INTRODUCTION TO K8S

Kubernetes, also known as K8s, is an open-source container orchestration system. It automates deploying, scaling, and managing containerized applications. Kubernetes is used by companies of all sizes, from startups to large enterprises.

Kubernetes is a complex system, but it can be broken down into a few key concepts:

- **Pods**: A pod is the smallest unit of deployment in Kubernetes. A pod contains one or more containers, which are lightweight, isolated processes that share resources from a host machine.
- **Nodes**: A node is a physical or virtual machine that runs Kubernetes. Nodes are responsible for running pods and providing resources to them.
- **Clusters**: A cluster is a group of nodes that are managed by Kubernetes.
- **Services**: A service is a logical grouping of pods that are exposed to the outside world. Services can be used to expose pods to a network, or to load balance traffic between pods.

Kubernetes provides a number of features that make it a powerful tool for managing containerized applications. These features include:

- **Autoscaling**: Kubernetes can automatically scale pods up or down based on demand. This ensures that your applications have the resources they need without overprovisioning.
- **Health checks**: Kubernetes can automatically restart pods that are unhealthy. This ensures that your applications are always available.
- **Rollouts and rollbacks**: Kubernetes can automatically deploy new versions of your applications. This can be done in a rolling fashion, which means that new pods are deployed gradually, or in a blue-green fashion, which means that new pods are deployed alongside existing pods and then switched over when they are ready.
- **Logging and monitoring**: Kubernetes provide a number of tools for collecting logs and metrics from your applications. This data can be used to troubleshoot problems and to optimize your applications.

Kubernetes is a powerful tool for managing containerized applications. It is used by companies of all sizes, from startups to large enterprises. If you are looking for a way to deploy, scale, and manage containerized applications, Kubernetes is a great option.

Here are some of the benefits of using Kubernetes:

- **Scalability**: Kubernetes can easily scale your applications up or down based on demand. This can save you money on infrastructure costs.
- **Availability**: Kubernetes can automatically restart pods that are unhealthy, ensuring that your applications are always available.
- **Security**: Kubernetes provides a number of features that can help you secure your applications, such as role-based access control (RBAC) and network policies.
- **Ease of use**: Kubernetes has a large and active community that provides support and tools to help you get started.

If you are considering using Kubernetes, here are some of the things you should keep in mind:

- **Learning curve**: Kubernetes is a complex system, so it can take some time to learn how to use it effectively.
- **Cost**: Kubernetes can be more expensive than other container orchestration systems, such as Docker Swarm.
- **Complexity**: Kubernetes is a complex system, so it can be difficult to troubleshoot problems.

Overall, Kubernetes is a powerful and flexible container orchestration system that can help you manage your applications more effectively. However, it is important to be aware of the learning curve and the potential costs before you decide to use it.

KUBERNETES FEATURE

- **Container Orchestration**: Kubernetes automatically schedules containers on different nodes based on resource availability and constraints, ensuring optimal resource utilization.
- **Scaling and Load Balancing**: Kubernetes enables horizontal scaling by automatically replicating containers based on defined metrics and distributing traffic across them using load balancing techniques.
- **Service Discovery and Load Balancing**: Kubernetes provides a built-in service discovery mechanism that allows containers to find and communicate with each other using DNS or environment variables. Load balancing is also handled transparently by Kubernetes.
- **Self-Healing:** Kubernetes monitors the health of containers and automatically restarts failed containers or replaces them with new ones. It also provides advanced health checks and recovery mechanisms.
- **Rolling Updates and Rollbacks**: Kubernetes supports seamless updates of applications by gradually rolling out new versions while ensuring high availability. In case of issues, it also allows for easy rollbacks to a previous known state.
- **Storage Orchestration**: Kubernetes provides mechanisms to manage storage for containers, including persistent volumes and volume claims. This allows you to store and retrieve data from within your containers.
- **Secrets and Configuration Management**: Kubernetes offers built-in support for securely managing sensitive information such as passwords, API keys, and TLS certificates. It also provides mechanisms for managing configuration data for your applications.

WHERE K8S IS USEFUL?

Kubernetes is useful in a variety of scenarios and environments. Here are some common use cases where Kubernetes shines:

Containerized Application Deployment:

Kubernetes is primarily designed for deploying and managing containerized applications.

If you have an application that is packaged in containers, whether it's a monolithic application (components of an application are tightly coupled and packaged together as a single unit.) or a microservices-based architecture (collection of small, loosely coupled, and independently deployable services), Kubernetes provides an excellent platform for orchestrating and scaling those containers across a cluster of nodes.

Scalability and High Availability:

- Kubernetes excels in scenarios where scalability and high availability are crucial.
- It can automatically scale your application by adding or removing container replicas based on resource usage or other defined metrics.
- It ensures that your application remains highly available even in the event of container or node failures.

Microservices Architecture:

- Kubernetes is well-suited for managing microservices-based architectures.
- It allows you to deploy each microservice as a separate container and provides mechanisms for service discovery, load balancing, and communication between microservices.
- Kubernetes also enables independent scaling and versioning of individual microservices.

Multi-Cloud and Hybrid Cloud Environments:

- Kubernetes is cloud-agnostic and can run on various cloud providers such as AWS, Google Cloud, Azure, and on-premises data centers.
- This makes it suitable for organizations that operate in multi-cloud or hybrid cloud environments, allowing consistent deployment and management of applications across different infrastructure platforms.

Development and Testing Environments:

- Kubernetes provides a consistent environment for development and testing teams.
- Developers can run their applications locally in containers using tools like Minikube or Docker Desktop Kubernetes, which mimic the Kubernetes environment.
- This helps ensure that applications behave consistently in development, testing, and production environments.

CI/CD Pipelines:

- Kubernetes integrates well with CI/CD (Continuous Integration/Continuous Deployment) pipelines.
- It allows for automated application deployment, rolling updates, and rollbacks.
- Kubernetes can be seamlessly integrated with popular CI/CD tools like Jenkins, GitLab CI/CD, or Argo, enabling organizations to automate the deployment and delivery of their applications.

Big Data and AI/ML Workloads:

- Kubernetes can efficiently handle resource-intensive workloads such as Big Data processing and AI/ML (Artificial Intelligence/Machine Learning).
- By leveraging Kubernetes' ability to scale containers and distribute workloads across a cluster, organizations can run data-intensive applications and leverage distributed processing frameworks like Apache Spark or TensorFlow.

Internet of Things (IoT) Applications:

- Kubernetes can be utilized in IoT scenarios to manage and deploy containerized applications on edge devices or IoT gateways.
- It allows organizations to deploy and manage containers on resource-constrained devices efficiently, ensuring consistency and scalability in IoT deployments.

KUBERNETES BENEFITS

Scalability and Elasticity:

- Kubernetes enables horizontal scaling of applications by automatically distributing container replicas across multiple nodes.
- It can scale up or down based on workload demands, ensuring efficient resource utilization and the ability to handle increased traffic or processing requirements.

High Availability and Fault Tolerance:

- Kubernetes enhances application reliability by automatically monitoring and managing container health.
- It restarts or replaces failed containers, reschedules them on healthy nodes, and maintains the desired state of the application. This ensures high availability and minimizes downtime.

Automated Deployments and Rollbacks:

- Kubernetes simplifies the deployment process by allowing you to define your application's desired state in declarative configurations.
- It automates the deployment of containers, rolling updates, and rollbacks, reducing manual effort and minimizing the risk of errors during deployments.

Container Orchestration and Management:

- Kubernetes abstracts away the complexities of managing containerized applications.
- It handles container scheduling, resource allocation, networking, and service discovery, allowing developers to focus on application development rather than infrastructure management.

Service Discovery and Load Balancing:

- Kubernetes provides built-in service discovery mechanisms that allow containers to easily find and communicate with each other.
- It also offers load balancing for distributing traffic across containers, ensuring efficient utilization of resources and optimizing application performance.

Storage Orchestration:

- Kubernetes offers flexible and scalable storage options for containers.
- It provides mechanisms for managing persistent volumes, allowing data to persist even if containers are terminated or rescheduled.

Infrastructure Flexibility:

- Kubernetes is cloud-agnostic and can run on various cloud platforms (such as AWS, Google Cloud, Azure) as well as on-premises infrastructure.
- This flexibility allows organizations to adopt a multi-cloud or hybrid cloud strategy while maintaining consistency in application deployment and management.

Ecosystem and Community Support:

- Kubernetes has a vibrant ecosystem with a wide range of tools and integrations available.

- It is supported by a large and active community of developers, which ensures continuous improvement, bug fixes, and the availability of resources and expertise.

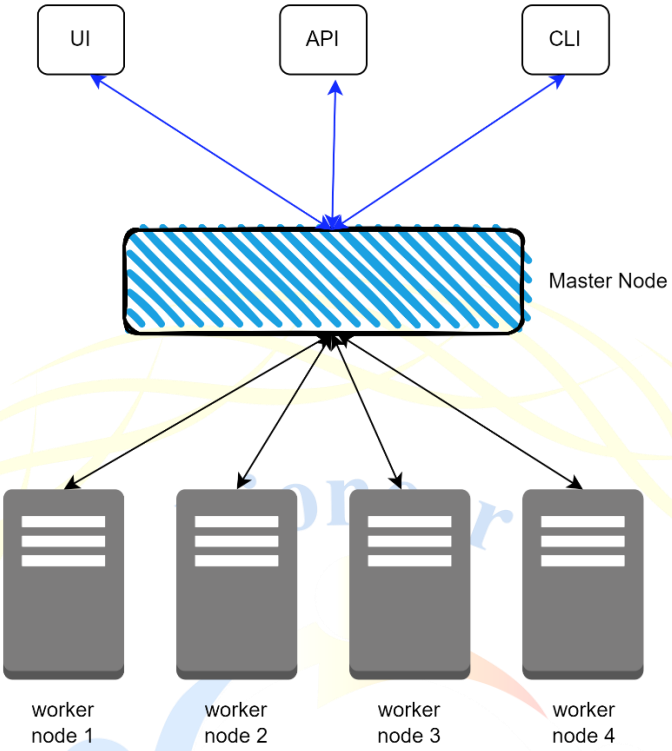
DevOps Enablement:

- Kubernetes aligns well with DevOps practices by providing the ability to automate application deployments, scaling, and management.
- It integrates with popular CI/CD tools, enabling organizations to establish efficient and automated development and deployment pipelines.

Cost Efficiency:

- Kubernetes optimizes resource utilization and allows efficient scaling, resulting in cost savings.
- It can dynamically allocate and deallocate resources based on demand, preventing overprovisioning and reducing infrastructure costs.

KUBERNETES ARCHITECTURE



A high-level K8s architecture has 4 components:

1. Master node or Control plane
2. Worker node(s)
3. Pods
4. Containers

Master Node: The master node is the control plane of the Kubernetes cluster. It coordinates and manages the cluster's operations. The master node typically runs the following components:

API Server:

- The API server exposes the Kubernetes API, which allows users and other components to interact with the cluster.
- It serves as the central control point for managing the cluster's resources and receiving and processing requests.

Scheduler:

- The scheduler is responsible for placing containers on worker nodes based on resource availability, constraints, and scheduling policies.
- It ensures that containers are distributed across the cluster efficiently.

Controller Manager:

- The controller manager runs various controllers that monitor the state of the cluster and ensure that the desired state is maintained.
- These controllers handle tasks such as scaling deployments, managing replication, handling node failures, and maintaining the overall health of the cluster.

etcd:

- etcd is a distributed key-value store used by Kubernetes to store the cluster's configuration data and state.
- It provides a reliable and highly available data store for the master node.
- Worker Nodes: Worker nodes, also known as minion nodes, are the machines where containers are scheduled and run.
- Runs the pods and containers (inside the pods)

Each worker node typically runs the following components:

Kubelet:

- The Kubelet is responsible for managing containers on a node.
- It interacts with the API server to receive instructions for creating, starting, stopping, and monitoring containers. It ensures that containers are running and healthy on the node.

Container Runtime:

- The container runtime is the software responsible for running containers, such as Docker or containerd.
- It provides the necessary environment to execute containerized applications.

kube-proxy:

- kube-proxy is responsible for network proxying and load balancing.
- It manages network connectivity and routing between services and containers, enabling communication within the cluster.

Networking:

- Kubernetes requires a networking solution to enable communication between containers running on different nodes.
- There are multiple networking options available, such as the Container Network Interface (CNI), which allows for different network plugins to be used, including overlay networks, software-defined networks, or bare-metal networks.

Add-ons: Kubernetes provides various add-ons and extensions that enhance its functionality. These include:

Dashboard - The Kubernetes Dashboard is a web-based user interface that provides a graphical representation of the cluster's resources and allows users to manage and monitor their applications.

Ingress Controller - An Ingress Controller manages inbound network traffic to services within the cluster and allows for the configuration of routing rules and load balancing for external access.

DNS - Kubernetes includes a DNS add-on that provides a DNS service within the cluster, enabling containers to communicate with each other using DNS names.

Metrics Server - The Metrics Server collects resource usage metrics from the cluster and makes them available to the Kubernetes API. It is used for auto-scaling and resource management.

