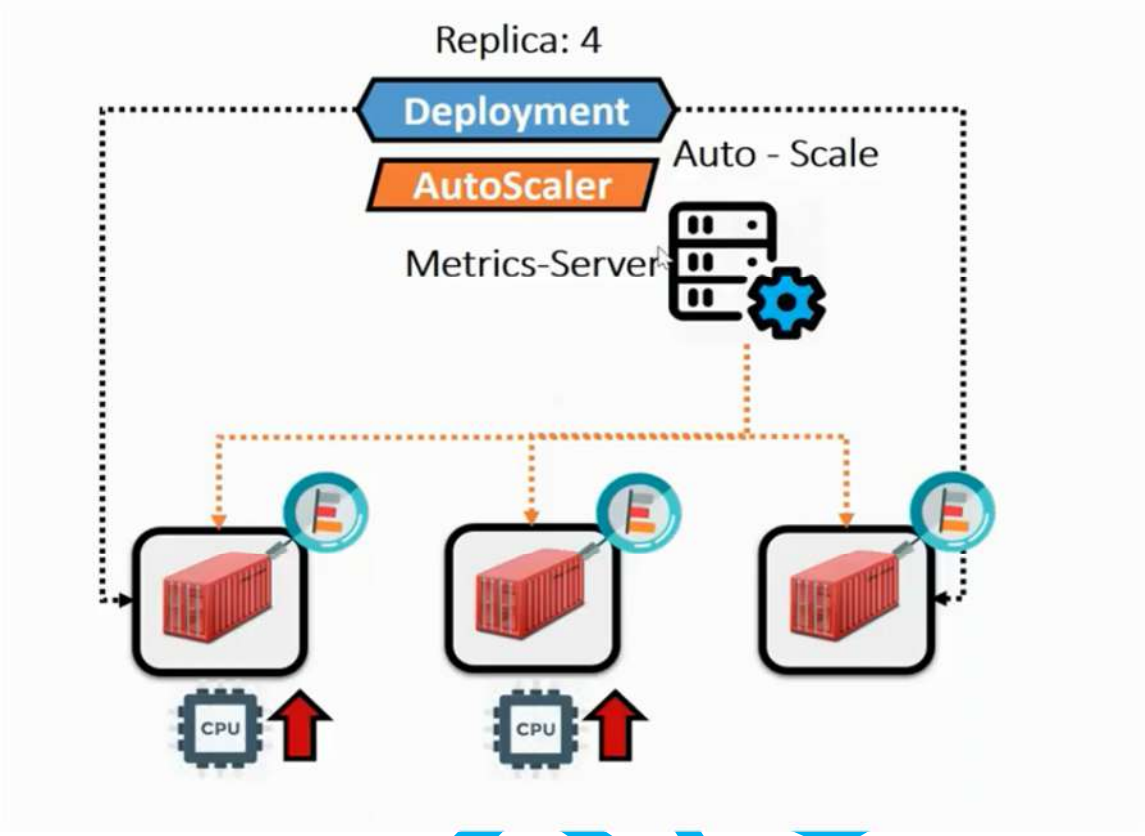
commands:
=========

```
kubectl apply -f memorylr.yaml
kubectl get limitrange
kubectl describe limitrange mem-min-max-demo-lr
```

memorypod.yaml

```
==============
apiVersion: v1
kind: Pod
metadata:
  name: constraints-mem-demo
spec:
  containers:
  - name: constraints-mem-demo-ctr
    image: nginx
    resources:
      limits:
        memory: "400Mi"
        #cpu: "0.4"
      requests:
        memory: "300Mi"
        #cpu: "0.3"
commands:
=========
kubectl apply -f memorypod.yaml
kubectl get pods
kubectl describe pod constraints-mem-demo
kubectl delete -f memorypod.yaml
```

Prepared by PPNREDDY .... 😊 😊 😊



Kubernetes Horizontal Pod Autoscaler **(HPA)** automatically scales the number of pods in a replication controller, deployment, or replica set based on observed CPU utilization, memory usage, or custom metrics. HPA adjusts the number of pod replicas to match the current demand on the application, ensuring optimal resource utilization while maintaining performance.

Key Features of HPA:

1. **Automatic Scaling:** HPA adjusts the number of pods based on metrics like CPU, memory, or custom metrics (e.g., request rate, latency).
2. **Custom Metrics:** Besides CPU and memory, HPA supports custom metrics (like queue length, request per second) via the Kubernetes Metrics API or third-party metrics providers like Prometheus.
3. **Thresholds:** HPA allows you to specify a target threshold for the average metric (e.g., target 80% CPU utilization), and it will increase or decrease the number of pods to maintain that threshold.

How HPA Works:

HPA periodically queries the Kubernetes Metrics API to get the current resource usage of the pods it is monitoring. It compares the current resource utilization against the target and then adjusts the number of replicas accordingly.

For example, if CPU usage is higher than the target threshold (e.g., 70%), HPA scales up the pods. If the CPU usage is lower, HPA scales down the pods.

Components Involved:

1. **Metrics Server:** Required to provide CPU and memory metrics to the HPA. It aggregates metrics from the Kubelet and exposes them through the Kubernetes Metrics API.
2. **Controller Manager:** The HPA controller runs as part of the Kubernetes controller manager, which periodically checks the metrics and adjusts the desired replica count.

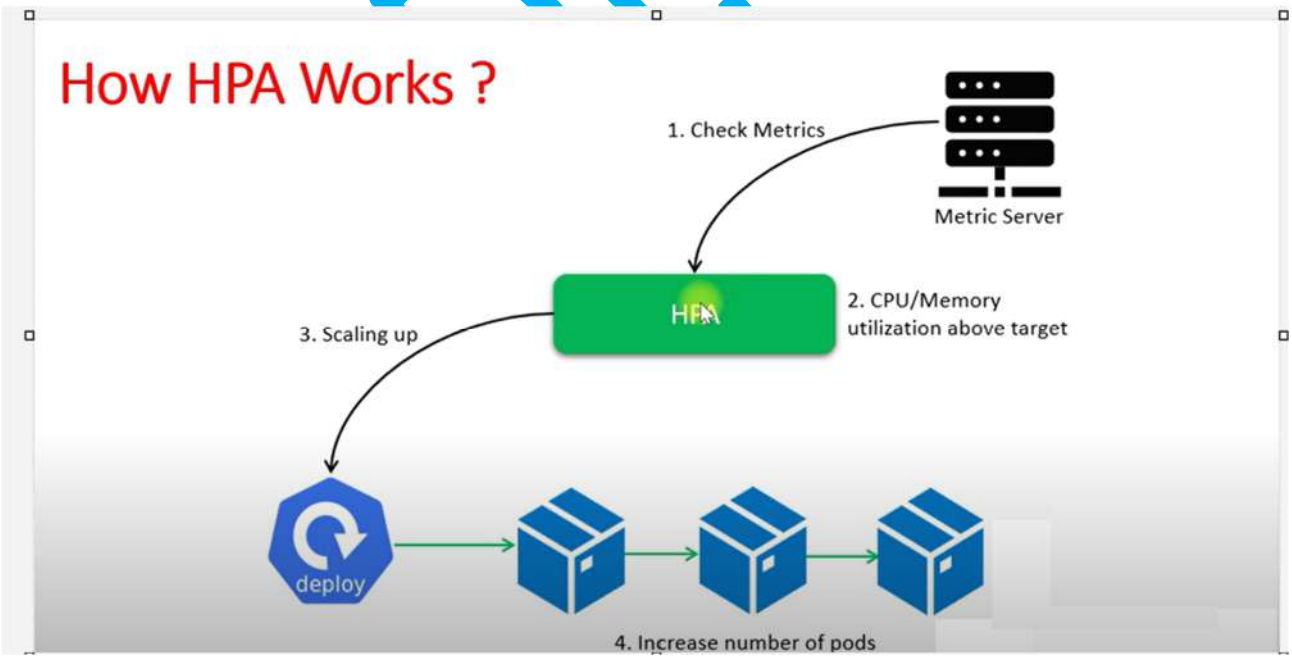
Key Configuration Parameters:

- **Min Replicas (minReplicas):** The minimum number of pods that should always be running.
- **Max Replicas (maxReplicas):** The maximum number of pods that the HPA can scale to.
- **Metrics (metrics):** Defines which metrics to monitor (e.g., CPU, memory, custom metrics).

Install metric server

git clone https://github.com/psddevops/jenkins_pipelines.git
cd ./jenkins_pipelines/metricserver/
kubectl apply -f .
kubectl top nodes

```
ubuntu@ip-10-1-1-187:~$ kubectl top nodes
NAME            CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
ip-10-1-1-18    36m          3%     466Mi           54%
ip-10-1-1-187   111m         11%    695Mi           81%
ip-10-1-1-188   88m          8%     479Mi           55%
ubuntu@ip-10-1-1-187:~$ cd metricserver/
```



Install metricserver

git clone https://github.com/psddevops/jenkins_pipelines.git

cd jenkins_pipelines

cd metricserver

kubectl apply -f .

kubectl top nodes

hpa-deployment.yaml
==========

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

hpa-service.yaml

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

hpa.yaml

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

Increasing load

```
kubectl run -i --tty load-generator --rm --image=busybox --restart=Never -- /bin/sh -c "while sleep 0.02; do wget -q -O- http://hpa-demo-deployment; done"
```

Commands:

=========

kubectl apply -f hpa-deployment.yaml

kubectl apply -f hpa-service.yaml

kubectl apply -f hpa.yaml

kubectl get deployments

kubectl get rs

kubectl get hpa

kubectl get pods

kubectl describe hpa hpa-demo-deployment

Let's increase the load

kubectl run -i --tty load-generator --rm --image=busybox --restart=Never -- /bin/sh -c "while sleep 0.02; do wget -q -O- http://hpa-demo-deployment; done"