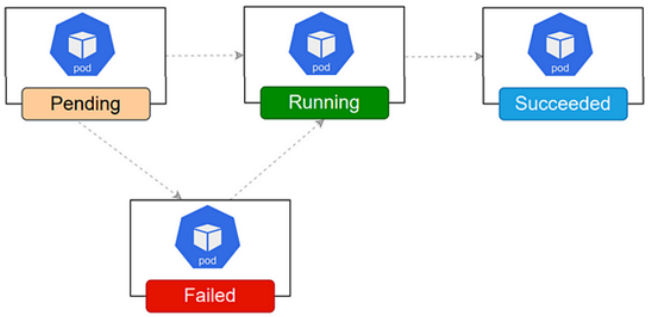
POD & ITS LIFE CYCLE

To keep it simple:

- Virtualization = Virtual Machines
- Docker = Containers
- Kubernetes = Pods

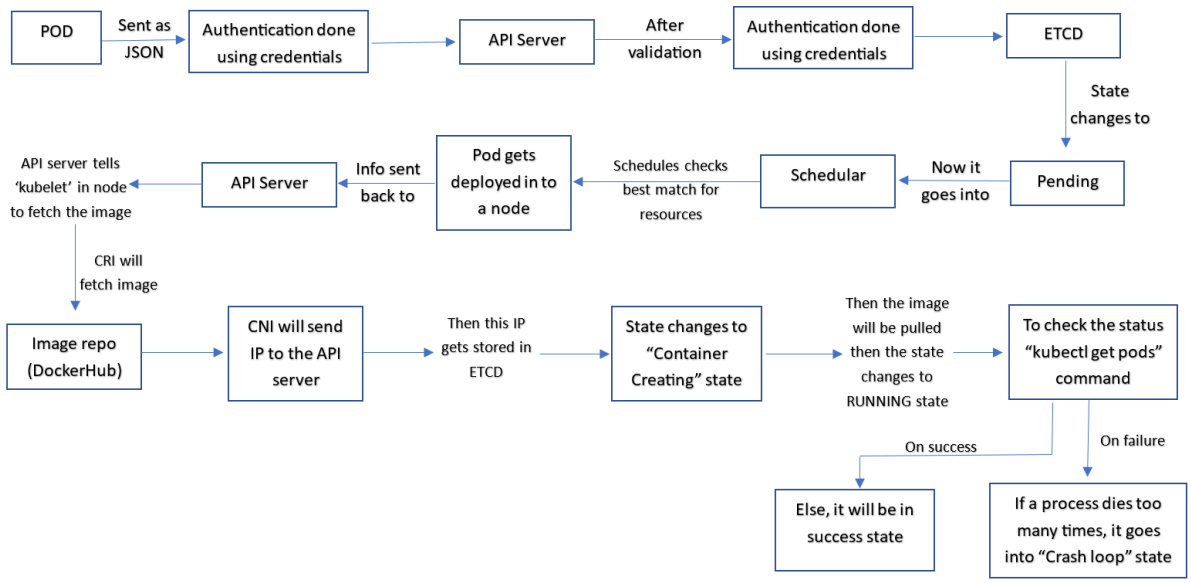
In simple terms,

- Pods are the workloads that runs on worker nodes
- A pod is the smallest and most fundamental unit of deployment.
- It represents a single instance of a running process in the cluster.
- A pod encapsulates one or more containers, storage resources, and network resources, and it is the basic building block of an application in Kubernetes.
- Pods play a vital role in Kubernetes, providing a logical unit for managing containers and enabling applications to be scheduled and deployed as a cohesive unit.
- They enable efficient resource sharing, easy inter-container communication, and encapsulation of application components within a single entity.



1. When a pod is created, it goes in PENDING state.
2. PENDING state means, your pod is accepted for the deployment but hasn't scheduled onto any of the VMs (nodes).
3. Schedular check for resources like CPU & RAM requirements and puts it into a specific VM within cluster.
4. Now POD status changes from PENDING to CREATING.
5. In CREATING state, image is getting pulled from centralized repository (in our case Docker Hub). If the image is already present locally, this pulling will be skipped.
6. Once image is pulled, container status changes from CREATING to RUNNING state.
7. If it fails to pull the image due to any reason, it will change to FAILED state.
8. RUNNING state means, now the program/application is running properly.
9. Now in case, if a service within fails or crashes due to any reason, schedular will restart the pod.
10. But if it crashes frequently, the container state changes to "CRASH LOOP BACK OFF" state. Here the pod tries to heal itself. But if it fails to do so, you need to start looking into commands like "kubectl get pods" or "kubectl describe pod <pod-name>" or even need to check logs.

POD LIFE CYCLE – DETAILED



this happens on MASTER node

1. pod created --> pending state --> deployment is accepted, but hasn't scheduled on any node.
2. "schedular" will check for the resources (CPU, Mem) & then it will be pushed to that worker node.

this happens on WORKER node

3. pod status changes from PENDING to CREATING
4. in this state, image will be checked (locally), else it will download the image from DockerHub.
5. once, image is pulled, container state changed from CREATING to RUNNING.
6. if it fails due to any reason, the state will change to FAILED.
7. RUNNING state means that the application is deployed.
8. if the container fails, K8 will restart the container/pod.
9. if the crash happens frequently, then state changes to "CRASHLOOP BACK OFF"
10. then pod tries to HEAL itself. if success -> OK, else – troubleshooting manually.
    - #kubectl get pods
    - #kubectl describe pod <pod-name>
    - #check logs

INSTALLING K8S ON CLOUD

Kubernetes Cluster Nodes (3):

- Cloud: Google Compute Engine (GCE)
- Master (1): 2 vCPUs - 4GB Ram
- Worker (2): 2 vCPUs - 2GB RAM
- OS: Ubuntu 18.04 or CentOS/RHEL 7

Firewall Rules (Ingress):

- Master Node: 2379,6443,10250,10251,10252
- Worker Node: 10250,30000-32767

For better visibility., Add below lines to ~/.bashrc

- Master Node:
  - PS1="\e[0;33m[\u@\h \W]\$ \e[m "
- Worker Node:
  - PS1="\e[0;36m[\u@\h \W]\$ \e[m "

Pre-requisites: Disable Swap

- # swapoff -a
- # sed -i.bak -r 's/(.+ swap .+)/#\1/' /etc/fstab

Bridge Traffic:

- lsmod | grep br_netfilter
- sudo modprobe br_netfilter
- lsmod | grep br_netfilter

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

Installing Docker (Run it on MASTER & WORKER Nodes)

```
# apt-get update -y
# apt-get install -y  apt-transport-https ca-certificates curl software-properties-common gnupg2
# curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
# sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
   $(lsb_release -cs) \
   stable"
```

Installing Docker

```
apt-get update && sudo apt-get install -y \
```

```
  containerd.io \
  docker-ce \
  docker-ce-cli
```

Setting up the Docker "daemon":

```
cat <<EOF | sudo tee /etc/docker/daemon.json
{
  "exec-opts": ["native.cgroupdriver=systemd"],
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "100m"
  },
  "storage-driver": "overlay2"
}
EOF

# mkdir -p /etc/systemd/system/docker.service.d
```

Start and enable docker

```
systemctl daemon-reload
systemctl enable docker
systemctl restart docker
systemctl status docker
```

Getting required package files (On Master & Worker node)

```
# apt-get update -y && sudo apt-get install -y apt-transport-https curl
# curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -
# cat <<EOF | sudo tee /etc/apt/sources.list.d/kubernetes.list
deb https://apt.kubernetes.io/ kubernetes-xenial main
EOF
```

Installing Kubeadm, Kubelet, Kubectl: (On Master & Worker node)

```
# apt-get update -y && apt-get install -y kubelet kubeadm kubectl
# apt-mark hold kubelet kubeadm kubectl
```

Start and enable Kubelet: (On Master & Worker node)

```
# systemctl daemon-reload
# systemctl enable kubelet
# systemctl restart kubelet
# systemctl status kubelet
```

Execute below commands to remove container config file:

rm /etc/containerd/config.toml

systemctl restart containerd

Initializing CONTROL-PLANE (Run it on MASTER Node only)

```
# kubeadm init
```

Commands to access kubectl

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

Joining the worker nodes to the master node in K8s cluster

```
# kubeadm join 10.190.0.2:6443 --token u81ks2.84wwdap851fqghd0 \
      --discovery-token-ca-cert-hash
sha256:a2f38616d810bfe21c4499b359dd7921b4faea8d5422b4f7d461925668fab503
```

If you lose joining command (run on master node)

```
# kubectl token create --print-join-command
```

Installing POD-NETWORK add-on (Run it on MASTER Node only)

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

After executing joining command on worker nodes, verify the joined nodes on master

```
# kubectl get nodes
# kubectl get nodes -o wide
```

Installing K8s on-prem

1. Turn off SELinux, Firewalld

2. Master node should have static IP.

3. update your repos (master, workers)
apt-get update -y

4. turn off/disable SWAP (master, workers) -- IMPORTANT STEP
        # swapoff -a
        # vim /etc/fstab

5. install openssh (master, workers)
        # apt-get install -y openssh-server
        # apt-get update -y

6. adding the kube repo
        # apt-get install -y apt-transport-https curl
        # curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -
        # cat <<EOF > /etc/apt/sources.list.d/kubernetes.list
        > deb http://apt.kubernetes.io kubernetes-xenial main
        > EOF
        # apt-get update -y

7. install docker, kubelet, kubectl, kubeadm (master, workers)
A . set up docker

1. sudo apt-get update -y
2. sudo apt-get install ca-certificates curl gnupg -y
3. sudo install -m 0755 -d /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
sudo chmod a+r /etc/apt/keyrings/docker.gpg
4. echo \
"deb [arch="$(dpkg --print-architecture)" signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu \
"$(. /etc/os-release && echo "$VERSION_CODENAME")" stable" | \
sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
5. sudo apt-get update -y
6. sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin -y

8. install kube commands:

apt-get install -y kubelet kubectl kubeadm

9. edit the config file:

vim /etc/systemd/system/kubelet.service.d/10-kubeadm.conf
& add this to the bottom of the file
Environment="cgroup-drive=systemd/cgroup-driver=cgroupfs"
:wq!

On master node:
---
10. Initialize k8s
kubeadm init

11. Create kube directory
mkdir -p $HOME/.kube

12. Copy config file to this dir.
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config

13. Provide the permissions
sudo chown $(id -u):$(id -g) $HOME/.kube/config

14. Check the nodes status
kubectl get nodes

15. List node information in detail.
kubectl get pods -o wide --all-namespace

On worker nodes:
16. join the cluster (workers) - using the output given by init command
kubeadm join ……

KUBERNETES NETWORK ADD-ONS

There are several networking plugins available for Kubernetes. Here is a comprehensive list of networking plugins commonly used in Kubernetes:

Flannel	Contiv	Nuage Networks
Calico	Project Calico with BGP	Kuryr
Weave	kube-iptables-tailer	NSX-T
Cilium	Multus CNI	Azure CNI
Antrea	SR-IOV	AWS VPC CNI
Canal (Combination of Calico and Flannel)	Macvlan	GKE VPC Network Peering
kube-router	IPvlan	CNI-Genie
Romana	OpenContrail	Weave Net Multicast
Kube-OVN	Gobetween	kube-bridge

Some commonly used pod networks are as follows:

Flannel:

- Flannel is a simple and lightweight network fabric designed for Kubernetes.
- It uses the VXLAN overlay network to create a virtual network connecting nodes.
- Flannel assigns a subnet to each node and ensures that containers on different nodes can communicate with each other.
- It is easy to set up and widely used in Kubernetes clusters.
- Suitable for small to medium-sized clusters and provides good performance.
- GitHub repository: coreos/flannel

Calico:

- Calico provides network policy enforcement and secure network connectivity for Kubernetes.
- It uses standard Linux networking components such as BGP routing and iptables to enforce policies and route traffic.
- Calico can scale to large clusters and supports advanced networking features.
- It offers network segmentation and isolation, allowing fine-grained control over network traffic between pods and nodes.
- Suitable for large-scale deployments and environments that require strong network policies.
- GitHub repository: projectcalico/calico

Weave:

- Weave provides a simple and flexible network overlay for Kubernetes.
- It creates a virtual network that connects pods across different hosts using VXLAN or UDP tunnels.
- Weave enables direct communication between pods without requiring external load balancers.
- It includes features like network encryption, DNS-based service discovery, and network segmentation.
- Suitable for small to medium-sized clusters and environments that require easy setup and flexible networking options.
- GitHub repository: weaveworks/weave

Cilium:

- Cilium is a networking and security plugin that provides API-aware network and security enforcement for Kubernetes.
- It uses eBPF (extended Berkeley Packet Filter) technology to enable fast packet processing and fine-grained control.
- Cilium offers observability, load balancing, network security policies, and transparent encryption.
- It is suitable for large-scale deployments and environments that require advanced security and observability features.
- GitHub repository: cilium/cilium

Antrea:

- Antrea is a Kubernetes networking plugin specifically designed for Kubernetes native networking.
- It uses Open vSwitch (OVS) as the data plane and supports Kubernetes Network Policy enforcement.
- Antrea leverages the OVS hardware acceleration capabilities for improved performance.
- It provides basic networking and security features while focusing on simplicity and ease of use.
- Suitable for small to medium-sized clusters and environments that prioritize simplicity and Kubernetes-native features.
- GitHub repository: vmware-tanzu/antrea

POD CONFIG FILE

To create a pod (recommended for dev/test environments)

```
# kubectl run --generator=run-pod/v1 nginx-pod --image=nginx
```

Note: do not run this command directly on production.

The suggested way to create & run a pod in production environment is via YAML config file.

Dummy code: # cat nginx-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  labels:
    app: nginx
    tier: dev
spec:
  containers:
  - name: nginx-container
    image: nginx
    env:
    - name: DEMO_GREETING
      value: "Hello from the environment"
    ports:
    - containerPort: 80
```

Details about the YAML file:

There are various sections of a YAML config file:

1. apiVersion

- It's a required field.
- Specifies K8s versioning used.
    - Ex: apiVersion: v1

2. Kind

- It's a required field.
- It defines the kind of object you want to create.
- Object like: Pod, Deployment, Services.
- Ex:-
    - Kind: Pod
    - Kind: Deployment

3. Metadata

- It's a required field.
- Contains metadata about the resources like name, label, annotations.
- This information manages & identifies resources within the cluster.

4. Spec

- It's a required field.
- Defines the desired state & configuration of the resources.
- Here you define containers, volume, images, ports, environment variables etc.

5. Container

- This comes under the "spec".
- Here we specify container name, image, env variables, ports & volumes.

6. Volume

- To define persistent storage within resources.
- This allows you to specifies desired storage volume.

7. Selector

- This comes under "spec".
- This is used for resources like services, replica set or deployment.

######## these are some of the sections of the config file #########

Expanding the YAML services:

Kind	API Version
Pod	v1
ReplicationController	v1
Service	v1
Secret	v1
ServiceAccount	v1
PersistentVolume	v1
PersistentVolumeClaim	v1
ConfigMap	v1
Namespace	v1
ComponentStatus	v1

Kind	API Version
ReplicaSet	apps/v1
Deployment	apps/v1
DaemonSet	apps/v1
Stateful	apps/v1

Kind	API Version
Role	rbac.authorization.k8s.io/v1
RoleBinding	rbac.authorization.k8s.io/v1
ClusterRole	rbac.authorization.k8s.io/v1
ClusterROleBinding	rbac.authorization.k8s.io/v1

Kind	API Version
Job	Batch/v1
Kind	API Version
CronJob	batch/v1beta1

PODS – TASKS

To create POD using YAML:

kubectl create -f <file>.yaml //creating object

#kubectl apply -f <file>.yaml //apply object

To display:

kubectl get pods <pod-name> //listing all the pods with status

kubectl get pods <pod-name> -o wide //listing pods in detailed format

kubectl get pods <pod-name> -o json //listing pods info in JSON format

kubectl get pods <pod-name> -o yaml //listing pods info in YAML format

To describe a pod/useful in troubleshooting

kubectl describe pods <pod-name>

To display a pod using label (if added to the YAML config file)

kubectl get pods --show-labels

kubectl get pods -l app=nginx // Print Pods with particular label

To edit a pod in a running state:

kubectl edit pods <pod-name>

To check logs:

kubectl logs pods <pod-name>

To delete a pod:

kubectl delete pods <pod-name>

REPLICASET

- A replicaset is a resource object that is used to ensure a specific no. of pod replicas are running & maintained within a cluster.
- It is one of the core controllers provided by K8s & is responsible for managing & scaling pods.
- ReplicaSet always ensures that a specified no. of pods replica are always available & running.
- If a pod fails or terminated, RS automatically replaces it with a new pod to maintain the desired count.
- Similarly, if a count is increased, RS creates additional pods to maintain the desired count.

Key features:

1. Specifying POD template
    a. A RS defines a template that specifies the desired config for the pods it manages.
    b. The template includes details like container, image, command, arguments, environment variables, etc.
2. Specifying replica count
    a. A RS "spec.replica" field specifies the desired no. of pods replicas to be maintained by RS.
3. Selector
    a. RS uses label selectors to identify the set of pods to manage.
4. Scalability
    a. RS support scaling UP or DOWN the no. of replicas by simply modification the "spec.replica" field.

If replicaset is not mentioned in the config file then default of "1" will be taken & it process the config. 2 major points about replicaset:

- Earlier we had "Replication Controller" (old version) which was replaced by "ReplicaSet" (new).
- We don't create replicas & pod manually.

- This deployment itself creates replicaset in the backend for us.
- RS then manages the no. of RS.

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: my-replicaset
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-container
        image: my-image:latest
        ports:
        - containerPort: 8080
```

ReplicaSet vs Replication Controller:

- apiVersion tag is missing in ReplicaSet (app/v1), not in Replication Controller (v1).
- A new "matchlabels" in present in ReplicaSet.

Applying the config

kubectl apply -f replicaset.yaml

Listing replicas

kubectl get replicasets my-replicaset

Display ReplicaSet (rs)

kubectl get rs

kubectl get rs <RS-NAME> -o wide

kubectl get rs <RS-NAME> -o yaml

kubectl get rs -l <LABEL>

To increase/decrease number of replicas, need to edit the config file.

vim replicaset.yaml

// change the replicaset number accordingly.

:wq!

OR by using below command (replicaset can be used as rs in short):

kubectl scale rs my-replicaset --replicas=3

kubectl get replicasets my-replicaset

Then apply the changes

kubectl apply -f replicaset.yaml

& verify

kubectl get replicasets my-replicaset

Print Details of ReplicaSet

kubectl describe rs <RS-NAME>

To check if replicaset is working or not

kubectl get pods

Scaling Applications

kubectl scale rs <RS-NAME> --replicas=[COUNT]

Editing ReplicaSet

kubectl edit rs <RS-NAME>

Then delete one pod from this list

kubectl delete pod my-replicaset-pw449

Check replicaset again

kubectl get replicasets my-replicaset //here status & desired will be as old config.

Check the pod status

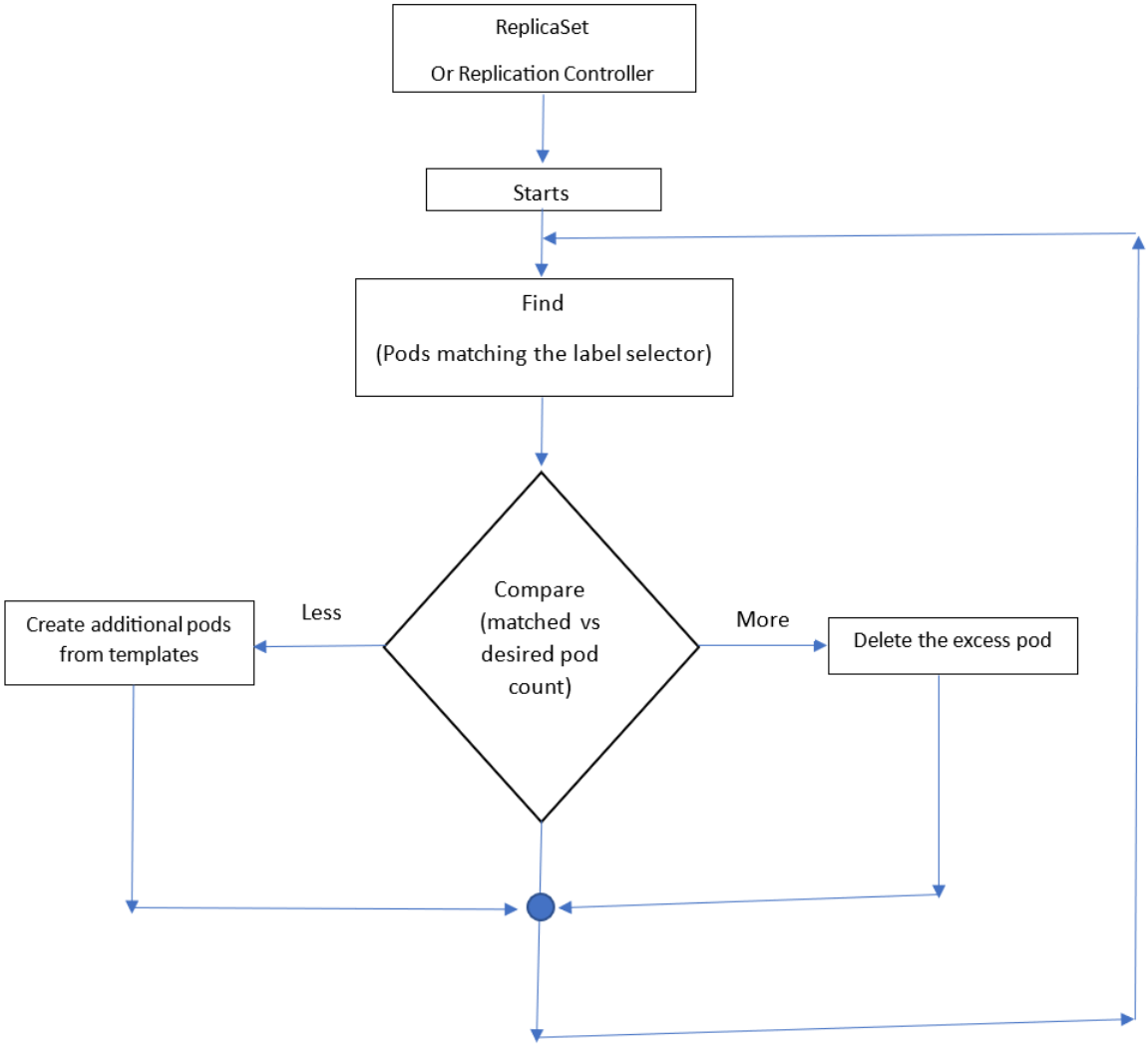
kubectl get pods

Deleting ReplicaSet

kubectl delete rs <RS-NAME>

REPLICATION LOOP

Replication controller & replicaset are based on replication loop. It's basically a loop.



REPLICASET YAML FILE

Simple replica set example:

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: my-replicaset
spec:
  replicas: 2
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-container
        image: nginx:1.16
        ports:
        - containerPort: 8080
```

Explanation:

apiVersion	API version of resource type
Kind:ReplicaSet	Resource type
Replicas	3
{key:tier, operator:In, values:[frontend]}	Here we are using set-based operator
template: metadata: name: my-pod labels: app: my-app tier: frontend spec: containers: - name: my-container image: nginx:latest ports: - containerPort: 80	POD template used for creating new pods.

REPLICASET – TASKS

Creating/Applying replica config file

kubectl create -f [file-name].yaml

kubectl apply -f [file-name].yaml

Displaying the details of replica set (replica set == rs)

kubectl get rs <replication-name>

kubectl get rs <replication-name> -o wide

kubectl get rs <replication-name> -o json

kubectl get rs <replication-name> -l [KEY=NAME]

Describing a replica set

kubectl describe rs <replication-name>

Scaling a replica set

kubectl scale rs <replication-name>--replicas=<COUNT>

Editing a replica set

kubectl edit rs <replication-name>

Deleting a replica set

kubectl delete rs <replication-name>

NAMESPACES

- It's a way of partitioning the entire K8s cluster into multiple virtual partitions.
- You can partition your K8s cluster into:
    - Various teams
    - Applications
    - Environments
    - Or by any other custom requirement.

By default, k8s install 2 namespaces

1. Kube-system namespace

This contains all the system related pods like:

- API-server,
- controller master,
- ETCD,
- schedular,
- Kube-proxy.

2. Default namespace

We create all the K8s objects under this & manage it.

To create a custom namespace

kubectl create namespace dev

kubectl create namespace prod

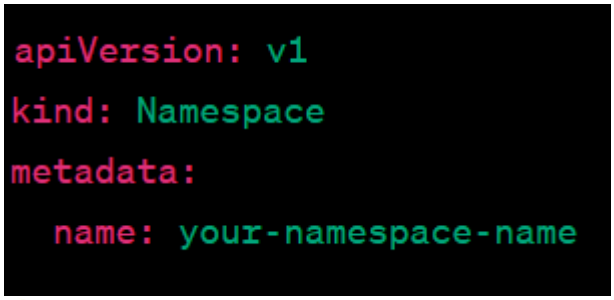
In these namespaces, we can create 2 resources with identical configurations & identical names, if required & it one does not affect other.

Note:- Nodes are shared across cluster. They are not restricted to a single namespace. Namespaces are highly effective between apps, teams & environments.

Namespace YAML config

YAML configuration file for creating a namespace in Kubernetes:

```
apiVersion: v1
kind: Namespace
metadata:
  name: your-namespace-name
```



To create a namespace using this configuration file

kubectl apply -f namespace.yaml

Instead, we can also use:

kubectl create namespace dev

kubectl create namespace qa

kubectl create namespace prod

To create a reason in a specific namespace:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    env: dev
  name: nginx-pod
  namespace: dev
spec:
  …
```

Namespace - needs to be added/applied on metadata of the config file.

To list the resources/pods running in a specific NS.

kubectl get pods --namespace=dev

NAMESPACE – TASKS

Listing all namespaces:

kubectl get namespaces

Creating namespace using file:

kubectl create -f filename.yaml

kubectl apply -f filename.yaml

Creating NS using cmd:

kubectl create namespace <ns-name>

Display namespace (ns):

kubectl get ns <ns-name>

kubectl get ns -o wide

kubectl get ns -o yaml

kubectl get pods --namespace=<name>

Describe:

kubectl describe ns <name>

Set as default namespace:

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

kubectl config view | grep namespace

Updating the namespace config:

kubectl edit ns <name>

Deleting namespace:

kubectl delete ns <name>

Command to create a pod in custom namespace:

kubectl run nginx --image=nginx --namespace=dev

Validate:

kubectl get pods

kubectl get pods -n dev

ETCD BACKUP & RESTORE

- ETCD is key-value database.
- It stores:
  - Current set of pods.
  - Deployment
  - Replicasets
  - ConfigMaps
  - Jobs
  - & all other configs as key-value pairs.

Backup approach:

1. Managed service
   a. GKE, AKS, EKS will take care of all backup & updates.
2. Self-managed service
   a. You have to take the backup on regular basis (manually).

Why need to take backup?

1. When there is a "disaster".
   a. Ex:- when someone accidently delete the namespace, where all your data resides
2. When you want to replicate the entire environment.
   b. Ex:- you want to replicate entire production env. to staging env.
3. When you want to migrate.
   c. Ex:- when you want to migrate one environment to another.