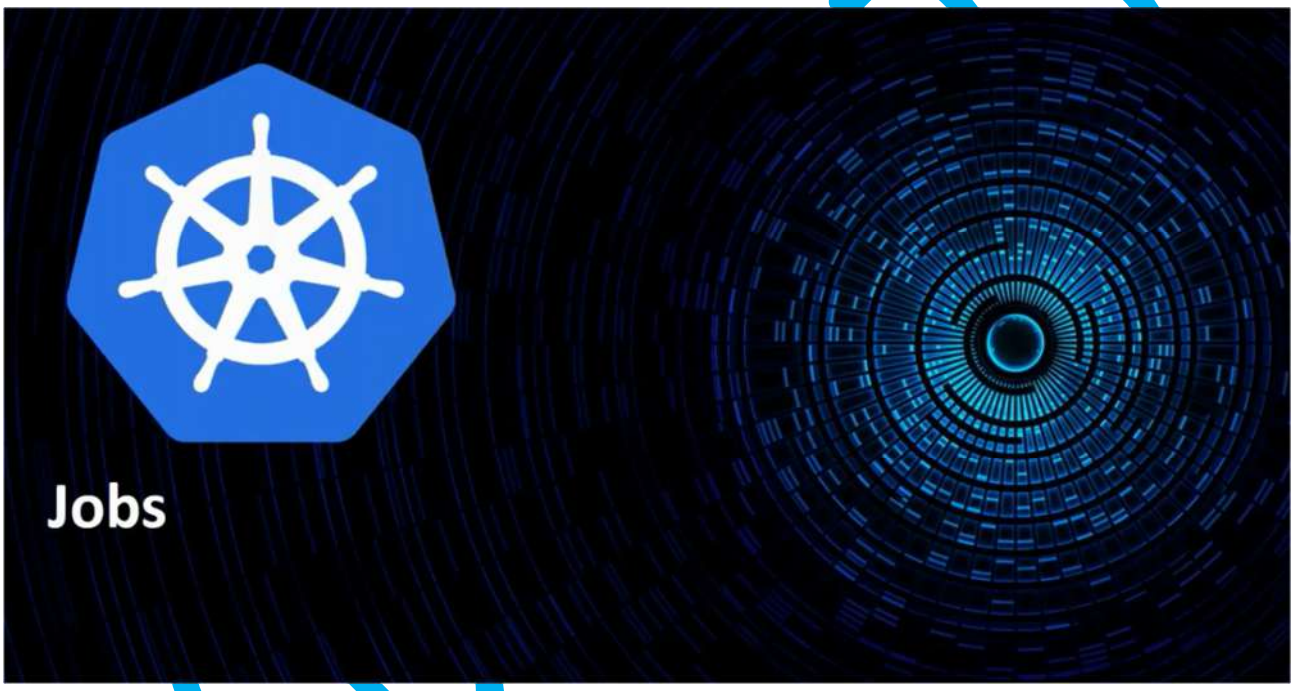
kubectl delete -f hpa-deployment.yaml

kubectl delete -f hpa-service.yaml

kubectl delete -f hpa.yaml

```
ubuntu@ip-10-1-1-187:~/demo$ kubectl get deployments
NAME                    READY   UP-TO-DATE   AVAILABLE   AGE
hpa-demo-deployment     1/1     1            1           4m16s
ubuntu@ip-10-1-1-187:~/demo$ kubectl get rs
NAME                               DESIRED   CURRENT   READY   AGE
hpa-demo-deployment-75f99fc9f6     1         1         1       4m23s
ubuntu@ip-10-1-1-187:~/demo$ kubectl get hpa
NAME                    REFERENCE                          TARGETS       MINPODS   MAXPODS   REPLICAS   AGE
hpa-demo-deployment     Deployment/hpa-demo-deployment     cpu: 0%/50%   1         10        1          54s
ubuntu@ip-10-1-1-187:~/demo$ kubectl get pods
NAME                                    READY   STATUS    RESTARTS   AGE
hpa-demo-deployment-75f99fc9f6-rtsbp    1/1     Running   0          4m58s
ubuntu@ip-10-1-1-187:~/demo$
```

```
ubuntu@ip-10-1-1-187:~$ kubectl get pods
NAME                                    READY   STATUS                   RESTARTS   AGE
hpa-demo-deployment-75f99fc9f6-bf8j2    0/1     Completed                0          111s
hpa-demo-deployment-75f99fc9f6-gkrsf    0/1     ContainerStatusUnknown   1          111s
hpa-demo-deployment-75f99fc9f6-hxjzv    1/1     Running                  0          96s
hpa-demo-deployment-75f99fc9f6-md8hr    0/1     Pending                  0          41s
hpa-demo-deployment-75f99fc9f6-nt6th    0/1     Pending                  0          41s
hpa-demo-deployment-75f99fc9f6-pvfzp    1/1     Running                  0          96s
hpa-demo-deployment-75f99fc9f6-rtsbp    1/1     Running                  0          32m
load-generator                          1/1     Running                  0          2m24s
ubuntu@ip-10-1-1-187:~$ kubectl get hpa
NAME                    REFERENCE                          TARGETS       MINPODS   MAXPODS   REPLICAS   AGE
hpa-demo-deployment     Deployment/hpa-demo-deployment     cpu: 0%/50%   1         10        5          28m
ubuntu@ip-10-1-1-187:~$
```



Prepared by PPNREDDY .... 😊 😊 😊

Jobs

We have **ReplicaSets, DaemonSets, StatefulSets,** and **Deployments** they all share one common property, they ensure that their pods are always running. If a pod fails, the controller restarts it or reschedules it to another node to make sure the application the pods is hosting keeps running.

In Kubernetes, a **Job** is a controller that ensures a certain number of **Pods** are successfully completed. Unlike a regular controller like a Deployment (which runs Pods indefinitely), Jobs are designed to run tasks that should terminate once they finish successfully. A Job creates one or more Pods and ensures they complete their work.

Key Features of Kubernetes Jobs:

1. **One-time Execution**: Jobs are typically used for short-lived tasks. Once the Job's tasks (Pods) complete successfully, the Job is considered finished.
2. **Pod Failure Handling**: If a Pod fails during execution, Kubernetes can create a new Pod to retry the task, depending on the Job's configuration.
3. **Parallelism**: Jobs can run multiple Pods in parallel. The Job can be configured to run either:
    - One Pod that completes the task.
    - Multiple Pods running the same task to finish faster.

job1.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: testjob
spec:
  template:
    metadata:
      name: testjob
    spec:
      containers:
      - name: counter
        image: centos:7
        command: ["/bin/bash", "-c", "echo 'Hi-k8s job'; sleep 5"]
      restartPolicy: Never

commands:
=========
kubectl apply -f job1.yaml
watch kubectl get pods

job2.yaml
=========
apiVersion: batch/v1
```

```
kind: Job
metadata:
  name: testjob
spec:
  parallelism: 5                    # Runs for pods in parallel
  activeDeadlineSeconds: 10  # Timesout after 30 sec
  template:
    metadata:
      name: testjob
    spec:
      containers:
      - name: counter
        image: centos:7
        command: ["/bin/bash", "-c", "echo 'Hi-k8s job'; sleep 20"]
      restartPolicy: Never

commands:
=========
kubectl apply -f job2.yaml
watch kubectl get pods

job3.yaml
=========
apiVersion: batch/v1
kind: CronJob
metadata:
 name: ppreddy
spec:
 schedule: "*/2 * * * *"
 jobTemplate:
   spec:
     template:
       spec:
         containers:
         - image: ubuntu
           name: croncont
           command: ["/bin/bash", "-c", "echo 'Hi-k8s job'; sleep 20"]
         restartPolicy: Never
```

Prepared by PPNREDDY .... 😊 😊 😊

Scenario 01: Seeding a Database

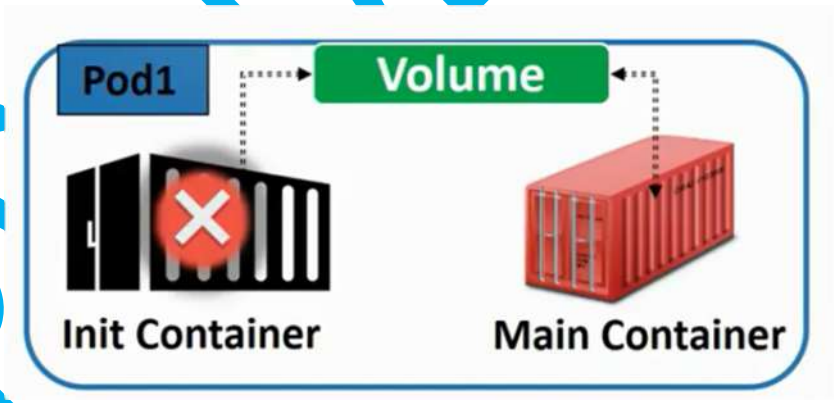
Scenario 02: Delaying The Application Launch Until The Dependencies Are Ready

Scenario 03: Clone a Git repository into a Volume

Scenario 04: Generate configuration file dynamically

Init containers in Kubernetes are specialized containers that run and complete before the main application container in a Pod starts. They are used to perform setup tasks or ensure certain conditions are met before the main container runs.

- Init containers are specialized containers that run before app containers in a Pod.
- Init containers always run to completion.
- If a Pod's init container fails, Kubernetes repeatedly restarts the Pod until the init container success.
- Init containers do not support readiness probes.



Key Features of Init Containers:

1. **Execution Order**:
   - Init containers run sequentially, one after another.
   - Each init container must complete successfully before the next one starts.
   - The main application container starts only after all init containers complete.

2. **Purpose**:
   - **Set up prerequisites** (e.g., fetching configuration files, waiting for external services to become available).
   - **Validate conditions** (e.g., checking database readiness).
   - **Customize the environment** (e.g., setting permissions or creating directories).
3. **Lifecycle**:
   - Init containers are ephemeral. Once completed, their states are not persisted.
4. **Isolation**:
   - They run in the same Pod as the application containers, so they share resources like volumes and network.
5. **Flexibility**:
   - You can use different images for init containers than the application container, allowing you to use specialized tools

init-pod-1.yaml

```
==============
apiVersion: v1
kind: Pod
metadata:
  name: initcontainer
spec:
  initContainers:
  - name: c1
    image: centos
    command: ["/bin/sh", "-c", "echo STAYHOME-STAYSAFE > /tmp/xchange/testfile; sleep 60"]
    volumeMounts:
      - name: xchange
        mountPath: "/tmp/xchange"
  containers:
  - name: c2
    image: centos
    command: ["/bin/bash", "-c", "while true; do echo `cat /tmp/data/testfile`; sleep 5; done"]
    volumeMounts:
      - name: xchange
        mountPath: "/tmp/data"
  volumes:
  - name: xchange
    emptyDir: {}
```

Commands:

```
kubectl apply -f init-pod-1.yaml
kubectl exec initcontainer -it -c c2 -- /bin/bash
kubectl exec initcontainer -it -c c1 -- /bin/bash
kubectl delete -f init-pod-1.yaml
```

init-pod-2.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: init-pod
spec:
  initContainers:
    - name: ic1
      image: busybox
      command: ['sh', '-c', 'echo Initializing ... ic1... && sleep 5']
    - name: ic2
      image: busybox
      command: ['sh', '-c', 'echo Initializing ... ic2... && sleep 5']
  containers:
    - name: main-init-cont
      image: httpd:latest
      ports:
       - containerPort: 80
```

Commands:

```
kubectl apply -f init-pod-2.yaml
kubectl exec initcontainer -it -c main-init-cont -- /bin/bash
kubectl exec initcontainer -it -c c1 -- /bin/bash
kubectl exec initcontainer -it -c c2 -- /bin/bash
kubectl delete -f init-pod-1.yaml
```

Prepared by PPNREDDY .... 😊 😊 😊

- Software deployment includes all the process required for preparing a software application to run and operate in a specific environment.
- It involves installation, configuration, testing and making changes to optimize the performance of the software.

Types of deployments

Recreate: Version A is terminated then version B is rolled out.

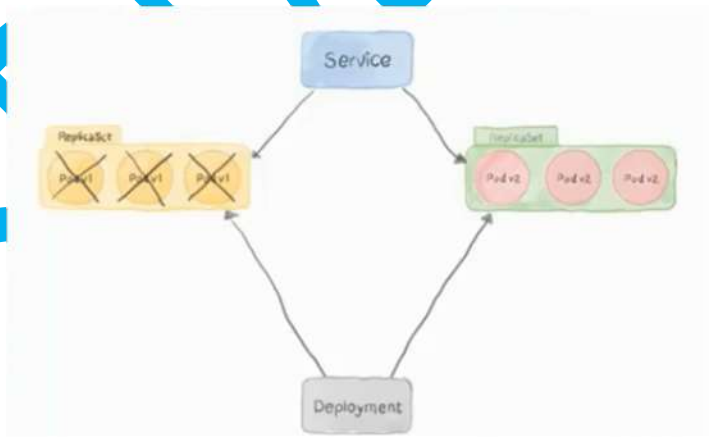
Ramped (also known as rolling-update or incremental): Version B is slowly rolled out and replacing version A.

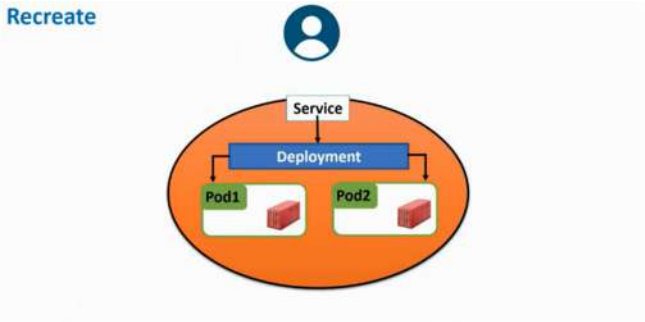
Blue/Green: Version B is released alongside version A, then the traffic is switched to version B.

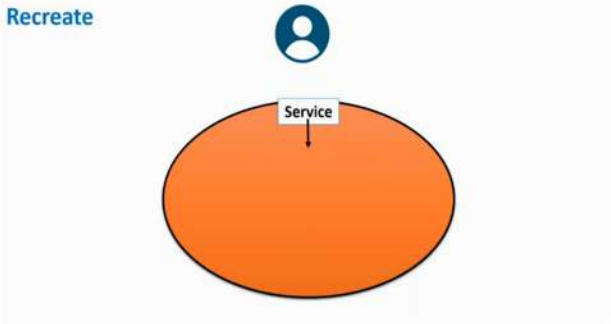
Canary: Version B is released to a subset of users, then proceed to a full rollout.

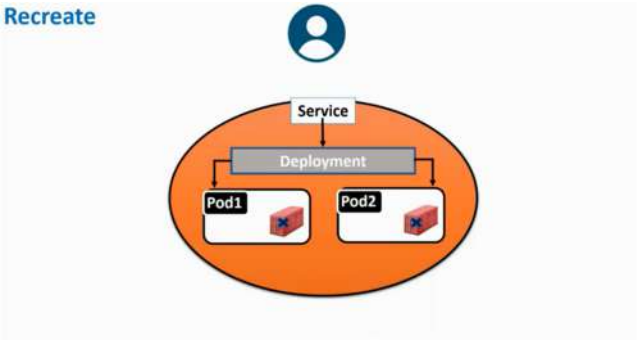
Recreate deployment

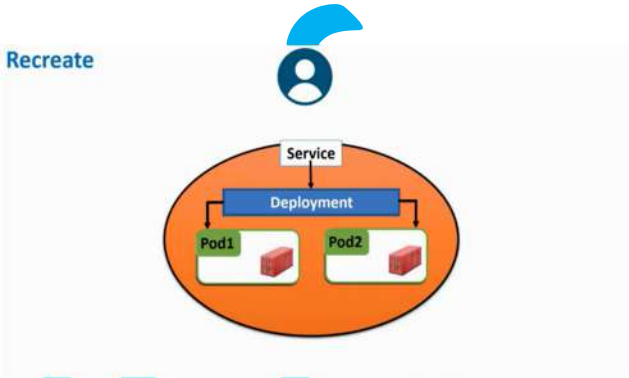
Recreating a deployment in Kubernetes involves replacing all the Pods managed by the Deployment with new ones. Application state entirely renewed, But Downtime that depends on both shutdown and boot duration of the application. It's suitable for development environment.











build-1.yaml

```
============
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-1.10
spec:
  replicas: 2
  selector:
    matchLabels:
      name: nginx
      version: "1.10"
  template:
    metadata:
      labels:
        name: nginx
        version: "1.10"
    spec:
      containers:
        - name: nginx
          image: nginx:1.10
          ports:
            - name: http
```

```
          containerPort: 80

build-2.yaml
============
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-1.10
spec:
  replicas: 2
  strategy:
    type: Recreate
  selector:
    matchLabels:
      name: nginx
      version: "1.10"
  template:
    metadata:
      labels:
        name: nginx
        version: "1.10"
    spec:
      containers:
        - name: nginx
          image: nginx:1.11
          ports:
            - name: http
              containerPort: 80


service.yaml
============
---
  kind: Service              # Defines to create Service type Object
  apiVersion: v1
  metadata:
    name: democipservice
  spec:
    ports:
      - port: 80             # Containers port exposed
        targetPort: 80       # Pods port
```

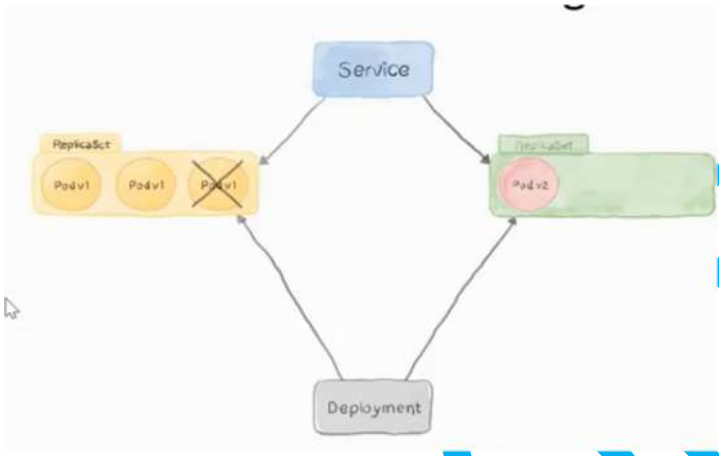
```
  selector:
    name: nginx
    version: "1.10"       # Apply this service to any pods which has the specific label
  type: ClusterIP         # Specifies the service type i.e ClusterIP or NodePort
```

Commands:

```
=========
kubectl apply -f build-1.yaml
kubectl apply -f service.yaml

kubectl describe service/democipservice
curl -s http://10.102.234.176:80/version

curl -s http://10.102.234.176:80/version

kubectl delete -f build-1.yaml
kubectl apply -f service.yaml
```

Ramped/Serial (also known as rolling-update or incremental):

- Version B is slowly rolled out and replacing version A.
- Suitable for QA, Stage or UAT environment.
- Stateful applications suitable on PROD environment

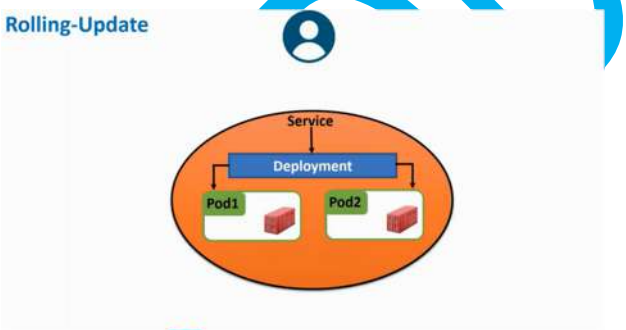


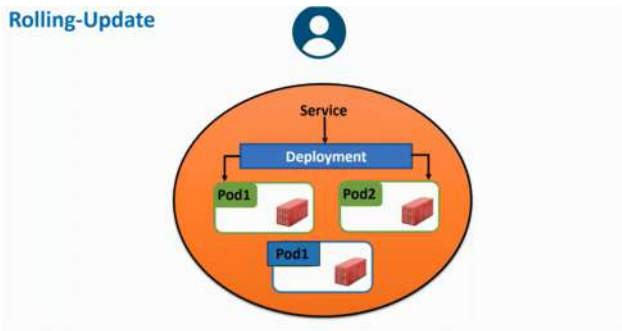
Pros:

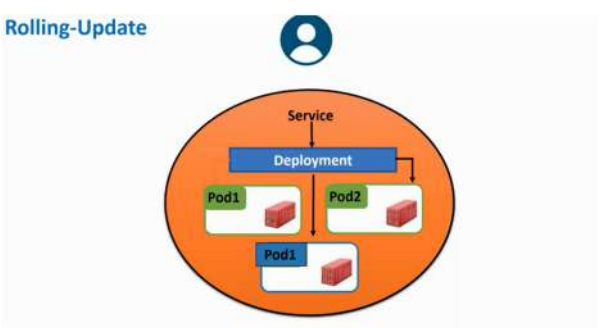
- Version is slowly released across instances.
- Convenient for stateful applications that can handle rebalancing of the data.

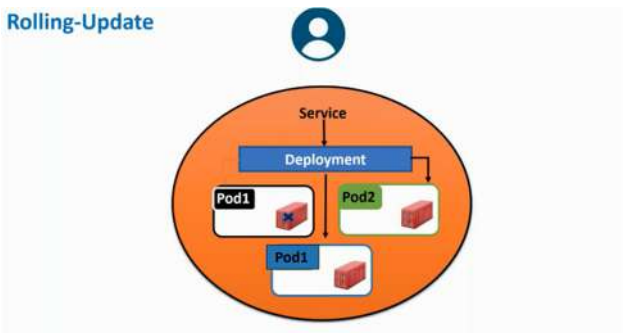
Cons:

- Rollout/rollback can take time.
- Supporting multiple APIs is hard.
- No control over traffic.

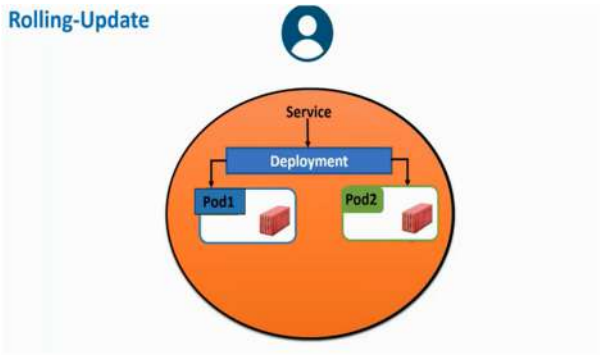


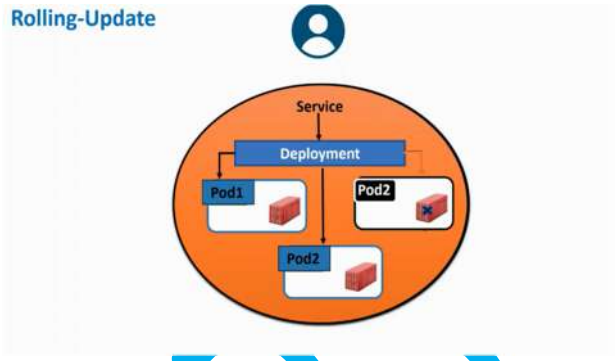


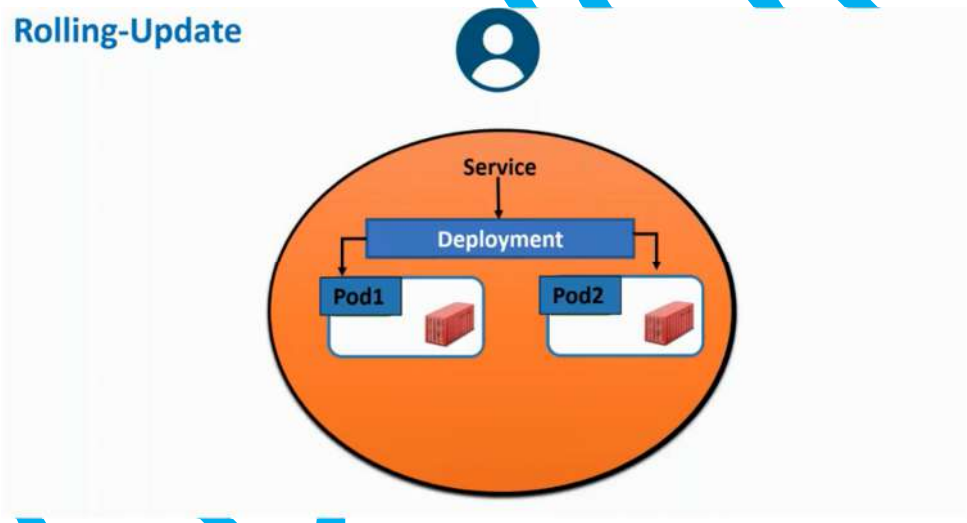




Prepared by PPNREDDY .... ☺ ☺ ☺







Demo

```
 rdbuild-1.yaml
==============
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-1.10
spec:
  replicas: 2
  selector:
```

```
    matchLabels:
      name: nginx
      version: "1.10"
  template:
    metadata:
      labels:
        name: nginx
        version: "1.10"
    spec:
      containers:
        - name: nginx
          image: nginx:1.10
          ports:
            - name: http
              containerPort: 80
```

rdbuild-2.yaml

==========

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-1.10
spec:
  replicas: 3
  selector:
    matchLabels:
      name: nginx
      version: "1.10"
  template:
    metadata:
      labels:
        name: nginx
        version: "1.10"
    spec:
      containers:
        - name: nginx
          image: nginx:1.11
          ports:
            - name: http
              containerPort: 80
```

service.yaml
==========

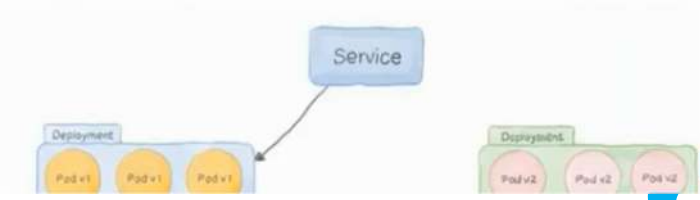
```
 kind: Service              # Defines to create Service type Object
 apiVersion: v1
 metadata:
   name: democipservice
 spec:
   ports:
    - port: 80             # Containers port exposed
      targetPort: 80       # Pods port
   selector:
      name: nginx
      version: "1.10"      # Apply this service to any pods which has the specific label
   type: ClusterIP         # Specifies the service type i.e ClusterIP or NodePort
```

Commands:
=========
```
kubectl apply -f rdbuild-1.yaml
kubectl apply -f service.yaml