We will be using "etcdctl" tool, which allows us to

- Add new entries
- Delete entries
- Take snapshots of entire K8s cluster.
- Restore ETCD db.
- Using commands like:
  - Put, get, save, restore, status.

If it's not available, then download it from: https://github.com/etcd-io/etcd/releases

Pre-requisites: Installing "etcdctl" client:

Ensure if "etcdctl" command line tool is available by running below command.

```
export ETCDCTL_API=3
etcdctl version
```

If not available, then you can do the same by following below steps.

```
mv /tmp/etcd-download-test/etcdctl /usr/bin
ETCD_VER=v3.4.14
# Choose either URL
GOOGLE_URL=https://storage.googleapis.com/etcd
GITHUB_URL=https://github.com/etcd-io/etcd/releases/download
DOWNLOAD_URL=${GOOGLE_URL}
rm -f /tmp/etcd-${ETCD_VER}-linux-amd64.tar.gz
```

```
rm -rf /tmp/etcd-download-test && mkdir -p /tmp/etcd-download-test
curl -L ${DOWNLOAD_URL}/${ETCD_VER}/etcd-${ETCD_VER}-linux-amd64.tar.gz -o /tmp/etcd-${ETCD_VER}-linux-amd64.tar.gz
tar xzvf /tmp/etcd-${ETCD_VER}-linux-amd64.tar.gz -C /tmp/etcd-download-test --strip-components=1
rm -f /tmp/etcd-${ETCD_VER}-linux-amd64.tar.gz
mv /tmp/etcd-download-test/etcdctl /usr/bin
```

Reference: https://github.com/etcd-io/etcd/releases

Deploy sample Pod for testing:
kubectl run nginx-pod --image=nginx
Create a pod:
kubectl run nginx-pod --image=nginx
Validate :
kubectl get pods
To take snapshot:
ETCDCTL_API=3 etcdctl --endpoints=https://127.0.0.1:2379 \
--cacert=<trusted-ca-file> --cert=<cert-file> --key=<key-file> \
snapshot save <backup-file-location>

- **to find the location of CA file**
- # kubectl -n kube-system describe pod etcd-kmaster
  - ➔ --cacert=/etc/kubernetes/pki/etcd/ca.crt
- **To find cert file**
  - ➔ --cert=/etc/kubernetes/pki/etcd/server.crt
- **To find the key file**
  - ➔ --key=/etc/kubernetes/pki/etcd/server.key
- **backup location**
  - ➔ **/opt/snapshot-pre-boot.db**

ETCDCTL_API=3 etcdctl --endpoints=https://127.0.0.1:2379 \
--cacert=/etc/kubernetes/pki/etcd/ca.crt --cert=/etc/kubernetes/pki/etcd/server.crt --key=/etc/kubernetes/pki/etcd/server.key snapshot save /opt/snapshot-pre-boot.db
delete the pod:
kubectl delete pod nginx-pod
Now to restore:
ETCDCTL_API=3 etcdctl --data-dir=/var/lib/etcd2 snapshot restore /opt/snapshot-pre-boot.db
Now edit the etcd config file:
vim /etc/kubernetes/manifests/etcd.yaml
on line 79:
path: /var/lib/etcd2
:wq!
Wait for some time so that the config can link & check the get you deleted in the beginning.:
kubectl get pods
kubectl get pod nginx-pod

ROLE & ROLE-BINDING

Create a user "appuser"

```
# useradd appuser
```

Creating Kubernetes test User Account(appuser) (using x509 for testing RBAC)

```
# Generating Key
openssl genrsa -out appuser.key 2048

# Generaing Certificate Signing request (csr):
openssl req -new -key appuser.key -out appuser.csr -subj "/CN=appuser"

# Singing CSR using K8s Cluster "Certificate" and "Key"
openssl x509 -req -in appuser.csr \
-CA /etc/kubernetes/pki/ca.crt \
-CAkey /etc/kubernetes/pki/ca.key \
-CAcreateserial \
-out appuser.crt -days 300

# Adding user credentials to "kubeconfig" file
kubectl config set-credentials appuser  --client-certificate=appuser.crt --client-key=appuser.key

# Creating context for this user and associating it with our cluster:
kubectl config set-context appuser-context --cluster=kubernetes --user=appuser

# Displaying K8s Cluster Config
kubectl config view
```

Creating Namespaces and Pod for testing RBAC

Creating test Namespace:

```
# kubectl create ns dev-ns
```

Creating test Pod:

```
# kubectl run nginx-pod --image=nginx -n dev-ns
# kubectl get pods -n dev-ns
```

Test Before Deploying (gives error):

```
# kubectl get pods -n dev-ns --user=appuser
```

Creating a "Role" & "RoleBinding"

```
# cat Role.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: dev-ns
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
---
```

```
# RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
  namespace: dev-ns
subjects:
- kind: User
  name: appuser
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

Verify the role exists or not:

kubectl get role -n dev-ns

Apply the config

kubectl apply -f role.yaml

Verify the role again:

kubectl get role -n dev-ns

Creating Resources Imperatively (Commands)

```
# role
kubectl create role pod-reader --verb=get --verb=list --verb=watch --resource=pods --
namespace=dev-ns

# rolebinding
kubectl create rolebinding read-pods --role=pod-reader --user=appuser --namespace=dev-ns

run below command, that gave error earlier:
# kubectl get pods -n dev-ns --user=appuser
```

Display Role and RoleBinding

```
# role
kubectl get role -n dev-ns

# rolebinding
kubectl get rolebinding -n dev-ns

describing:
# kubectl describe role -n dev-ns
# kubectl describe rolebinding -n dev-ns
```

Testing RBAC

```
Pod Operations: get, list, watch - in "dev-ns" namespace:
kubectl auth can-i get pods -n dev-ns --user=appuser       // Yes
kubectl auth can-i list pods -n dev-ns --user=appuser      // Yes
```

```
kubectl auth can-i delete pods -n dev-ns --user=appuser    // No
kubectl auth can-i update pods -n dev-ns --user=appuser    // No
```

Pod Operations: get, list, watch - in "NON dev-ns" namespace

```
kubectl auth can-i get pods -n kube-system --user=appuser
kubectl auth can-i list pods -n kube-system --user=appuser
kubectl auth can-i watch pods -n kube-system --user=appuser

Shows error for below cmds:
kubectl get pods --user=appuser # queries default namespace
kubectl get pods -n kube-system --user=appuser
```

Creating Objects in "dev-ns" namespace

```
kubectl auth can-i create pods -n dev-ns --user=appuser
kubectl auth can-i create services -n dev-ns --user=appuser
kubectl auth can-i create deployments -n dev-ns --user=appuser

kubectl run redis-pod -n dev-ns --image=redis --user=appuser
kubectl create deploy redis-deploy -n dev-ns --image=redis --user=appuser
```

Clean up:

```
# kubectl config unset contexts.appuser-context
# kubectl config unset users.appuser

# kubectl config view

# kubectl get pod nginx-pod -n dev-ns --user=appuser
# kubectl get pods -n dev-ns --user=appuser

# kubectl delete role pod-reader -n dev-ns
# kubectl delete rolebinding read-pods -n dev-ns
```

ClusterRole & ClusterRoleBinding

Creating Kubernetes test User Account(appmonitor) (using x509 for testing RBAC)

```
# Generating Key
openssl genrsa -out appmonitor.key 2048

# Generaing Certificate Signing request (csr):
openssl req -new -key appmonitor.key -out appmonitor.csr -subj "/CN=appmonitor"

# Singing CSR using K8s Cluster "Certificate" and "Key"
openssl x509 -req -in appmonitor.csr \
-CA /etc/kubernetes/pki/ca.crt \
-CAkey /etc/kubernetes/pki/ca.key \
-CAcreateserial \
-out appmonitor.crt -days 300

# Adding user credentials to "kubeconfig" file
kubectl config set-credentials appmonitor  --client-certificate=appmonitor.crt --client-
key=appmonitor.key

# Creating context for this user and associating it with our cluster:
kubectl config set-context appmonitor-context --cluster=kubernetes --user=appmonitor

# Displaying K8s Cluster Config
kubectl config view
```

Creating Namespaces and Pod for testing RBAC

```
# kubectl create ns test-ns1
# kubectl create ns test-ns2
```

Creating test Pod

```
# kubectl run nginx-pod-default --image=nginx
# kubectl run redis-pod-ns1 --image=redis -n test-ns1
# kubectl run httpd-pod-ns2 --image=busybox -n test-ns2
```

Test Before Deploying (gives error):

```
# kubectl get pods --user=appmonitor
# kubectl get pods -n test-ns1 --user=appmonitor
# kubectl get pods -n test-ns2 --user=appmonitor
# kubectl get pods -n kube-system --user=appmonitor
# kubectl get pods -A --user=appmonitor
```

Creating a "ClusterRole" & "ClusterRoleBinding" (using YAML)

```
# cat clusterrole.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: clusterrole-monitoring
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
---
# ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: clusterrole-binding-monitoring
subjects:
- kind: User
  name: appmonitor
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: clusterrole-monitoring
  apiGroup: rbac.authorization.k8s.io


apply role
# kubectl apply -f clusterrole.yaml
```

Test Before Deploying (No error this time):

```
# kubectl get pods --user=appmonitor
# kubectl get pods -n test-ns1 --user=appmonitor
# kubectl get pods -n test-ns2 --user=appmonitor
# kubectl get pods -n kube-system --user=appmonitor
# kubectl get pods -A --user=appmonitor
```

Creating Resources Imperatively (Commands)

```
# Cluster-role
kubectl create clusterrole clusterrole-monitoring --verb=get,list,watch --resource=pods