kubectl describe service/democipservice
curl -s http://10.100.54.100:80/version

kubectl apply -f rdbuild-2.yaml
kubectl apply -f service.yaml
curl -s http://10.100.54.100:80/version
kubectl delete -f rdbuild-2.yaml
kubectl delete -f service.yaml
```

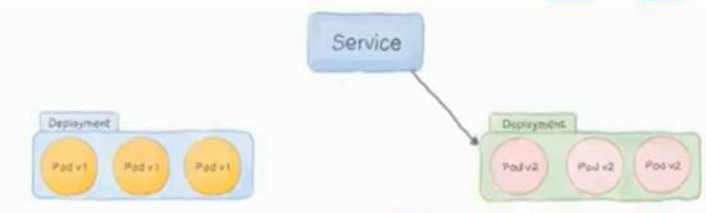
Blue-Green deployment

- Version B is released alongside version A, then the traffic is switched to version B.
- Suitable for Production environment

Version-1



Version-2

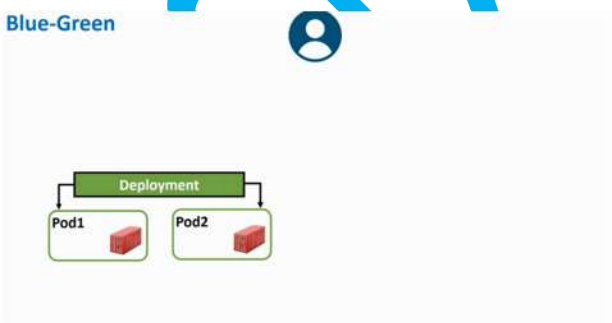


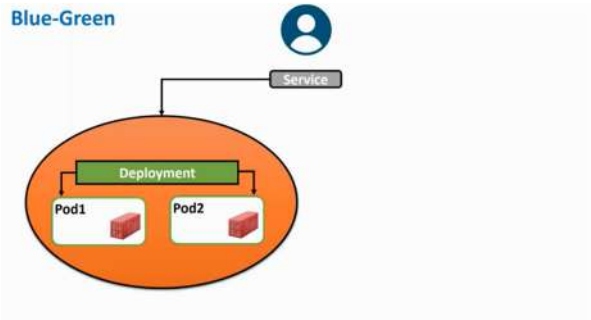
Pro:

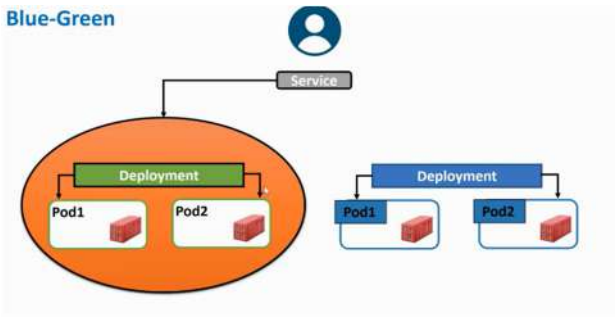
- Instant rollout/rollback.
- Avoid versioning issue, change the entire cluster state in one go.

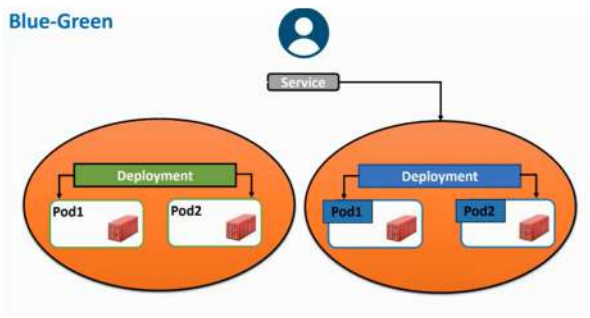
Cons:

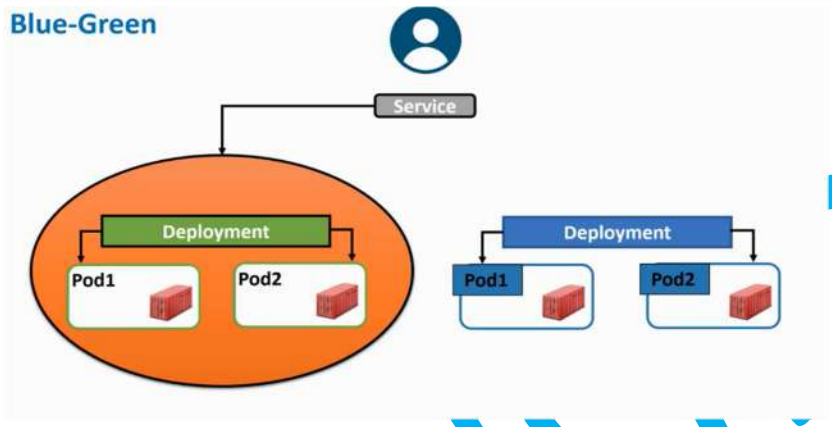
- Requires double the resources.
- Proper test of the entire platform should be done before releasing to production.
- Handling stateful applications can be hard











Demo

blue.yaml

```
---------
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-1.10
spec:
  replicas: 2
  selector:
    matchLabels:
      name: nginx
      version: "1.10"
  template:
    metadata:
      labels:
        name: nginx
        version: "1.10"
    spec:
      containers:
        - name: nginx
          image: nginx:1.10
          ports:
            - name: http
              containerPort: 80
```

bgservice.yaml

```
---
 kind: Service              # Defines to create Service type Object
 apiVersion: v1
 metadata:
  name: democipservice
 spec:
  ports:
   - port: 80              # Containers port exposed
     targetPort: 80        # Pods port
  selector:
     name: nginx
     version: "1.10"       # Apply this service to any pods which has the specific label
  type: ClusterIP          # Specifies the service type i.e ClusterIP or NodePort
```

green.yaml

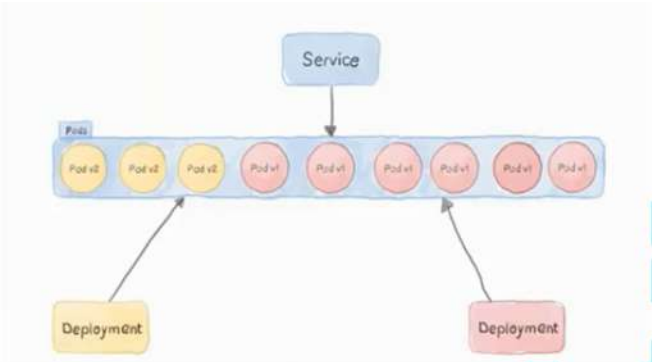
```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: nginx-1.11
spec:
 replicas: 2
 selector:
  matchLabels:
   name: nginx
   version: "1.11"
 template:
  metadata:
   labels:
    name: nginx
    version: "1.11"
  spec:
   containers:
    - name: nginx
      image: nginx:1.11
      ports:
       - name: http
         containerPort: 80
```

Commands:

```
watch kubectl get all

kubectl apply -f blue.yaml
kubectl apply -f bgservice.yaml
kubectl describe service/democipservice

curl -s http://10.110.71.254/version

kubectl apply -f green.yaml

update "bgservice.yaml" and apply the changes
   selector:
      name: nginx
      version: "1.11"       # Apply this service to any pods which has the specific label
   type: ClusterIP          # Specifies the service type i.e ClusterIP or NodePort


kubectl apply -f bgservice.yaml
curl -s http://10.110.71.254/version

kubectl delete -f bgservice.yaml
kubectl delete -f blue.yaml
kubectl delete -f green.yaml
```

Canary Deployment

- Version B is released to a subset of users, then proceed to a full rollout.
- Suitable for Production environment.

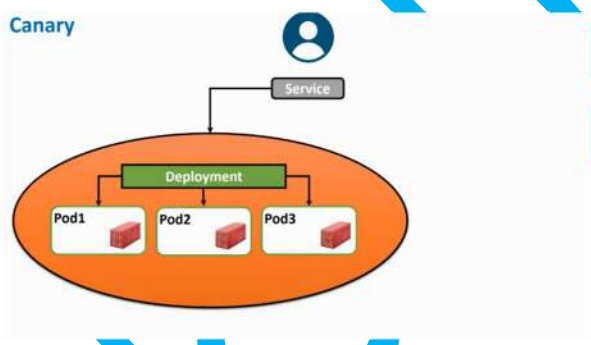


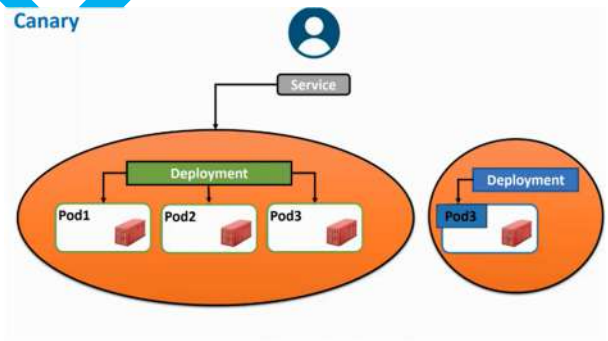
Pro:

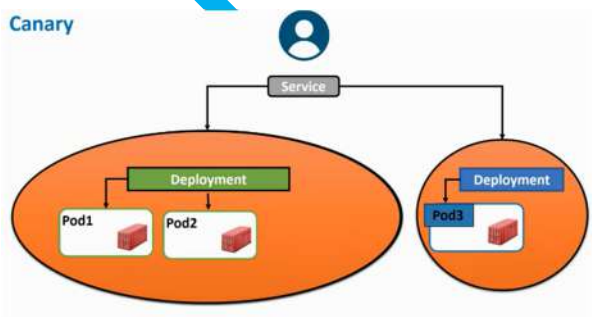
- Version released for a subset of users.
- Convenient for error rate and performance monitoring.
- Fast rollback

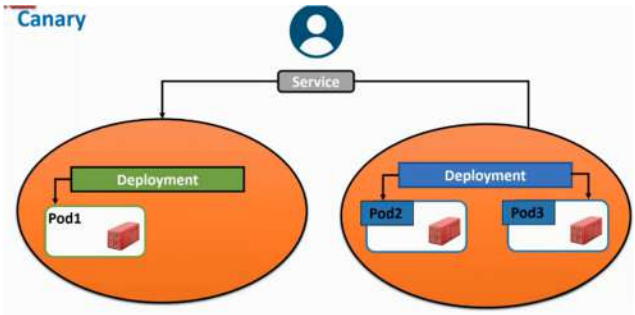
Cons:

- Slow Rollout.
- Fine-tuned traffic distribution can be expensive

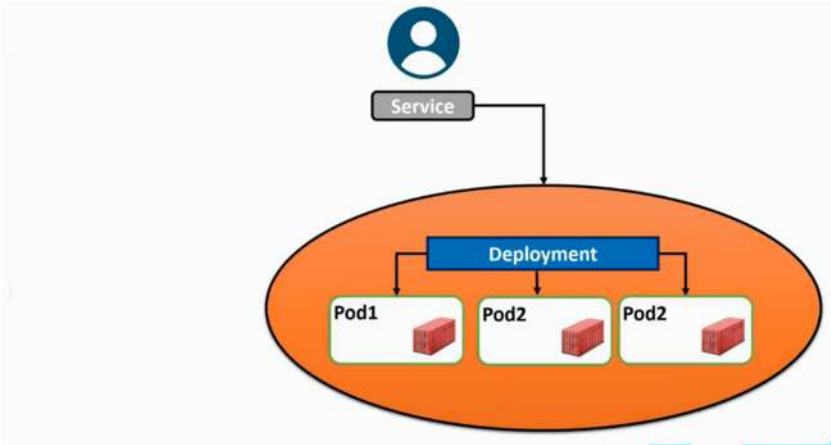








Prepared by PPNREDDY .... ☺ ☺ ☺



Demo

cbuild-1.yaml

==========

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-1.10
spec:
  replicas: 3
  selector:
    matchLabels:
      name: nginx
  template:
    metadata:
      labels:
        name: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.10
          ports:
            - name: http
              containerPort: 80
```

cbuild-2.yaml

==========

```
apiVersion: apps/v1
kind: Deployment
metadata:
```

```
  name: nginx-1.11
spec:
  replicas: 1
  selector:
    matchLabels:
      name: nginx
  template:
    metadata:
      labels:
        name: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.11
          ports:
            - name: http
              containerPort: 80
```

cservice.yaml

```
=============
---
  kind: Service              # Defines to create Service type Object
  apiVersion: v1
  metadata:
    name: democipservice
  spec:
    ports:
      - port: 80             # Containers port exposed
        targetPort: 80       # Pods port
    selector:
        name: nginx          # Apply this service to any pods which has the specific label
    type: ClusterIP          # Specifies the service type i.e ClusterIP or NodePort
```

Commands:

```
=========
kubectl apply -f cbuild-1.yaml
kubectl apply -f cservice.yaml

kubectl describe service/democipservice
curl -s http://10.104.61.120/version
```

kubectl apply -f cbuild-2.yaml
curl -s http://10.104.61.120/version

update the replicas:

cbuild-1.yaml --> replcas:3
cbuild-2.yaml --> replcas:1
curl -s http://10.104.61.120/version

cbuild-1.yaml --> replcas:2
cbuild-2.yaml --> replcas:2
kubectl apply -f cbuild-1.yaml
kubectl apply -f cbuild-2.yaml
curl -s http://10.104.61.120/version

cbuild-1.yaml --> replcas:1
cbuild-2.yaml --> replcas:3
kubectl apply -f cbuild-1.yaml
kubectl apply -f cbuild-2.yaml
curl -s http://10.104.61.120/version

kubectl delete -f cbuild-1.yaml
curl -s http://10.104.61.120/version
kubectl delete -f cbuild-1.yaml
kubectl delete -f cservice.yaml

Node affinity

Node affinity in Kubernetes is a feature that enables you to control which nodes a pod can be scheduled on based on the attributes (labels) of the nodes. It enhances the scheduler's ability to make intelligent decisions about where to place pods. Node affinity comes in two main forms.

Required Node Affinity (requiredDuringSchedulingIgnoredDuringExecution)

This is a **hard** requirement; if no suitable nodes are available, the Pod will not be scheduled.

Mandatory rules: Pods are only scheduled on nodes that match the specified criteria.

Syntax: If no nodes satisfy the condition, the pod will remain unscheduled.

Use case: Critical constraints, such as ensuring a workload only runs on specific node types (e.g., nodes with SSDs or specific hardware).

```
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: "disktype"
          operator: "In"
          values:
          - "ssd"
```

Preferred Node Affinity (preferredDuringSchedulingIgnoredDuringExecution)

This is a **soft requirement**.

Soft preferences: Tries to schedule pods on nodes that match the criteria but will schedule them elsewhere if no matches are found.

Syntax: Includes a weight field (1-100), which determines the priority of the preference.

Use case: Desirable constraints, such as preferring a workload to run on nodes in a particular availability zone.

```
affinity:
  nodeAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
    - weight: 100
      preference:
        matchExpressions:
        - key: "region"
          operator: "In"
          values:
          - "us-west-1"
```

Key Components of Node Affinity

- **key**: The node label key to match (e.g., disktype).
- **operator**: Specifies the matching logic:
  - In: The node label value must match one of the specified values.
  - NotIn: The node label value must not match any of the specified values.
  - Exists: The node must have the label key, regardless of its value.
  - DoesNotExist: The node must not have the label key.
  - Gt/Lt: The node label value must be greater/less than a specified value (for numeric labels).
- **values**: Specifies the acceptable label values.

hardna.yaml

```
==========
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: disktype
                operator: In
                values:
                  - ssd
  containers:
    - name: uc
      image: ubuntu:latest
      command: ["/bin/bash", "-c", "while true; do echo Hello-DevOps; sleep 5; done"]
```

commands

```
========
kubectl get nodes --show-labels
kubectl apply -f hardna.yaml
kubectl get pods -o wide
kubectl delete -f hardna.yaml
```

softna.yaml

```
==========
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
  affinity:
    nodeAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 1
          preference:
            matchExpressions:
              - key: disktype
                operator: In
                values:
                  - ssd1
  containers:
    - name: uc
      image: ubuntu:latest
      command: ["/bin/bash", "-c", "while true; do echo Hello-DevOps; sleep 5; done"]
```

Commands

```
========
kubectl apply -f softna.yaml
kubectl get pods -o wide
kubectl delete -f softna.yaml
```

Node anti-affinity

Node anti-affinity in Kubernetes is a scheduling concept that helps you ensure that certain pods are not scheduled on specific nodes or are distributed across nodes to avoid being placed on the same one. It works by expressing preferences or rules to guide the Kubernetes scheduler.

Key Components of Node Anti-Affinity

1. **Anti-Affinity Rules**:
   - Define conditions under which pods should **avoid** being placed together or on specific nodes.
2. **Node Selector Terms**:
   - Specify the labels on nodes that the anti-affinity rule will evaluate.
3. **Types of Node Anti-Affinity**:
   - **RequiredDuringSchedulingIgnoredDuringExecution**:
     - The scheduler **must** respect this rule when placing pods.
     - Ignored once the pod is running (i.e., it won't evict running pods even if the rule is violated).
   - **PreferredDuringSchedulingIgnoredDuringExecution**:
     - The scheduler **tries** to respect this rule but is not strictly required to do so.
     - Used for softer constraints or preferences.

hardnaa.yaml

```
============
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: disktype
                operator: NotIn
                values:
                  - hdd
  containers:
    - name: uc
      image: ubuntu:latest
      command: ["/bin/bash", "-c", "while true; do echo Hello-DevOps; sleep 5; done"]
```

commands
========
kubectl get nodes --show-labels
kubectl apply -f hardnaa.yaml
kubectl get pods -o wide
kubectl delete -f hardnaa.yaml

softna.yaml
===========

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
  affinity:
    nodeAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 1
          preference:
            matchExpressions:
              - key: disktype
                operator: NotIn
                values:
                  - ssd1
  containers:
    - name: uc
      image: ubuntu:latest
      command: ["/bin/bash", "-c", "while true; do echo Hello-DevOps; sleep 5; done"]
```

Commands
========
kubectl apply -f softnaa.yaml
kubectl get pods -o wide
kubectl delete -f softnaa.yaml

Pod affinity

Pod affinity and pod anti-affinity in Kubernetes are mechanisms to influence how pods are scheduled relative to other pods based on labels.

Pod affinity allows you to specify rules to schedule pods **close to** other pods that match specific criteria. It is often used when co-locating pods improves performance, such as applications that communicate heavily or share data.

Types of Pods Affinity

1. **Hard Affinity (RequiredDuringSchedulingIgnoredDuringExecution):**
   - The rule **must** be satisfied for the pod to be scheduled.
   - If no node meets the conditions, the pod will remain unscheduled.
2. **Soft Affinity (PreferredDuringSchedulingIgnoredDuringExecution):**
   - The rule **should** be satisfied, but the scheduler may place the pod elsewhere if it cannot meet the condition.

samplepod.yaml

```
---
 kind: Pod
 apiVersion: v1
 metadata:
  name: samplepod
  labels:                  # Specifies the Label details under it
   app: psddevops
 spec:
  containers:
   - name: c1
     image: ubuntu:latest
     command: ["/bin/bash", "-c", "while true; do echo Hello-psddevops; sleep 5 ; done"]
  nodeSelector:
   disktype: ssd
```

Commands

```
kubectl apply -f samplepod.yaml
kubectl get pods -o wide
```

testpod.yaml

```
---
 kind: Pod
 apiVersion: v1
 metadata:
  name: testpod
```

```
  labels:                 # Specifies the Label details under it
    team: dev
 spec:
  containers:
   - name: c1
     image: ubuntu:latest
     command: ["/bin/bash", "-c", "while true; do echo Hello-psddevops; sleep 5 ; done"]
  nodeSelector:
    disktype: hdd
```

Commands

```
kubectl apply -f testpod.yaml
kubectl get pods -o wide
```

hardpa.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: web-server
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: team
            operator: In  # verify In, NotIn conditions
            values:
            - dev
        topologyKey: "kubernetes.io/hostname"
  containers:
  - name: web-server
    image: nginx
```

Commands:

```
kubectl apply -f hardpa.yaml
kubectl get pods -o wide
```

softpa.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: web-server
spec:
  affinity:
    podAffinity:
      # Soft Rule: Prefer to co-locate with frontend pods
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 80
        podAffinityTerm:
          labelSelector:
            matchExpressions:
            - key: app
              operator: NotIn
              values:
              - psddevops
          topologyKey: "kubernetes.io/hostname"
  containers:
  - name: web-server
    image: nginx
```

Commands:

```
kubectl apply -f softpa.yaml
```


Explanation of Fields

Hard Rule (requiredDuringSchedulingIgnoredDuringExecution)

- **labelSelector**: Defines the label criteria to match pods for affinity.
- **topologyKey**: Defines the scope of affinity, such as node-level (kubernetes.io/hostname) or zone-level (failure-domain.beta.kubernetes.io/zone).
- **Effect**: If no matching pods are on a node, the pod remains unscheduled.

Soft Rule (preferredDuringSchedulingIgnoredDuringExecution)

- **weight**: A value from 1-100 indicating priority; higher weight = stronger preference.
- **Effect**: Scheduler prefers nodes with matching pods but will schedule on other nodes if necessary.

Pod anti-affinity

Pod anti-affinity ensures that pods **avoid** being scheduled close to other pods with specific labels. This is useful for improving fault tolerance or ensuring resource distribution.

Types of Pod Anti-Affinity

1. **Hard Anti-Affinity (RequiredDuringSchedulingIgnoredDuringExecution):**
   - Pods **must not** run on the same node as the specified pods.
   - The pod remains unscheduled if no suitable node is available.
2. **Soft Anti-Affinity (PreferredDuringSchedulingIgnoredDuringExecution):**
   - The scheduler tries to avoid placing pods together, but it's not guaranteed.

Use Cases

Pod Affinity

- **Performance Optimization**:
  - Co-locate microservices that communicate heavily to minimize network latency.
- **Shared Resources**:
  - Place pods near shared caches, storage, or databases.

Pod Anti-Affinity

- **High Availability**:
  - Distribute replicas across different nodes to avoid single points of failure.
- **Resource Isolation**:
  - Prevent resource-intensive pods from running on the same nodes.

samplepod.yaml

```
==============
---
 kind: Pod
 apiVersion: v1
 metadata:
  name: samplepod
  labels:                # Specifies the Label details under it
   app: psddevops
 spec:
  containers:
   - name: c1
     image: ubuntu:latest
     command: ["/bin/bash", "-c", "while true; do echo Hello-psddevops; sleep 5 ; done"]
  nodeSelector:
   disktype: ssd
```

Commands
========
kubectl apply -f samplepod.yaml
kubectl get pods -o wide

testpod.yaml
============
```
---
 kind: Pod
 apiVersion: v1
 metadata:
  name: testpod
  labels:                  # Specifies the Label details under it
   team: dev
 spec:
  containers:
   - name: c1
     image: ubuntu:latest
     command: ["/bin/bash", "-c", "while true; do echo Hello-psddevops; sleep 5 ; done"]
  nodeSelector:
   disktype: hdd
```

Commands
========
kubectl apply -f testpod.yaml
kubectl get pods -o wide

hardpaa.yaml
============

```
apiVersion: v1
kind: Pod
metadata:
 name: webserverpod
spec:
 affinity:
  podAntiAffinity:
   requiredDuringSchedulingIgnoredDuringExecution:
   - labelSelector:
```

```
        matchExpressions:
        - key: app
          operator: In
          values:
          - psddevops
      topologyKey: "kubernetes.io/hostname"
  containers:
  - name: nginx-cont
    image: nginx
```

softpaa.yaml