# Cluster-rolebinding
kubectl create clusterrolebinding clusterrole-binding-monitoring --clusterrole=clusterrole-
monitoring --user=appmonitor
```

Display ClusterRole and ClusterRoleBinding

clusterrole
kubectl get clusterrole | grep clusterrole-monitoring

clusterrolebinding
kubectl get clusterrolebinding | grep clusterrole-binding-monitoring

kubectl describe clusterrole clusterrole-monitoring
kubectl describe clusterrolebinding clusterrole-binding-monitoring

Testing ClusterRole & ClusterRoleBinding

Pod Operations: get, list, watch - in "kube-system", "default", "test-ns1", and "test-ns2" namespaces:
kubectl auth can-i get pods -n kube-system --user=appmonitor
kubectl auth can-i get pods -n default --user=appmonitor
kubectl auth can-i get pods -n test-ns1 --user=appmonitor
kubectl auth can-i get pods -n test-ns2 --user=appmonitor

Creating Objects in "default" (or in any other) namespace: --> Shows NO

kubectl auth can-i create pods --user=appmonitor
kubectl auth can-i create services --user=appmonitor
kubectl auth can-i create deployments --user=appmonitor
kubectl run redis-pod --image=redis --user=appmonitor
kubectl create deploy redis-deploy --image=redis --user=appmonitor

Deleting Objects in "default" (or in any other) namespace: --> Shows NO

kubectl auth can-i delete pods --user=appmonitor
kubectl auth can-i delete services --user=appmonitor
kubectl auth can-i delete deployments --user=appmonitor
kubectl delete pods nginx-pod --user=appmonitor

Cleanup

Delete ClusterRole and ClusterRoleBinding:
kubectl delete clusterrole clusterrole-monitoring
kubectl delete clusterrolebinding clusterrole-binding-monitoring

Removing User and Context from Cluster Config
kubectl config unset users.appmonitor
kubectl config unset contexts.appmonitor-context

Ensure user "appmonitor" and its configuration is removed:
kubectl get pods --user=appmonitor
kubectl config view

Deleting Pods:
kubectl delete pod nginx-pod-default
kubectl delete pod redis-pod-ns1 -n test-ns1
kubectl delete pod httpd-pod-ns2 -n test-ns2

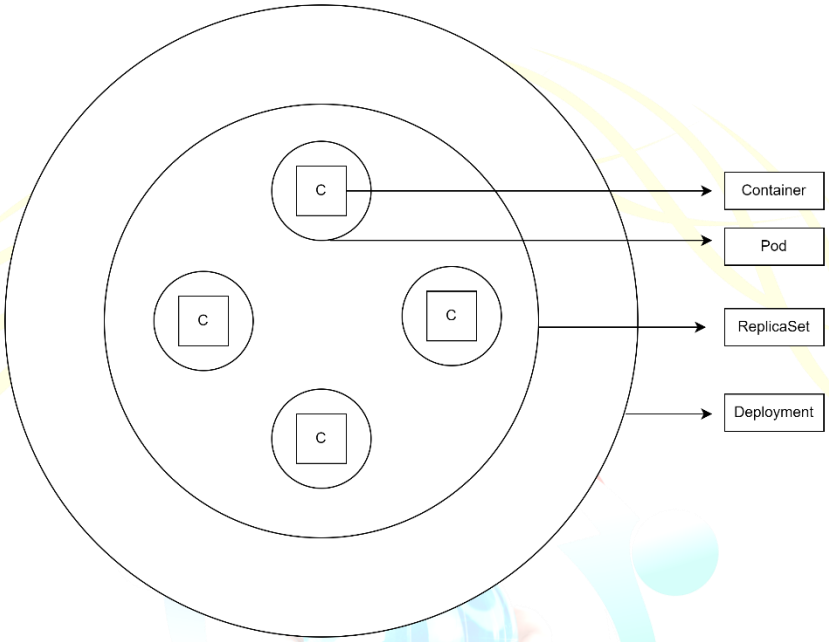
Deleting Namespace:
kubectl delete ns test-ns1
kubectl delete ns test-ns2

Validating:
kubectl get ns
kubectl get pods
kubectl get clusterrole | grep monitoring
kubectl get clusterrolebinding | grep monitoring

DEPLOYMENT

Whenever an app is deployed, there are few things to consider.

- Scaling of app.
    - Meaning, increasing & decreasing of instances as per configuration.
- Rollout & Rollback of version.
    - 'Deployment' will take care of rollout & rollback.



Deployment features:

Deployment supports following features under 2 categories.

Application deployment	Rollout
	Pause & Resume
	Rollback
Scaling	Replica
	Scale up & scale down

Deployment strategies:

1. Recreate
2. Rolling updates
3. Canary deployments
4. Blue-Green upgrade strategy

Recreate

- Aka dummy deployment
- We will shut down all deployment of "V1" & 2 more "V2" while upgrading V1 to V2.
- Here, at a time 2 pods will not be available (out of 4).
- Simple to use.
- Downtime - while migrating v1 to v2.
- This is ideal for DEV, QA environments.

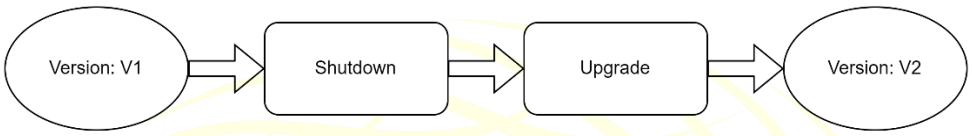


Figure: Recreate deployment strategy

Rolling updates

- K8s slowly rollout updates to all the instances one by one. It's a time-consuming process.
- This is a default strategy in K8s.

Working:

- When a pod goes under upgradation (Out of 4 pods), one pod with V1 is terminated & traffic get stopped towards this pod. Then a new pod with V2 will be spin & take its place.
- Meanwhile remaining V1 pods receives updates. Once this V2 pod is ready, the same procedure is applied to other 3 pods. When V2 is ready K8s will start sending the traffic to this V2. This will continue until all pods are replaced by V2 pods.

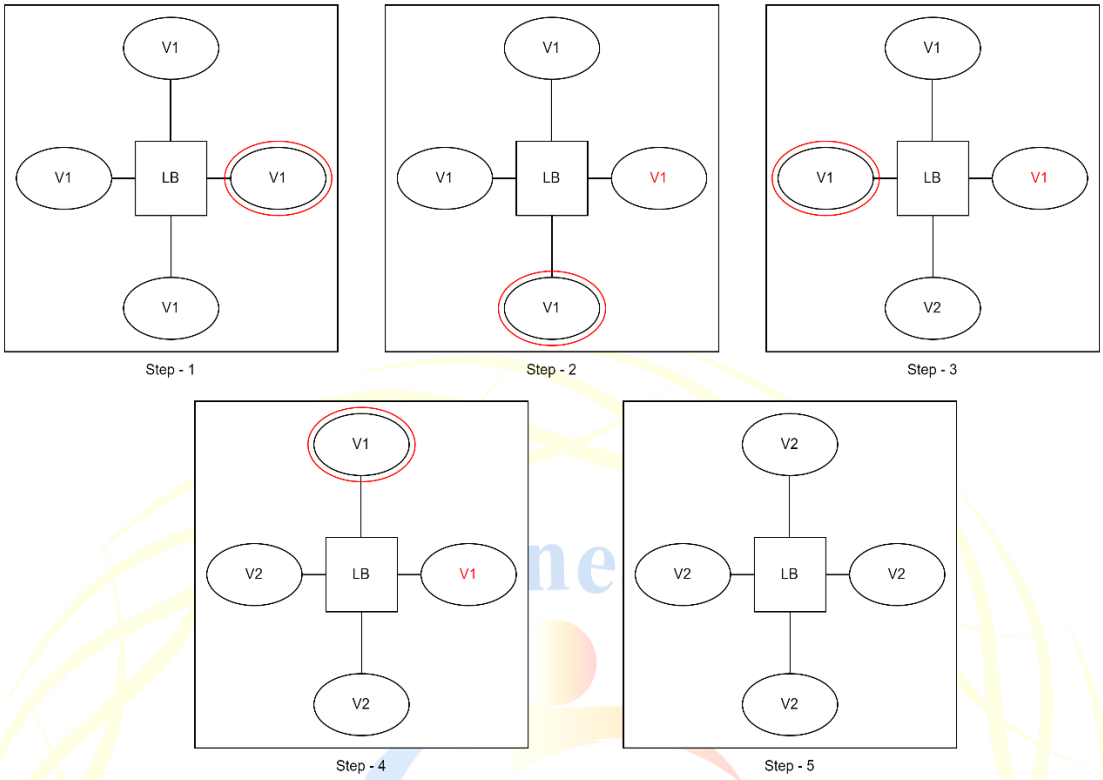


Figure: Rolling Update deployment strategy

Canary deployment

- We use this to test new version before it is rolled out.
- Users can rollout if there are issues with the new version.
- It is a slow rollout process.

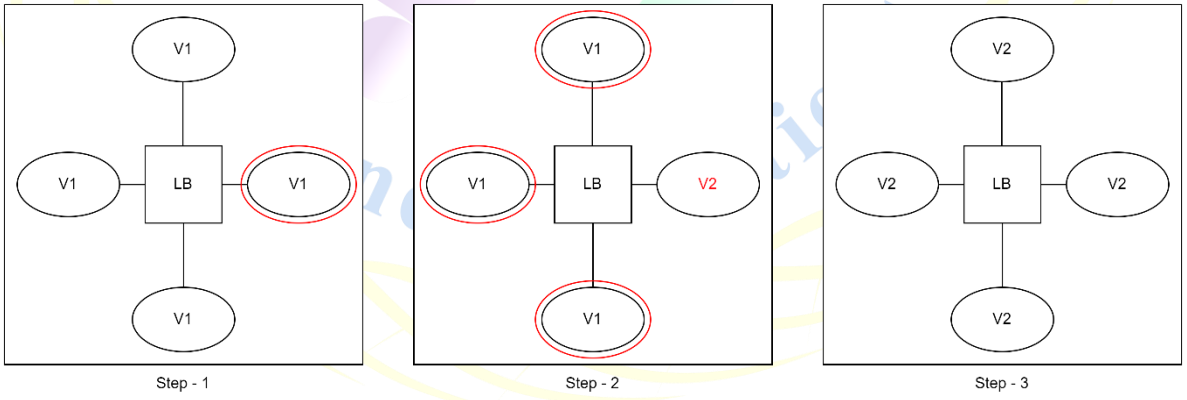


Figure: Canary deployment strategy

Blue/Green upgrade strategy

- We deploy same no. of never version instances along with the older version.
- Here goal is to switch the traffic from older version to newer version.
- In case if something is wrong with newer version, it is easy to roll back older version, else we remove older version.
- It is very expensive as it requires double resources.

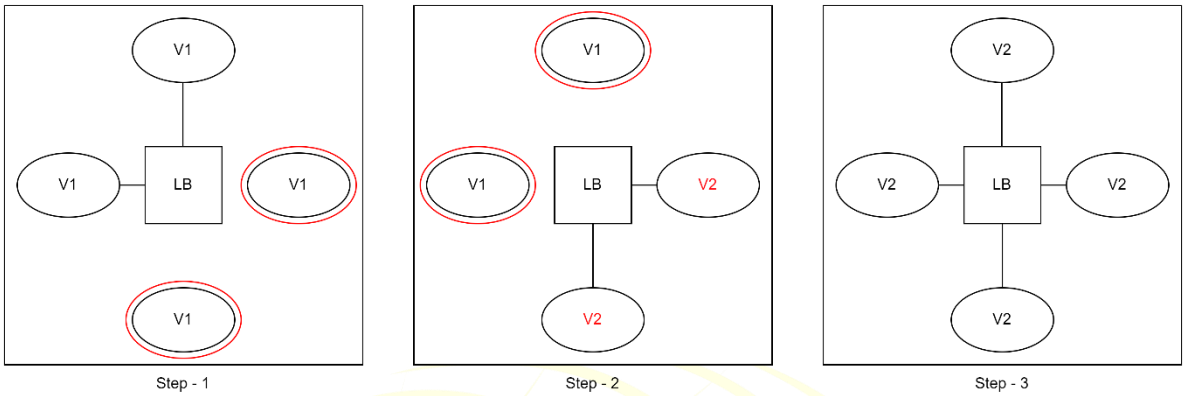


Figure: Blue-Green deployment strategy

Command to check the deployment strategy using kubectl command:

kubectl describe deploy nginx | grep StrategyType

DEPLOYMENT – YAML

Creating Deployment Declaratively (Using YAML file)

```
# cat nginx-deploy.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deploy
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx-app
  template:
    metadata:
      name: nginx-pod
      labels:
        app: nginx-app
    spec:
      containers:
      - name: nginx-container
        image: nginx:1.18
        ports:
        - containerPort: 80
```

Deployment of YAML:

```
# kubectl apply -f nginx-deploy.yaml
```

Creating Deployment "Imperatively" (from command line):

kubectl create deployment NAME --image=[IMAGE-NAME] --replicas=[NUMBER]

Displaying Deployment

kubectl get deploy <NAME>

kubectl get deploy <NAME> -o wide

kubectl get deploy <NAME> -o yaml

Describing Deployment

kubectl describe deploy <NAME>

Print Details of Pod Created by this Deployment

kubectl get pods --show-labels

kubectl get pods -l [LABEL]

EX: # kubectl get pods -l app=nginx-app

Scaling Applications:

kubectl scale deploy [DEPLOYMENT-NAME] --replicas=[COUNT]

Update the replica-count to 5

Edit the Deployment:

kubectl edit deploy [DEPLOYMENT-NAME]

Running operations directly on the YAML file:

SYNTAX: # kubectl [OPERATION] –f [FILE-NAME.yaml]

kubectl get –f [FILE-NAME.yaml]

kubectl describe –f [FILE-NAME.yaml]

kubectl edit –f [FILE-NAME.yaml]

kubectl delete –f [FILE-NAME.yaml]

kubectl create –f [FILE-NAME.yaml]

Delete the Deployment:

kubectl delete deploy <NAME>

kubectl get deploy

kubectl get rs

kubectl get pods

ROLLOUT & ROLLBACK UPGRADES

Creating Deployment "Imperatively" (from command line):

kubectl create deployment NAME --image=[IMAGE-NAME] --replicas=[NUMBER]

EX:- # kubectl create deployment nginx-deploy --image=nginx:1.18 --replicas=4

Upgrading Deployment with new Image:

kubectl set image deploy [DEPLOYMENT-NAME] [CONTAINER-NAME]=[CONTAINER-IMAGE]:[TAG] --record

EX: - # kubectl set image deploy nginx-deploy nginx=nginx:1.91 --record

Checking Rollout Status:

kubectl rollout status deploy [DEPLOYMENT-NAME]

EX - # kubectl rollout status deploy nginx-deploy

Waiting for deployment "nginx-deploy" rollout to finish: 2 out of 4 new replicas have been updated…

NOTE: There is some issue. To dig deep, let's check rollout history.

Checking Rollout History:

kubectl rollout history deploy [DEPLOYMENT-NAME]

EX -

```
root@master:# # kubectl rollout history deploy nginx-deploy
deployment.apps/nginx-deploy
REVISION  CHANGE-CAUSE
1         # kubectl set image deploy nginx-deploy nginx-container=nginx:1.91 --record=true
2         # kubectl set image deploy nginx-deploy nginx=nginx:1.91 --record=true
```

NOTE: From the output, you can see the commands that are run previously. If you notice, Image tag we used is 1.91 instead of 1.19. Let's rollback!

You can confirm the same from by running

kubectl get deploy nginx-deploy -o wide

Doing previous rollout "undo":

- # kubectl rollout undo deployment/[DEPLOYMENT-NAME] (OR)
- # kubectl rollout undo deployment [DEPLOYMENT-NAME] --to-revision=[DESIRED-REVISION-NUMBER]
- # kubectl rollout status deployment/[DEPLOYMENT-NAME]
- # kubectl get deploy [DEPLOYMENT-NAME] -o wide

Doing Rollout with correct Image version:

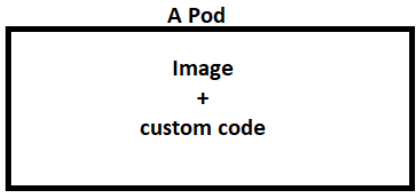
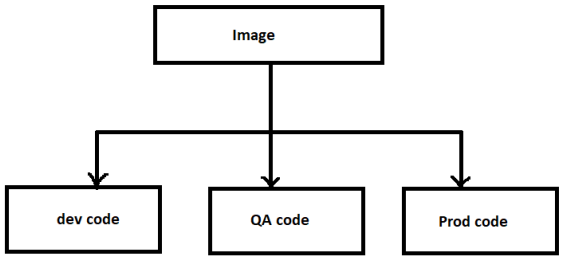
kubectl set image deploy [DEPLOYMENT-NAME] [CONTAINER-NAME]=[CONTAINER-IMAGE]:[TAG] --record

kubectl rollout status deploy [DEPLOYMENT-NAME]

kubectl get deploy [DEPLOYMENT-NAME] -o wide

CONFIGMAPS

- It is an API-object to store custom configuration data outside of your pod-object.
- This data should be "non-sensitive data".
- Sensitive data like certs, usernames, passwords shouldn't be part of ConfigMaps because this data on ConfigMaps will be shared with other teams.
- Assume you want to run containers in different environments with the same image but with different configurations inside the image.

Generally	For different environments:
	

ConfigMaps can allow you to decouple env-specific configurations from your contain image. So that your apps are easily portable.

ConfigMaps can mount data to the pods & containers using:

1. Key-value pair.
2. File with the data in it.

ConfigMaps are used as key-value pairs.

Key1:Value1

Key2:Value2

ConfigMaps as file

Key: nginx.conf

value:

http{

…

}

These ConfigMaps could be the used as

1. Environment variables
2. Arguments
3. Files in volumes

CONFIGMAPS – TASKS

Creating ConfigMaps Declaratively (Using YAML file)

Example-1:
--------- 	
apiVersion: v1
kind: ConfigMap
metadata:
  name: env-config-yaml
data:
  ENV_ONE: "va1ue1"
  ENV_TWO: "va1ue2"

Creating ConfigMap Imperatively (from Command line):

kubectl create configmap <NAME> <SOURCE>

From Literal value:

kubectl create configmap envconfigcmd fromliteral=ENV_ONE=value1 fromliteral=ENV_TWO=value2

From File:

kubectl create configmap myngixconfigfilecmd fromfile=/path/to/configmapfile.txt

kubectl create configmap myconfig fromfile=path/to/bar

Displaying ConfigMap:

kubectl get configmap <NAME>

kubectl get configmap <NAME> o wide

kubectl get configmap <NAME> o yaml

kubectl get configmap <NAME> o json

kubectl describe configmap <NAME>

Editing ConfigMap:

kubectl edit configmap <NAME>

Injecting ConfigMap into Pod As Environment Variables:

```
# cat cm-pod-env.yaml
apiVersion: v1
kind: Pod
metadata:
  name: cm-pod-env
spec:
  containers:
    - name: test-container
      image: nginx
      env:
      - name: ENV_VARIABLE_1
        valueFrom:
          configMapKeyRef:
            name: env-config-yaml
            key: ENV_ONE
      - name: ENV_VARIABLE_2
        valueFrom:
          configMapKeyRef:
            name: env-config-yaml
            key: ENV_TWO
  restartPolicy: Never
```

Deploy:

```
-------
kubectl apply -f cm-pod-env.yaml
```

Validate:

```
---------
kubectl exec cm-pod-env -- env | grep ENV
```

Injecting ConfigMap into Pod As Arguments(2/2)

```
# cat cm-pod-arg.yaml
apiVersion: v1
kind: Pod
metadata:
  name: cm-pod-arg
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      command: [ "/bin/sh", "-c", "echo $(ENV_VARIABLE_1) and $(ENV_VARIABLE_2)" ]
      env:
      - name: ENV_VARIABLE_1
        valueFrom:
          configMapKeyRef:
            name: env-config-yaml
            key: ENV_ONE
      - name: ENV_VARIABLE_2
        valueFrom:
```

```
        configMapKeyRef:
          name: env-config-yaml
          key: ENV_TWO
  restartPolicy: Never
```

Deploy:

```
---------
kubectl apply -f cm-pod-arg.yaml
```

Validate:

```
---------
kubectl logs cm-pod-arg
```

Injecting ConfigMap into As Files inside Volume(3/3)

```
# cat cm-pod-file-vol.yaml
apiVersion: v1
kind: Pod
metadata:
  name: cm-pod-file-vol
spec:
  volumes:
    - name: mapvol
      configMap:
        name: my-nginx-config-yaml
  containers:
    - name: test-container
      image: nginx
      volumeMounts:
      - name: mapvol
        mountPath: /etc/config
  restartPolicy: Never
```

Deploy:

```
-------
kubectl apply -f cm-pod-file-vol.yaml
```

Validate:

```
----------
kubectl exec configmap-vol-pod -- ls /etc/config
kubectl exec configmap-vol-pod -- cat /etc/config/etc/config/my-nginx-config.conf
```

Running operations directly on the YAML file:

- # kubectl [OPERATION] –f [FILE-NAME.yaml]
- # kubectl get –f [FILE-NAME.yaml]
- # kubectl delete –f [FILE-NAME.yaml]
- # kubectl get -f [FILE-NAME.yaml]
- # kubectl create -f [FILE-NAME.yaml]

Delete ConfigMap:

kubectl delete configmap <NAME>

SECRETS

In Kubernetes (K8s), secrets are objects used to store sensitive information, such as passwords, tokens, or other confidential data. They are designed to provide a secure and convenient way to manage and distribute sensitive information to containers running within a Kubernetes cluster.

Secrets in Kubernetes are typically created by the cluster administrator or developers responsible for deploying applications. They are stored within the cluster's etcd database, which is encrypted at rest. Secrets can be accessed by pods (containers) running within the same namespace as the secret or other authorized entities.

Types of secrets in K8s:

- **Generic**: Key-value pairs of arbitrary secrets.
- **Docker-registry**: Authentication information for accessing Docker registries.
- **TLS**: Certificates and private keys for secure communication.
- **SSH**: SSH keys used for authentication.
- **Service Account**: Automatically created secrets tied to service accounts.

SECRETS – TASKS

Creating Secrets using imperative method:	
echo -n 'admin' | base64	// copy its output
echo -n 'pas@word1' | base64	// copy its output

Using Base64 Encoding in creating Secre

# cat db-user-pass.yaml 	
apiVersion: v1
kind: Secret
metadata:
  name: db-user-pass
  namespace: default
data:
  username: <o/p of admin>
  password: <o/p of pwd>
 	
Deploy:
# kubectl apply –f secret-db-user-pass.yaml 	

Creating Secrets using imperative method:
kubectl create secret generic test-secret --from-literal='username=my-app' --from-literal='password=39528$vdg7Jb'

Creating Secrets Imperatively (From files):
echo -n 'admin' > ./username.txt echo -n '1f2d1e2e67df' > ./password.txt # kubectl create secret generic db-user-pass-from-file --from-file=./username.txt --from-file=./password.txt Example: # kubectl get secrets db-user-pass –o yaml

Injecting "Secrets" into Pod As Environmental Variables

```
# cat my-secrets-pod-env.yaml

apiVersion: v1
kind: Pod
metadata:
  name: secret-env-pod
spec:
  containers:
  - name: mycontainer
    image: redis
    env:
      - name: SECRET_USERNAME
        valueFrom:
          secretKeyRef:
            name: db-user-pass
            key: username
      - name: SECRET_PASSWORD
        valueFrom:
          secretKeyRef:
            name: db-user-pass
            key: password
  restartPolicy: Never
```

Validate:

```
# kubectl exec secret-env-pod -- env
# kubectl exec secret-env-pod -- env | grep SECRET
```

Injecting "Secrets" into Pod As Files inside the Volume:

```
# cat my-secrets-vol-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: secret-vol-pod
spec:
  containers:
    - name: test-container
      image: nginx
      volumeMounts:
        - name: secret-volume
          mountPath: /etc/secret-volume
  volumes:
    - name: secret-volume
      secret:
        secretName: test-secret
```

Validate:

```
# kubectl exec secret-vol-pod -- ls /etc/secret-volume
# kubectl exec secret-vol-pod -- cat /etc/secret-volume/username
# kubectl exec secret-vol-pod -- cat /etc/secret-volume/password
```

Displaying Secret:
kubectl get secret <NAME>
kubectl get secret <NAME> -o wide
kubectl get secret <NAME> -o yaml
kubectl get secret <NAME> -o json
kubectl describe secret <NAME>

Running operations directly on the YAML file:
kubectl [OPERATION] –f [FILE-NAME.yaml]
kubectl get –f [FILE-NAME.yaml]
kubectl delete –f [FILE-NAME.yaml]
kubectl get -f [FILE-NAME.yaml]
kubectl create -f [FILE-NAME.yaml]

Delete Secret:
kubectl delete secret <NAME>

NODESELECTOR

NodeSelector is a feature that allows you to schedule pods onto specific nodes in a cluster based on node labels. NodeSelector is used to define the requirements or preferences for pod placement, ensuring that pods are scheduled on nodes that meet certain criteria.

Each node in a Kubernetes cluster can be labeled with key-value pairs to identify their attributes, such as hardware specifications, availability zones, or other custom characteristics. NodeSelector enables you to specify label-based constraints when deploying pods, ensuring that they are placed only on nodes matching the specified criteria.

Labeling Node

```
# kubectl get nodes --show-labels
# kubectl label nodes worker-1 disktype=ssd
# kubectl get nodes --show-labels
# kubectl get pods -o wide
```

Deploying Node-Selector YAML

```
# cat nodeSelector-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nodeselector-pod
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  nodeSelector:
    disktype: ssd

Apply:
# kubectl apply -f ns.yaml
```

Testing

```
# kubectl get pods -o wide
# kubectl get nodes --show-labels

Let's Delete and Deploy "again" to ensure Pod is deployed on the same node which is lablelled above.
# kubectl delete -f ns.yaml
# kubectl apply -f ns.yaml
# kubectl get pods -o wide
# kubectl get nodes --show-labels
```

Cleanup

```
# kubectl label nodes worker-1 disktype-
```

```
# kubectl delete pods nodeselector-pod
```

METRICS SERVER

Kubernetes Metrics Server is a component of Kubernetes that collects resource usage metrics from Nodes and exposes them in the Kubernetes API server through the Metrics API for use by Horizontal Pod Autoscaler (HPA) and Vertical Pod Autoscaler (VPA). The metrics are meant for point-in-time analysis and aren't an accurate source for historical analysis. They can't be used as a monitoring solution or for other non-autoscaling purposes.

Metrics Server is not meant for non-autoscaling purposes. For example, don't use it to forward metrics to monitoring solutions, or as a source of monitoring solution metrics.

Here are some of the benefits of using Metrics Server:

- Scalability: Metrics Server can scale horizontally to meet the demands of your cluster.
- Availability: Metrics Server is designed to be highly available, with a built-in leader election mechanism.
- Performance: Metrics Server is designed to be efficient, with low overhead.
- Ease of use: Metrics Server is easy to deploy and manage.

Here are some of the requirements for using Metrics Server:

- Kubernetes 1.16 or later: Metrics Server is not supported on older versions of Kubernetes.
- Docker: Metrics Server requires Docker to be installed on the nodes in your cluster.
- kubectl: Metrics Server requires kubectl to be installed on your local machine.

Here are some of the limitations of using Metrics Server:

- Metrics Server does not collect all metrics: Metrics Server only collects a subset of metrics from Nodes. For a complete set of metrics, you should use a monitoring solution like Prometheus.
- Metrics Server is not a replacement for a monitoring solution: Metrics Server is a good source of data for autoscaling, but it is not a replacement for a full-featured monitoring solution.

Download Metrics Server Manifest

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

Modify Metrics Server Yaml File
vi components.yaml under spec, add: hostNetwork: true goto: 140 - --kubelet-insecure-tls :wq!

```
spec:
  selector:
    matchLabels:
      k8s-app: metrics-server
  strategy:
    rollingUpdate:
      maxUnavailable: 0
  template:
    metadata:
      labels:
        k8s-app: metrics-server
    spec:
      hostNetwork: true
      containers:
      - args:
        - --cert-dir=/tmp
        - --secure-port=4443
        - --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname
        - --kubelet-use-node-status-port
        - --metric-resolution=15s
        - --kubelet-insecure-tls
        image: registry.k8s.io/metrics-server/metrics-server:v0.6.3
        imagePullPolicy: IfNotPresent
        livenessProbe:
          failureThreshold: 3
          httpGet:
            path: /livez
            port: https
            scheme: HTTPS
          periodSeconds: 10
        name: metrics-server
```

Deploy Metrics Server
kubectl apply -f components.yaml

Verify Metrics Server Deployment

kubectl get pods -n kube-system

```
[root@master-node-k8 ~]# kubectl get pods -n kube-system
NAME                                      READY   STATUS    RESTARTS   AGE
calico-kube-controllers-57b57c56f-b7224   1/1     Running   14         21d
calico-node-456x2                         1/1     Running   14         21d
calico-node-lznts                         1/1     Running   15         21d
coredns-787d4945fb-7thhn                  1/1     Running   15         22d
coredns-787d4945fb-mkbn9                  1/1     Running   14         22d
etcd-master-node-k8                       1/1     Running   19         22d
kube-apiserver-master-node-k8             1/1     Running   16         22d
kube-controller-manager-master-node-k8    1/1     Running   19         22d
kube-proxy-47b6v                          1/1     Running   16         22d
kube-proxy-g2cwx                          1/1     Running   11         22d
kube-scheduler-master-node-k8             1/1     Running   19         22d
metrics-server-b76787867-lrc54            1/1     Running   0          6m46s
[root@master-node-k8 ~]#
```

Test Metrics Server Installation

kubectl top nodes
kubectl top pod
kubectl top pod -n kube-system

Configuring Container with "Memory" Requests and Limits:

First, get the output of kubectl top command to find resources that are "currently consumed".

kubectl top nodes

Then, deploy this Pod with memory requests and limits as mentioned below