```
============
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
  labels:
    app: backend
spec:
  affinity:
    podAntiAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 50
        podAffinityTerm:
          labelSelector:
            matchExpressions:
            - key: app
              operator: In
              values:
              - psddevops
          topologyKey: "kubernetes.io/hostname"
  containers:
  - name: nginx
    image: nginx
```

Taints and Tolerations

Taints and **Tolerations** in Kubernetes are mechanisms to ensure that pods are scheduled on the appropriate nodes.

Taints: Applied to nodes to repel certain pods.

Tolerations: Applied to pods to let them tolerate a node's taints.

This is useful for:

- Dedicated nodes for specific workloads (e.g., system pods, GPU workloads).
- Preventing overloading of specific nodes.

Use Cases

1. **Dedicated Nodes**: Assign certain nodes exclusively for specific workloads.
2. **High-Availability**: Prevent system-critical pods from being evicted or misplaced.
3. **Resource Isolation**: Separate resource-intensive workloads like GPU or AI processing.

Apply a Taint to Nodes

kubectl taint nodes worker1 dedicated=logging:NoSchedule

This taint ensures that no pods, except those with a matching toleration, are scheduled on worker1.

Apply a Toleration to Pods

To schedule pods on the tainted node, add a **toleration** in the pod's manifest

```
 apiVersion: v1
kind: Pod
metadata:
  name: logging-pod
spec:
  tolerations:
  - key: "dedicated"
    operator: "Equal"
    value: "logging"
    effect: "NoSchedule"
  containers:
  - name: logging-container
    image: nginx:latest
```

Key: Matches the taint key on the node.

Operator: Specifies how the toleration is applied (Equal or Exists).

Value: Matches the taint value.

Effect: Must match the taint effect (NoSchedule, PreferNoSchedule, or NoExecute).

Types of Taint Effects

1. **NoSchedule**: Pods without matching tolerations are not scheduled on the node.
2. **PreferNoSchedule**: Kubernetes avoids scheduling pods on the node if possible.
3. **NoExecute**: Existing pods without tolerations are evicted.

kubectl taint nodes worker-2 critical=important:NoExecute

```
apiVersion: v1
kind: Pod
metadata:
  name: important-pod
spec:
  tolerations:
  - key: "critical"
    operator: "Equal"
    value: "important"
    effect: "NoExecute"
  containers:
  - name: important-container
    image: nginx

Commands
========
kubectl apply -f ttpod1.yaml
kubectl get pods -o wide
kubectl delete -f ttpod1.yaml
```

*** Pods without this toleration are immediately evicted from worker-2

Removed the taints

kubectl taint nodes worker-1 dedicated=logging:NoSchedule-
kubectl taint nodes worker-2 critical=important:NoExecute-

Drain, Cordon and Uncordon

In Kubernetes, **drain**, **cordon**, and **uncordon** are commands used to manage node workloads, typically when performing maintenance or upgrades.

Cordon:

The cordon command marks a node as **unschedulable**, which prevents new pods from being scheduled on the node. Existing pods, however, remain unaffected and continue to run.

```
kubectl cordon <node-name>
```

Drain:

The drain command evicts all pods from a node and marks it as **unschedulable**. This is typically used to safely prepare a node for maintenance or decommissioning.

```
kubectl drain <node-name> --ignore-daemonsets --delete-emptydir-data
```

Options:

- --ignore-daemonsets: DaemonSet pods are ignored (since they are tied to the node).
- --delete-emptydir-data: Pods with emptyDir volumes are forcibly deleted.
- --force: Force deletion of pods that are not managed by a controller (e.g., static pods).

Uncordon:

The uncordon command reverses the cordon operation, marking the node as **schedulable** again. New pods can now be scheduled on the node.

```
kubectl uncordon <node-name>
```

drain-cordon-uncordon-deployment.yaml

```
==================================
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginxdeploy
spec:
  replicas: 2
  selector:
    matchLabels:
      name: nginx
  template:
    metadata:
      labels:
```

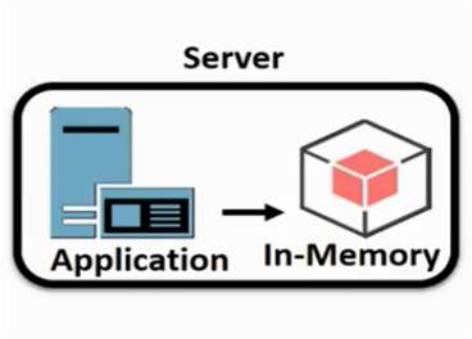
```
    name: nginx
  spec:
    containers:
     - name: nginx
       image: nginx:1.11
       ports:
         - name: http
           containerPort: 80
```

Commands:

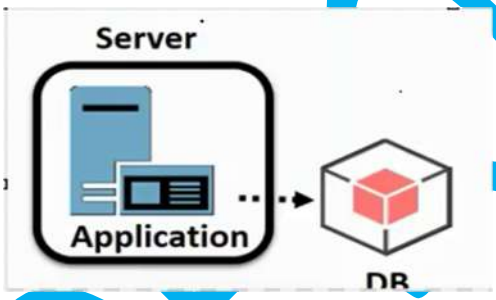
```
kubectl apply -f drain-cordon-uncordon-deployment.yaml
kubectl get pods -o wide

kubectl cordon worker-1
kubectl get nodes

kubectl drain worker-1 --ignore-daemonsets --delete-emptydir-data
kubectl get pods -o wide

kubectl uncordon worker-1
kubectl get pods -o wide
kubectl delete pod nginxdeploy-598d5c655b-bv6rz
kubectl get pods -o wide
kubectl delete -f drain-cordon-uncordon-deployment.yaml
```

Prepared by PPNREDDY .... 😊 😊 😊

Stateful applications store data about the state from client requests on the server itself and use that state to process further requests.
Examples of Stateful applications include MongoDB, Cassandra, and MySQL.

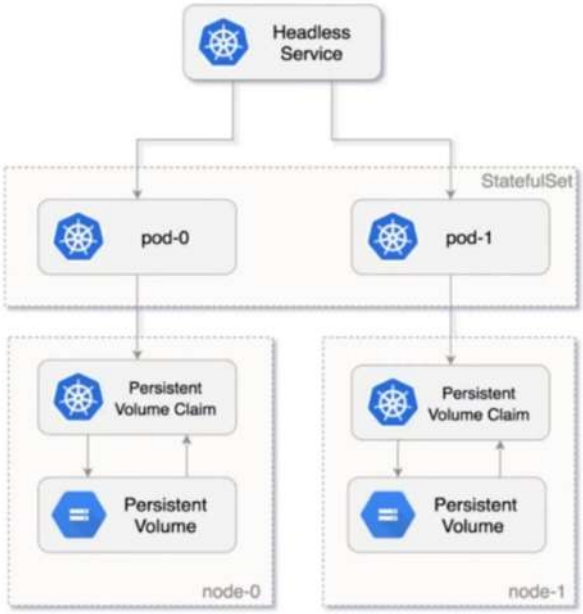


Stateless app is an application program that does not save client data generated in one session for use in the next session with that client.
Examples of stateless applications include major websites (Amazon, Google, Facebook, Twitter), WhatsApp, AWS (S3, EBS).



Deployments are usually used for stateless applications. However, you can save the state of deployment by attaching a Persistent Volume to it and make it stateful, but all the pods of a deployment will be sharing the same Volume and data across all of them Will be same.

- StatefulSet is a Kubernetes resource used to manage stateful applications. It manages the deployment and scaling of a set of Pods, and provides guarantee about the ordering and uniqueness of these Pods.
- StatefulSet is also a Controller but unlike Deployments, it doesn't create ReplicaSet rather itself creates the Pod with a unique naming convention. e.g. If you create a StatefulSet with name mongo, it will create a pod with name mongo-0, and for multiple replicas of a statefulset, their names will increment like mongo-0, mongo-1, mongo-2, etc.
- Every replica of a stateful set will have its own state, and each of the pods will be creating its own PVC (Persistent Volume Claim). So, a statefulset with 3 replicas will create 3 pods, each having its own Volume, so total 3 PVCs.
- By far the most common way to run a database, StatefulSets is a feature fully supported as of the Kubernetes 1.9 release. Using it, each of your pods is guaranteed the same network identity and disk across restarts, even if it's rescheduled to a different physical machine.



StatefulSet Deployments provide:

Stable, unique network identifiers:

Each pod in a StatefulSet is given a hostname that is based on the application name and increment. For example, mongo-1, mongo-2 and mongo-3 for a StatefulSet named "mongo" that has 3 instances running.

Stable, persistent storage:

Each and every pod in the cluster is given its own persistent volume based on the storage class defined, or the default, if none are defined. Deleting or scaling down pods will not automatically delete the volumes associated with them- so that the data persists. you could scale the StatefulSet down to 0 first, prior to deletion of the unused pods.

Ordered, graceful deployment and scaling:

Pods for the StatefulSet are created and brought online in order, from 1 to n, and they are shut down in reverse order to ensure a reliable and repeatable deployment and runtime. The StatefulSet will not even scale until all the required pods are running, so if one dies, it recreates the pod before attempting to add additional instances to meet the scaling criteria.

Ordered, automated rolling updates:

StatefulSets have the ability to handle upgrades in a rolling manner where it shuts down and rebuilds each node in the order it was created originally, continuing this until all the old versions have been shut down and cleaned up. Persistent volumes are reused, and data is automatically migrated to the upgraded version.

nginx-service.yaml

==========

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
```

```
spec:
  clusterIP: None  # Headless service
  selector:
    app: nginx
  ports:
    - port: 80
      targetPort: 80

storageclass.yaml
=================
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: gp2
provisioner: kubernetes.io/aws-ebs
volumeBindingMode: WaitForFirstConsumer

statefulset.yaml
=============

apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: nginx
spec:
  serviceName: "nginx"
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.21
          ports:
            - containerPort: 80
          volumeMounts:
```

```
        - name: nginx-data
          mountPath: /usr/share/nginx/html  # Persisted content directory
volumeClaimTemplates:
  - metadata:
      name: nginx-data
    spec:
      accessModes: ["ReadWriteOnce"]
      storageClassName: "gp2"  # Use the StorageClass created earlier
      resources:
        requests:
          storage: 1Gi
```

Prepared by PPNREDDY .... 😊 😊 😊

Kubernetes Role Based Access Control (RBAC)

When a request is sent to the API Server, it first needs to be authenticated (to make sure the requestor is known by the system) before it's authorized (to make sure the requestor is allowed to perform the action requested).

RBAC is a way to define which users can do what within a Kubernetes cluster.

If you are working on Kubernetes for some time, you may have faced a scenario where you have to give some users limited access to your Kubernetes cluster. For example, you may want a user, say Michale from development, to have access only to some resources that are in the development namespace and nothing else. To achieve this type of role-based access, we use the concept of Authentication and Authorization in Kubernetes.

Broadly, there are three kinds of users that need access to a Kubernetes cluster:

1. Developers/Admins:

Users that are responsible to do administrative or developmental tasks on the cluster. This includes operations like upgrading the cluster or creating the resources/workloads on the cluster.

2. End Users:

Users that access the applications deployed on our Kubernetes cluster. Access restrictions for these users are managed by the applications themselves. For example, a web application running on Kubernetes cluster, will have its own security mechanism in place, to prevent unauthorized access.

3. Applications/Bots:

There is a possibility that other applications need access to Kubernetes cluster, typically to talk to resources or workloads inside the cluster. Kubernetes facilitates this by using Service Accounts.