```
#memory-demo.yaml
apiVersion: v1
kind: Pod
metadata:
  name: memory-demo
spec:
  containers:
  - name: memory-demo-ctr
    image: polinux/stress
    resources:
      requests:
        memory: "100Mi"
      limits:
        memory: "200Mi"
    command: ["stress"]
    args: ["--vm", "1", "--vm-bytes", "150M", "--vm-hang", "1"]
```

Deploy

kubectl apply -f memory-demo.yaml

Validate

kubectl get pods -o wide
kubectl top nodes

SELF-HEALING / AUTO-HEALING

- A system that heals itself without any manual intervention even in an undesirable situation.
- K8s is self-healing system.
- K8s provides multiple solutions to heal itself. Like:
    - Replicaset
    - Deployment
    - Daemon set
    - Etc.
- **Replication:**
    - Kubernetes allows you to define the desired number of replicas for your application.
    - If a pod (an instance of an application) fails, Kubernetes automatically creates new replicas to maintain the desired state.
- **Health checks:**
    - Kubernetes performs health checks on pods using readiness and liveness probes.
    - Readiness probes determine if a pod is ready to serve traffic, while liveness probes check if a pod is still running.
    - If a pod fails these checks, Kubernetes automatically restarts or recreates the pod.
- **Rolling updates:**
    - When deploying new versions of applications, Kubernetes supports rolling updates.
    - It gradually replaces the old instances of the application with the new ones, ensuring minimal downtime and maintaining the desired state throughout the update process.
- **Pod rescheduling:**
    - If a node in the cluster fails or becomes unresponsive, Kubernetes detects it and automatically reschedules the affected pods to other healthy nodes.
    - This ensures that the application continues running even in the face of node failures.
- **Daemonsets:**
    - Kubernetes Daemonsets are used for running system-level daemons on every node in the cluster.
    - If a node is added or removed from the cluster, Daemonsets automatically manage the deployment or termination of daemons to maintain the desired state.
- **Self-monitoring:**
    - Kubernetes itself is self-monitoring, continuously checking the health and state of the cluster components.
    - If it detects any issues, Kubernetes attempts to resolve them automatically or alerts the cluster administrators.

MANIFEST & TEMPLATING

A Kubernetes manifest is a YAML or JSON file that describes a Kubernetes resource. Resources can be pods, services, deployments, and so on. Manifests are used to create, update, and delete Kubernetes resources.

Templating is a way to dynamically generate Kubernetes manifests. This can be useful for things like deploying different versions of an application to different environments, or for creating different configurations for different teams.

There are a few different ways to do templating in Kubernetes. One way is to use Helm, which is a package manager for Kubernetes. Helm provides a number of templates that can be used to deploy common applications.

Another way to do templating is to use Kustomize. Kustomize is a tool that can be used to customize Kubernetes manifests. Kustomize can be used to add, remove, or change the values of fields in a manifest.

Finally, you can also do templating by hand. This is the least common way to do templating, but it can be useful if you need to create a custom template that does not exist in Helm or Kustomize.

Here are some of the benefits of using templating in Kubernetes:

- **Reusability**: Templates can be reused to deploy different versions of an application to different environments, or to create different configurations for different teams.
- **Efficiency**: Templates can be used to generate multiple Kubernetes manifests from a single source file. This can save time and effort.
- **Flexibility**: Templates can be used to create custom Kubernetes manifests that do not exist in Helm or Kustomize.

Here are some of the limitations of using templating in Kubernetes:

- **Complexity**: Templates can be complex to create and maintain.
- **Security**: Templates can contain sensitive information, such as passwords or API keys. It is important to secure templates appropriately.
- **Troubleshooting**: If a template is not working as expected, it can be difficult to troubleshoot. It is important to have good documentation for your templates.

Overall, templating is a powerful tool that can be used to improve the efficiency and flexibility of Kubernetes deployments. However, it is important to be aware of the complexity and security implications of using templating before you decide to use it.

Here are some templating tools:

Helm: Helm is a widely used package manager and templating tool for Kubernetes. It enables you to define, install, upgrade, and manage applications and their dependencies as charts. Helm uses Go templates to generate Kubernetes manifests, allowing you to customize and parameterize resources.

Kustomize: Kustomize is a native configuration management tool in Kubernetes, available since version 1.14. It provides a simple way to customize and manage Kubernetes resources across different environments. Kustomize uses overlays and patches to modify base Kubernetes manifests, allowing you to apply configurations specific to each environment.

Ksonnet: Ksonnet is a tool that helps manage Kubernetes application configurations using a combination of parameterization, templates, and inheritance. It allows you to define reusable components and configurations in a hierarchy, making it easier to manage and maintain complex applications and environments.

Kubecfg: Kubecfg is a Kubernetes configuration utility that provides templating and configuration management capabilities. It uses a JSON or YAML-based configuration format and allows you to apply transformations to generate Kubernetes manifests. Kubecfg can be used to manage multi-environment deployments and support reusable components.

Kapitan: Kapitan is a generic templating tool that can be used for Kubernetes configurations along with other infrastructure-as-code use cases. It allows you to define and manage templates, rendering them into concrete configurations by using JSONnet, Jinja2, or other templating languages.

Ktmpl: Ktmpl is a lightweight command-line tool for Kubernetes manifest templating. It supports simple variable substitution and loop constructs, making it easy to generate Kubernetes YAML manifests from templates. Ktmpl is often used as part of CI/CD pipelines or configuration management scripts.

#cat deployment.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{.Values.name}}-config
data:
  key: value
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{.Values.name}}-deployment
spec:
  replicas: {{int .Values.replicas}}
  selector:
    matchLabels:
      app: {{.Values.name}}
  template:
    metadata:
      labels:
        app: {{.Values.name}}
    spec:
```

```
containers:
- name: {{.Values.name}}
  image: {{.Values.image}}
  envFrom:
  - configMapRef:
      name: {{.Values.name}}-config
  ports:
  - containerPort: 80
```

Values.yaml
name: harness-example replicas: 1 image: ${artifact.metadata.image}

NETWORKING IN K8S

Networking in Kubernetes (K8s) is a fundamental aspect that enables communication between pods, services, and external resources within a cluster. It provides a network overlay that abstracts the underlying infrastructure and allows containers running in different nodes to communicate as if they were on the same network.

Here are the key components and concepts related to networking in Kubernetes:

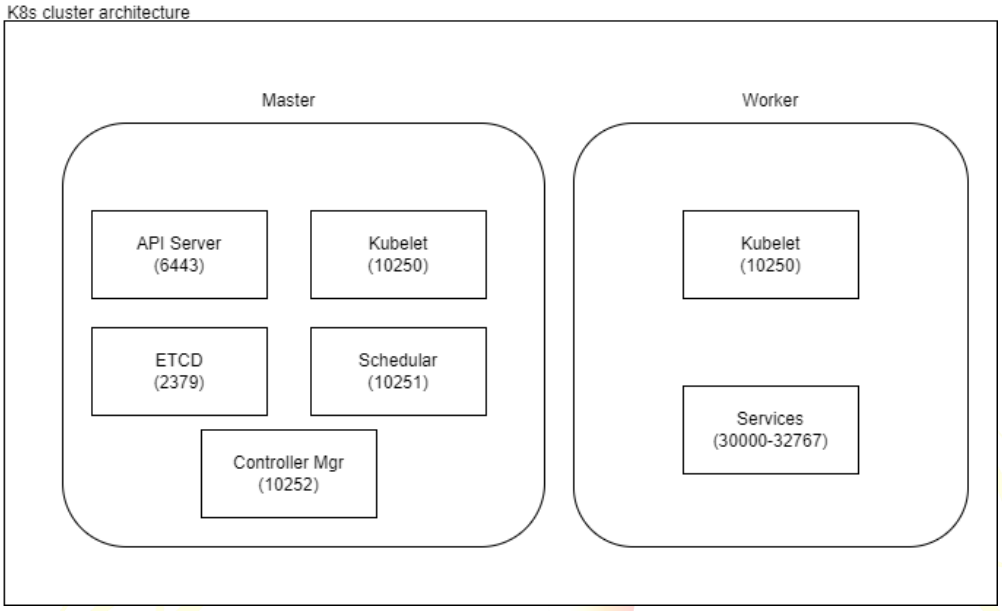
- **Pod Networking**: Pods are the smallest deployable units in Kubernetes. Each pod has its own unique IP address within the cluster, enabling communication between containers running within the same pod. By default, all containers within a pod share the same network namespace, allowing them to communicate over the localhost interface.
- **Cluster Networking**: Kubernetes requires a networking solution to enable communication between pods running on different nodes. Various networking plugins can be used, such as Flannel, Calico, Weave, and Cilium. These plugins provide network overlays, routing, and IP address management to establish connectivity between pods in the cluster.
- **Service Networking**: Services in Kubernetes provide a stable endpoint for accessing a group of pods, allowing load balancing and service discovery. Each service is assigned a virtual IP (ClusterIP) that remains stable even if the pods behind it change. Services can be exposed within the cluster or externally using different service types, such as ClusterIP, NodePort, LoadBalancer, or ExternalName.
- **Ingress**: Ingress is an API object in Kubernetes that allows external access to services within the cluster. It provides a way to configure rules for routing external traffic to different services based on hostnames, paths, or other criteria. Ingress controllers, such as Nginx Ingress Controller or Traefik, are responsible for implementing the Ingress rules and handling the incoming traffic.
- **DNS**: Kubernetes provides a built-in DNS service that allows pods and services to discover and communicate with each other using DNS names. Each service in the cluster is assigned a DNS name, and pods can resolve the DNS names of other pods or services to establish communication. DNS resolution in Kubernetes typically follows the <service-name>.<namespace>.svc.cluster.local format.
- **Network Policies**: Network Policies are Kubernetes resources used for enforcing network traffic rules within a cluster. They provide a way to define ingress and egress rules for pods and control communication between them based on IP addresses, ports, protocols, or labels. Network Policies help enforce security and segmentation requirements in multi-tenant or complex cluster environments.

Cluster networking – communication methods:

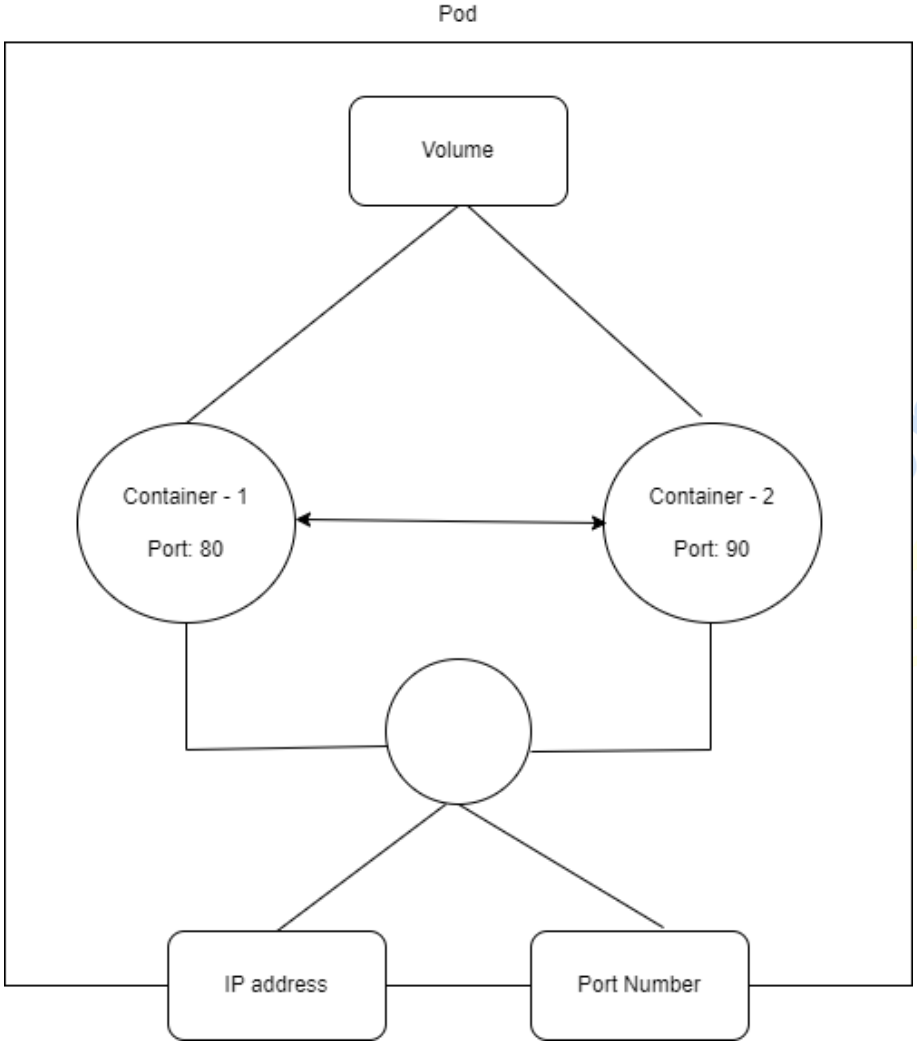
1. Container-to-container communication.
2. Pod-to-pod communication.
3. Pod-to-service communication.
4. External-to-service communication.

CNI stands for Container Networking Interface. In Kubernetes, CNI is a specification and set of standards that define how network plugins should interact with the underlying network infrastructure to provide networking capabilities for containers and pods.