RBAC concepts The RBAC model in Kubernetes is based on three elements:

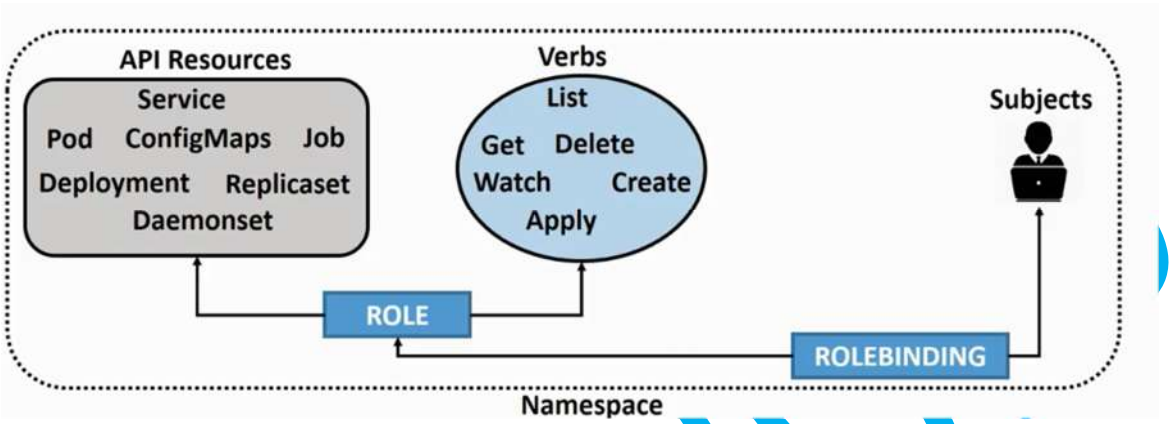
API Resources: The set of Kubernetes API Objects available in the cluster. Examples like Pods, Deployments, Services, Nodes, and PersistentVolumes, among others

Subjects: These are the objects (Users, Groups, Processes (Service Account)) allowed access to the API, based on Verbs and Resources.

Verbs: The set of operations that can be executed to the resources above. Different verbs are available (examples: get, watch, create, delete, etc.), but ultimately all of them are Create, Read, Update or Delete

Understanding RBAC API Objects: Roles & RoleBinding

- Roles will connect API Resources and Verbs specific to one namespace.
- Roles can be reused for different subjects by Binding to specific entity-subjects.
- If we want the role to be applied cluster-wide, the equivalent object is called ClusterRoles and use ClusterRoleBindings for binding to cluster-level subjects.




```yaml
kind: Role
apiVersion:
rbac.authorization.k8s.io/v1beta1
metadata:
   name: pod-read-create
   namespace: test
rules:
 - apiGroups: [""]
   resources: ["pods"]
   verbs: ["get", "list", "create"]
```

```yaml
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: salme-pods
  namespace: test
subjects:
- kind: User
  name: jsalmeron
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: ns-admin
  apiGroup: rbac.authorization.k8s.io
```

RBAC Role

A Role example named example-role which allows access to the mynamespace with get, watch, and list operations:

kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
namespace: mynamespace
name: example-role
rules:

- apiGroups: [""]
resources: ["pods"]
verbs: ["get", "watch", "list"]
In the rules above we:
1. apiGroups: [""] – set core API group
2. resources: ["pods"] – which resources are allowed for access
3. ["get", "watch", "list"] – which actions are allowed over the resources above

RBAC RoleBinding

To "map" those permissions to users we are using Kubernetes RoleBinding, which sets example-role in the mynamespace for the example-user user:

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
name: example-rolebinding
namespace: mynamespace
subjects:
- kind: User
name: example-user
apiGroup: rbac.authorization.k8s.io
roleRef:
kind: Role
name: example-role
apiGroup: rbac.authorization.k8s.io
```

Here we set:

- subjects: ▪ kind: User – an object type which will have access, in our case this is a regular user
- name: example-user – a user's name to set the permissions
- roleRef: ▪ kind: Role – what exactly will be attached to the user, in this case, it is the Role object type
- name: example-role – and the role name as it was set in the name: example-role in the example above

Role vs ClusterRole

Alongside with the Role and ClusterRole which are set of rules to describe permissions – Kubernetes also has RoleBinding and ClusterRoleBinding objects.
The difference is that Role is used inside of a namespace, while ClusterRole is cluster-wide permission without a namespace boundary, for example:

- allow access to a cluster nodes.
- resources in all namespaces.
- allow access to endpoints like /healthz

A ClusterRole looks similar to a Role with the only difference that we have to set its kind as ClusterRole:

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
name: example-clusterrole
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

ClusterRoleBinding example:

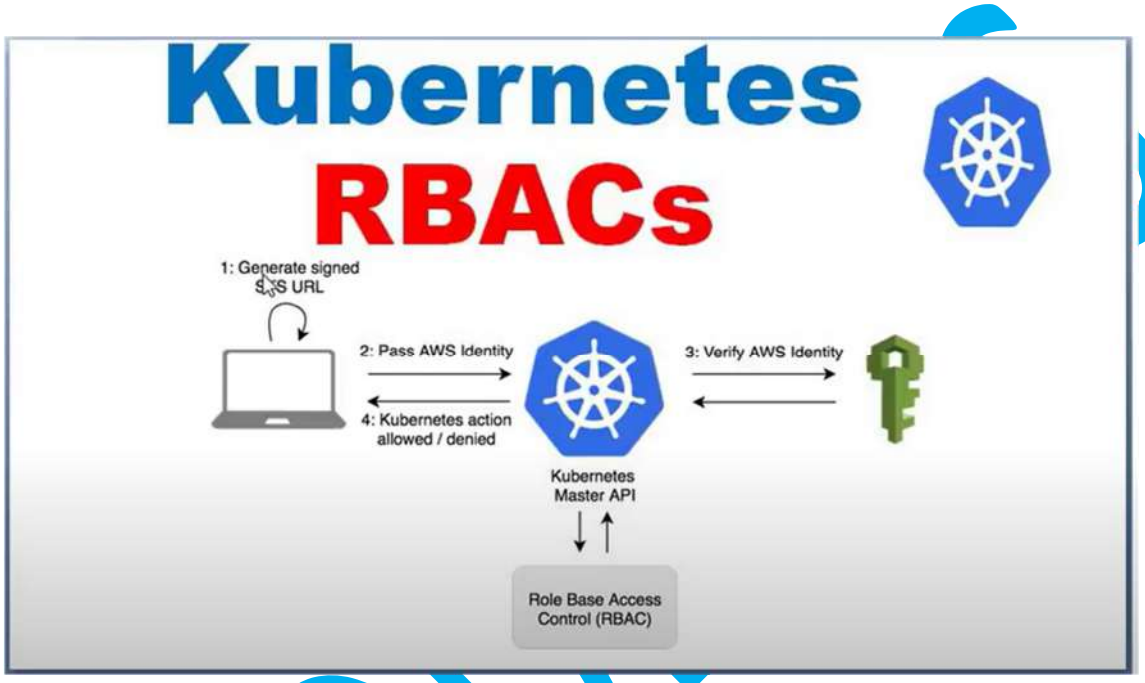
```
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
name: example-clusterrolebinding
subjects:
- kind: User
name: example-user
apiGroup: rbac.authorization.k8s.io
roleRef:
kind: ClusterRole
name: example-clusterrole
apiGroup: rbac.authorization.k8s.io
```

Keep in mind that once you'll create a Binding you'll not be able to edit its roleRef value – instead, you'll have to delete a Binding and recreate and again.

- Kubernetes uses RBAC to control different access levels to its resources depending on the rules set in Roles or ClusterRoles.
- Roles and ClusterRoles use API namespaces, verbs and resources to secure access.

- Roles and ClusterRoles are ineffective unless they are linked to a subject (User, serviceAccount...etc) through RoleBinding or ClusterRoleBinding.
- Roles work within the constraints of a namespace. It would default to the “default” namespace if none was specified.
- ClusterRoles are not bound to a specific namespace as they apply to the cluster as a whole.

Demo




```
kubectl create namespace rbac-test

kubectl get ns

## Deploy nginx pod on clsuster
kubectl create deploy nginx --image=nginx -n rbac-test

## Create IAM user and create access key
aws iam create-user --user-name rbac-user

{
    "User": {
        "Path": "/",
        "UserName": "rbac-user",
        "UserId": "AIDAXYKJRU5VPCZYTAKO2",
        "Arn": "arn:aws:iam::533267064682:user/rbac-user",
```

```
      "CreateDate": "2024-11-24T05:27:12+00:00"
    }
}

aws iam create-access-key --user-name rbac-user

{
    "AccessKey": {
        "UserName": "rbac-user",
        "AccessKeyId": "AKIAXYKJRU5VEJRTXTFL",
        "Status": "Active",
        "SecretAccessKey": "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",
        "CreateDate": "2024-11-24T05:28:21+00:00"
    }
}

Configure the credentials in separate window and login

PS C:\Users\ppred> aws configure
AWS Access Key ID [****************7KNV]: AKIAXYKJRU5VEJRTXTFL
AWS Secret Access Key [****************iZ5a]:
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
Default region name [us-east-2]:
Default output format [json]:
PS C:\Users\ppred> kubectl get pods
error: You must be logged in to the server (Unauthorized)
PS C:\Users\ppred>


Create user and run in admin cred

user.yaml
==========
apiVersion: v1
kind: ConfigMap
metadata:
  name: aws-auth
  namespace: kube-system
data:
  mapUsers: |
    - userarn: arn:aws:iam::533267064682:user/rbac-user
```

username: rbac-user

kubectl apply -f user.yaml (Run with root user)

Verify newly created user able to access or not

```
PS C:\Users\ppred> kubectl get pods
Error from server (Forbidden): pods is forbidden: User "rbac-user" cannot list resource "pods" in
API group "" in the namespace "default"
PS C:\Users\ppred>
```

Create role and rolebinding files and run with root

==========

role-user.yaml

==========

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: rbac-test
  name: pod-reader
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["list","get","watch"]
- apiGroups: ["extensions","apps"]
  resources: ["deployments"]
  verbs: ["get", "list", "watch"]
```

role-binding.yaml

==========

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: read-pods
```

```
  namespace: rbac-test
subjects:
- kind: User
  name: rbac-user
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io


PS C:\Users\ppred\demos\rbacdemo> kubectl apply -f .\role-user.yaml
role.rbac.authorization.k8s.io/pod-reader created
PS C:\Users\ppred\demos\rbacdemo> kubectl apply -f .\role-binding.yml
rolebinding.rbac.authorization.k8s.io/read-pods created
PS C:\Users\ppred\demos\rbacdemo>


Login to rbac-user and verify the pods
==================================
PS C:\Users\ppred> kubectl get pods -n rbac-test
NAME                     READY   STATUS    RESTARTS   AGE
nginx-676b6c5bbc-9z5z9   1/1     Running   0          22m
PS C:\Users\ppred>
```

Prepared by PPNREDDY .... 😊 😊 😊

EFK in Kubernetes stands for ***Elasticsearch, Fluentd, and Kibana*** a popular logging stack used for centralized log aggregation, processing, and visualization. It is widely used in Kubernetes environments to monitor and debug applications and infrastructure.

Components of EFK:

1. Elasticsearch

   Role: A distributed search and analytics engine used to store and index logs.

   Key Features:

   - Full-text search.
   - Scalable and distributed architecture.
   - Stores logs collected by Fluentd.

2. Fluentd

   Role: A log collector and forwarder. It gathers logs from Kubernetes nodes and applications, processes them, and sends them to Elasticsearch.

   Key Features:

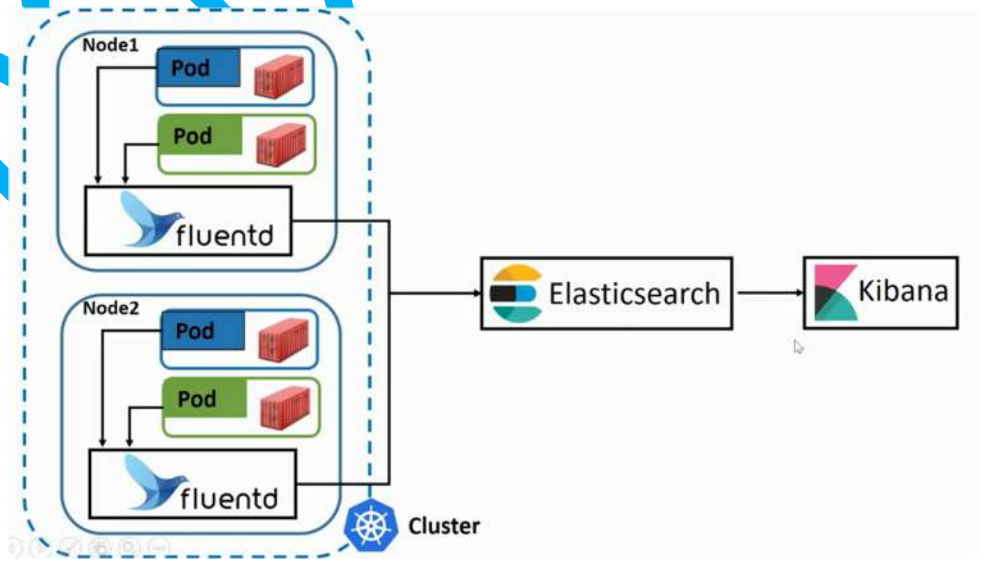
   - Lightweight and efficient.
   - Flexible plugins for various data sources and outputs.
   - Parses logs into structured formats for better analysis.

3. Kibana

   Role: A visualization and exploration tool for data stored in Elasticsearch.

   Key Features:

   - Visualize logs with dashboards.
   - Query logs using an intuitive UI.
   - Helps monitor and troubleshoot issues.



How EFK Works in Kubernetes

1. Log Sources:

   Kubernetes generates logs from:

   - **Nodes** (e.g., kubelet, container runtime).
   - **Pods** and their containers.

   Fluentd collects logs from these sources.

2. Log Processing:

   Fluentd processes raw logs:

   - Filters noise or irrelevant data.
   - Parses logs into JSON or other structured formats.
   - Tags logs for better indexing.

3. Log Storage:

   Processed logs are sent to Elasticsearch.

   Elasticsearch stores and indexes these logs for easy retrieval and analysis.

4. Log Visualization:

   Kibana connects to Elasticsearch.

   Users can create dashboards, query logs, and visualize data.

Why Use EFK in Kubernetes?

Centralized Logging: Consolidates logs from multiple sources into a single location.
Scalability: Handles logs from large-scale Kubernetes deployments.
Ease of Troubleshooting: Provides tools for searching and visualizing logs.
Custom Dashboards: Kibana enables customizable views for metrics and logs.

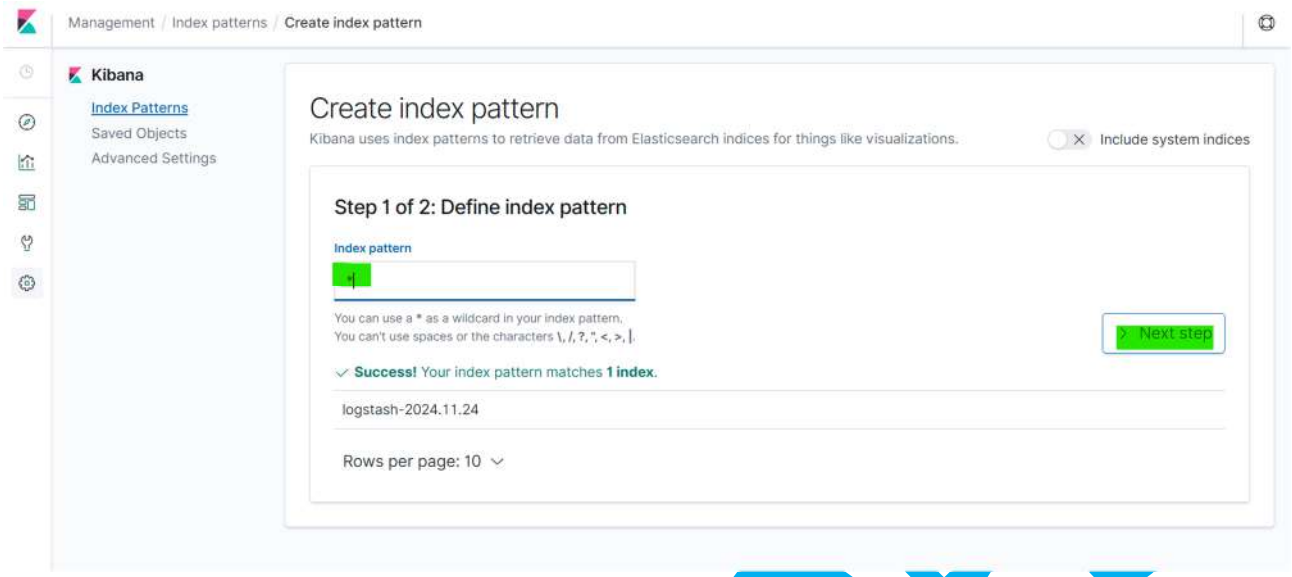
EFK manifest files try to clone below mentioned repo

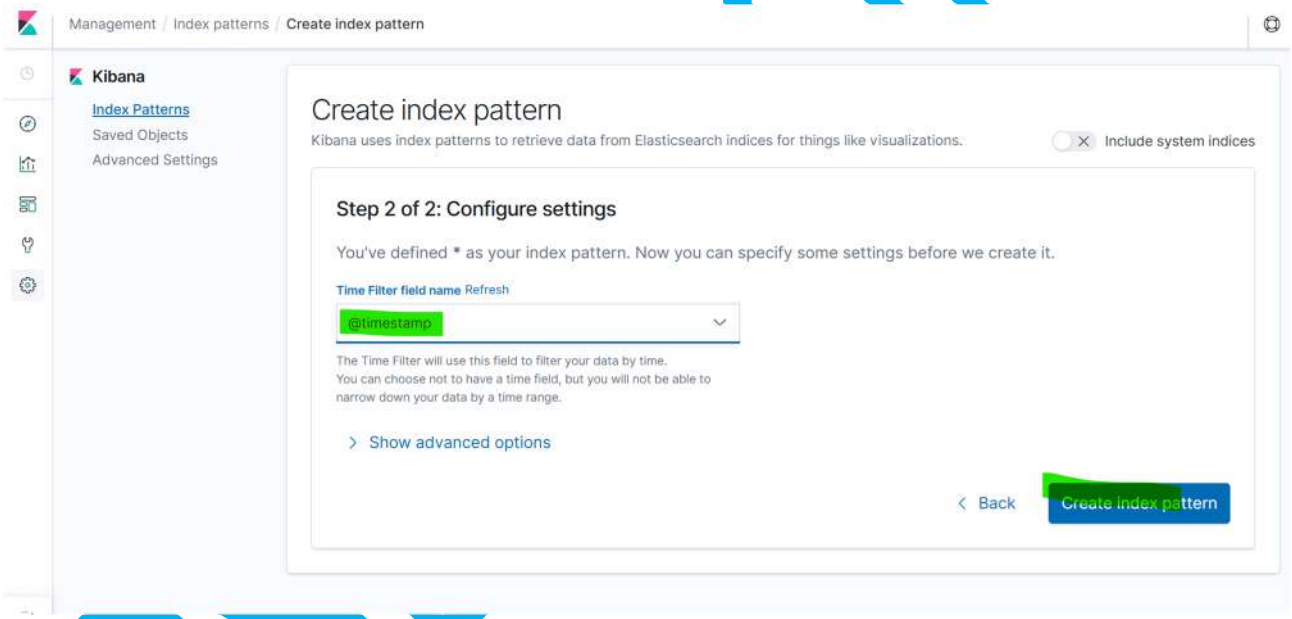
git clone https://github.com/psddevops/jenkins_pipelines.git

```
cd efk
ls -lrt
-rw-r--r-- 1 ppred 197609    69 Nov 24 11:27 Namespace.yml
-rw-r--r-- 1 ppred 197609 16597 Nov 24 11:27 Fluentd_ConfigMap.yaml
-rw-r--r-- 1 ppred 197609  2792 Nov 24 11:27 ElasticSearch_StatefulSet.yaml
-rw-r--r-- 1 ppred 197609   394 Nov 24 11:27 ElasticSearch_Service.yaml
-rw-r--r-- 1 ppred 197609  1179 Nov 24 11:27 Kibana_Deployment.yaml
-rw-r--r-- 1 ppred 197609  2466 Nov 24 11:27 Fluend_DaemonSet.yaml
-rw-r--r-- 1 ppred 197609   388 Nov 24 11:27 Kibana_Service.yaml
```

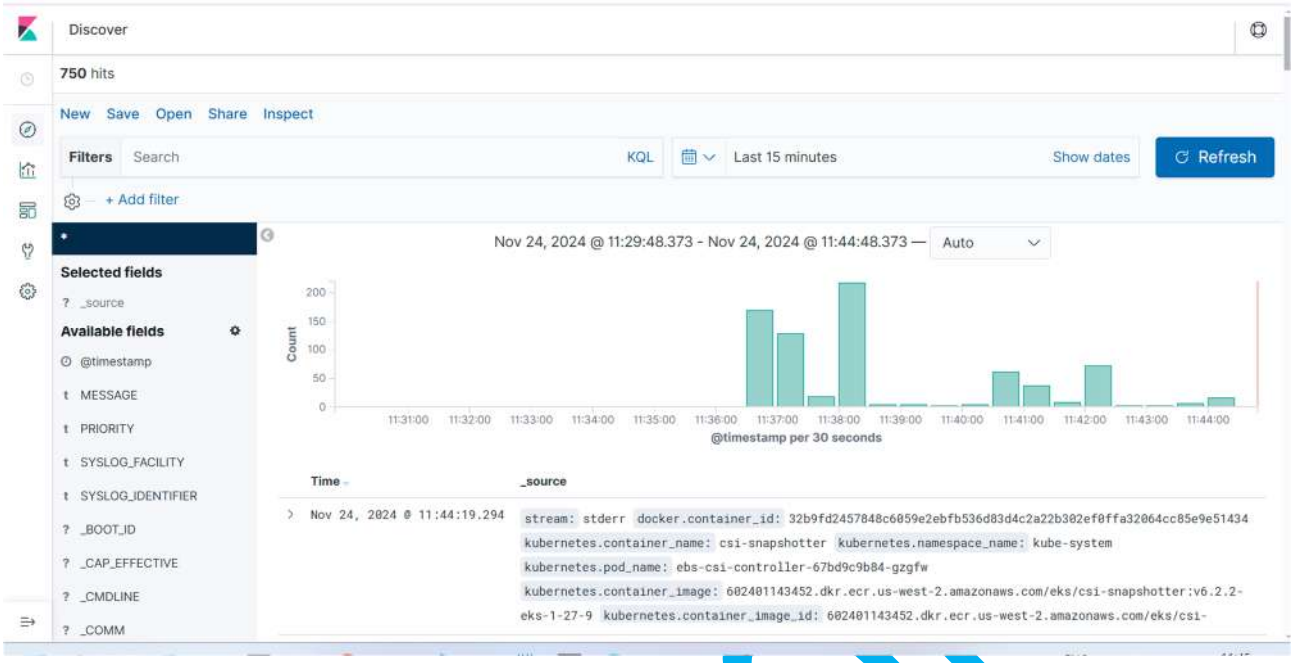
Prepared by PPNREDDY .... 😊 😊 😊

Kubectl apply -f .





Prepared by PPNREDDY .... 😊 😊 😊


Discover
750 hits
New Save Open Share Inspect
Filters Search KQL Last 15 minutes Show dates Refresh
+ Add filter
Selected fields
_source
Available fields
@timestamp
MESSAGE
PRIORITY
SYSLOG_FACILITY
SYSLOG_IDENTIFIER
_BOOT_ID
_CAP_EFFECTIVE
_CMDLINE
_COMM
Nov 24, 2024 @ 11:29:48.373 - Nov 24, 2024 @ 11:44:48.373 — Auto
Count
200
150
100
50
0
11:31:00 11:32:00 11:33:00 11:34:00 11:35:00 11:36:00 11:37:00 11:38:00 11:39:00 11:40:00 11:41:00 11:42:00 11:43:00 11:44:00
@timestamp per 30 seconds
Time
_source
Nov 24, 2024 @ 11:44:19.294
stream: stderr docker.container_id: 32b9fd2457848c6059e2ebfb536d83d4c2a22b302ef0ffa32064cc85e9e51434
kubernetes.container_name: csi-snapshotter kubernetes.namespace_name: kube-system
kubernetes.pod_name: ebs-csi-controller-67bd9c9b84-gzgfw
kubernetes.container_image: 602401143452.dkr.ecr.us-west-2.amazonaws.com/eks/csi-snapshotter:v6.2.2-
eks-1-27-9 kubernetes.container_image_id: 602401143452.dkr.ecr.us-west-2.amazonaws.com/eks/csi-