Networking ports:



Container to container



From container1, you want to access container2

curl localhost:90

From container2, you want to access container1

curl localhost:80

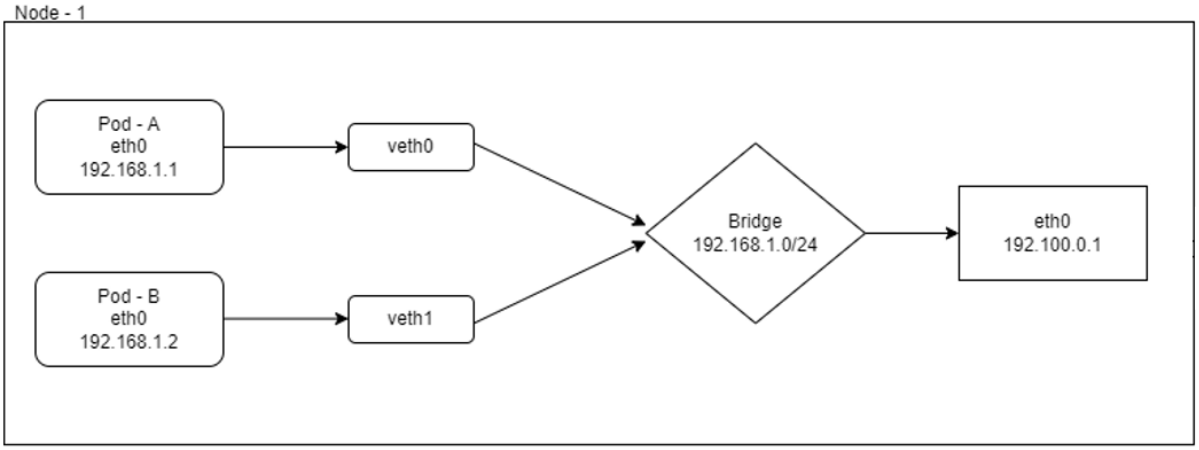
Creating container-to-container (intra-pod)

```
# cat intra-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: intrapod
spec:
  containers:
  - name: webserver
    image: nginx
  - name: centos
    image: centos
    command: ["/bin/sh","-c","while : ; do curl http://localhost:80/; sleep 10 ; done"]

Applying
# kubectl apply -f intra-pod.yaml
```

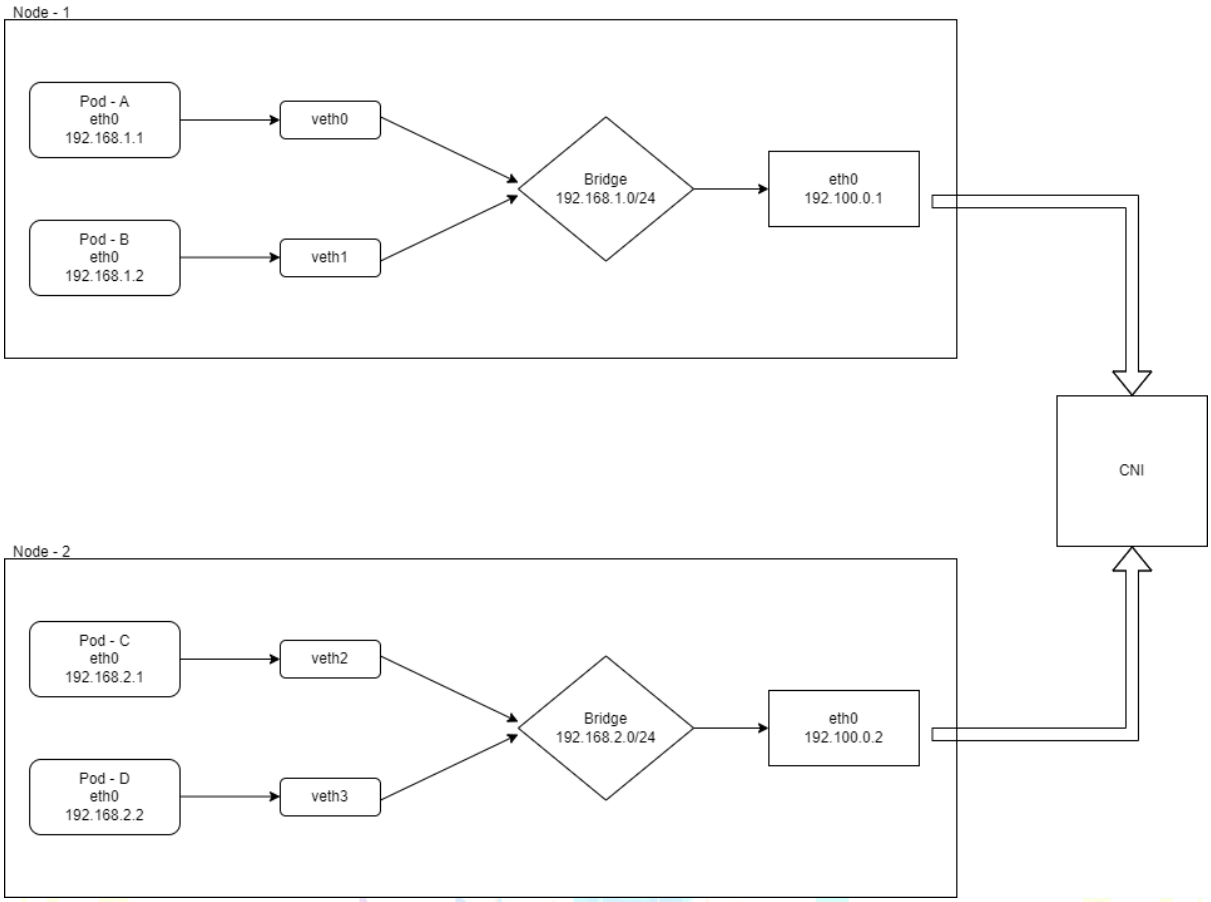
Pod-to-pod communication.

For single node.



Pod-A → vetho0 → bridge → veth1 → Pod-B

For pods in different nodes.



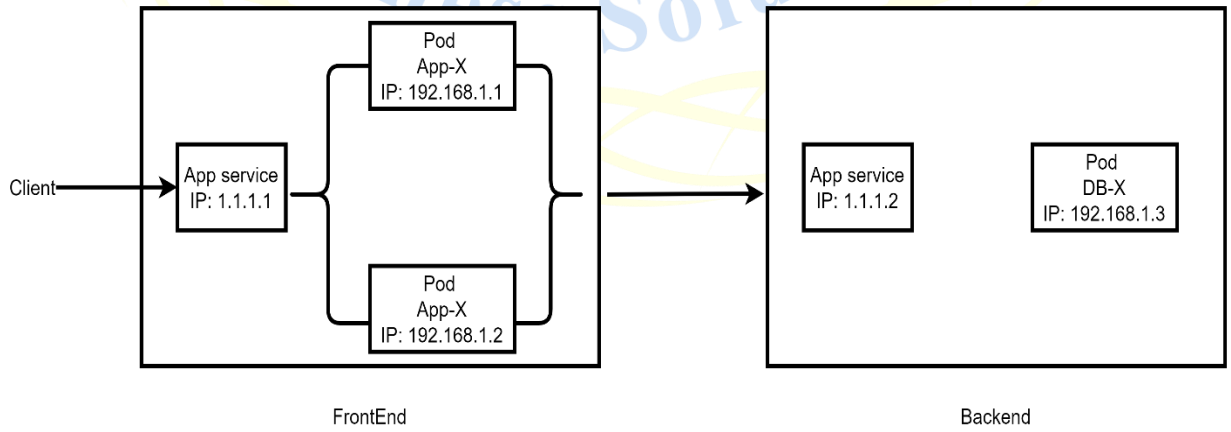
Pod-B → eth1 → veth1 → bridge (192.168.1.0/24) → CNI → bridge (192.168.2.0/24) → veth3 → Pod-D

SERVICE

A Service is an abstraction that defines a set of pods and a policy to access them. Services provide a stable endpoint and facilitate communication between different parts of an application or between applications within a Kubernetes cluster.

Here are the key aspects of Services in Kubernetes:

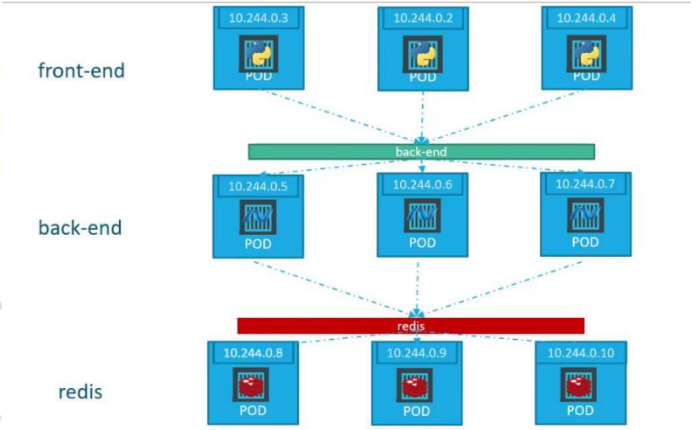
- **Pod Discovery and Load Balancing**: A Service acts as a stable endpoint for accessing a group of pods that perform the same function. It abstracts the underlying pod IP addresses and provides a single endpoint (ClusterIP) that remains consistent even if the pods are scaled up or down. Services distribute incoming traffic to the pods using load balancing, ensuring that requests are evenly distributed.
- **Service Types**: Kubernetes supports different types of Services to expose and access the pods. The common service types are:
    - **ClusterIP**: The default service type. It provides a virtual IP address accessible only within the cluster. This type is suitable for inter-pod communication within the cluster.
    - **NodePort**: Exposes the service on a static port on each node's IP address. It allows external access to the service by reaching any cluster node on the specified port.
    - **LoadBalancer**: Configures a cloud provider's load balancer to distribute external traffic to the service. It assigns an external IP address accessible from outside the cluster.
    - **ExternalName**: Maps the service to an external DNS name without any proxying or load balancing. It is commonly used to refer to services outside the cluster.
- **DNS-Based Service Discovery**: Services in Kubernetes have DNS names assigned to them, making it easy for other services or pods within the cluster to discover and communicate with them. The DNS name takes the form <service-name>.<namespace>.svc.cluster.local, allowing seamless service-to-service communication using DNS resolution.
- **Labels and Selectors**: Services are associated with a set of pods using labels and selectors. The selector is a label-based query used to match pods that should be included in the service. By applying labels to pods, you can control which pods are part of a specific service.
- **Headless Services**: In addition to the standard service types, Kubernetes supports headless services. A headless service does not allocate a ClusterIP and instead allows direct access to individual pod IPs behind the service. It is useful when you need direct access to specific pods or when you want to perform manual load balancing or service discovery.



SERVICE TYPES:

1. Cluster IP service
2. Node port service
3. Load balancer service
4. Ingress service

Cluster IP Service:



A Cluster IP service is a type of service that provides a stable internal IP address for accessing a set of pods within a cluster. It allows other resources within the cluster to communicate with the pods using a single, virtual IP address.

When you create a Cluster IP service, Kubernetes assigns it an internal IP address from the cluster's Service CIDR (Classless Inter-Domain Routing) range. This IP address is accessible only from within the cluster and is not exposed to the outside world. Cluster IP services are typically used for internal communication between different components within the cluster.

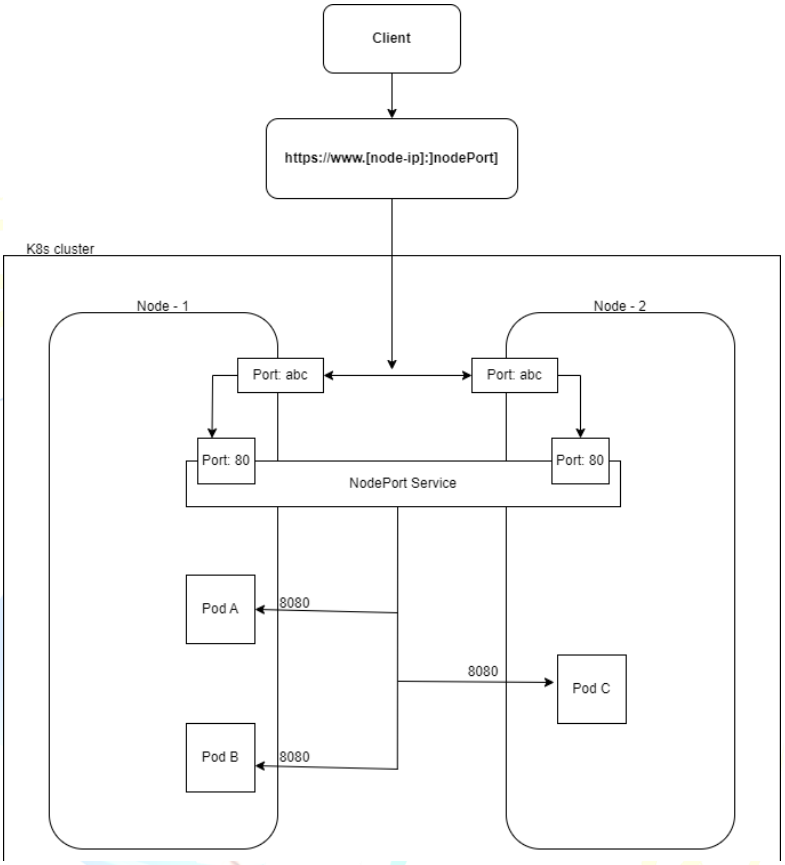
Here are some key features and characteristics of a Cluster IP service:

- **Internal access**: Cluster IP services can be accessed by other pods or services within the cluster using the service name and port.
- **Load balancing**: If multiple pods are part of the service, the Cluster IP service automatically load balances the traffic among the pods.
- **Stable IP address**: The IP address assigned to the Cluster IP service remains the same as long as the service exists, even if pods are created or destroyed.
- **Internal-only visibility**: Cluster IP services are not accessible from outside the cluster or from other networks unless additional configurations, such as using Kubernetes ingress resources or a VPN, are implemented.
- **DNS resolution**: Kubernetes automatically creates DNS records for Cluster IP services, allowing other pods or services to discover and communicate with them using their DNS names.

Node port service

In Kubernetes, a NodePort service is a way to expose a service externally by allocating a specific port on each node in the cluster. It allows traffic to be directed to the service on that port, which then gets forwarded to the appropriate pods.

When you create a NodePort service, Kubernetes automatically assigns a port from a predefined range (typically between 30000 and 32767) to the service. This allocated port is exposed on every node's IP address, allowing external traffic to reach the service.



Here's a high-level overview of how a NodePort service works:

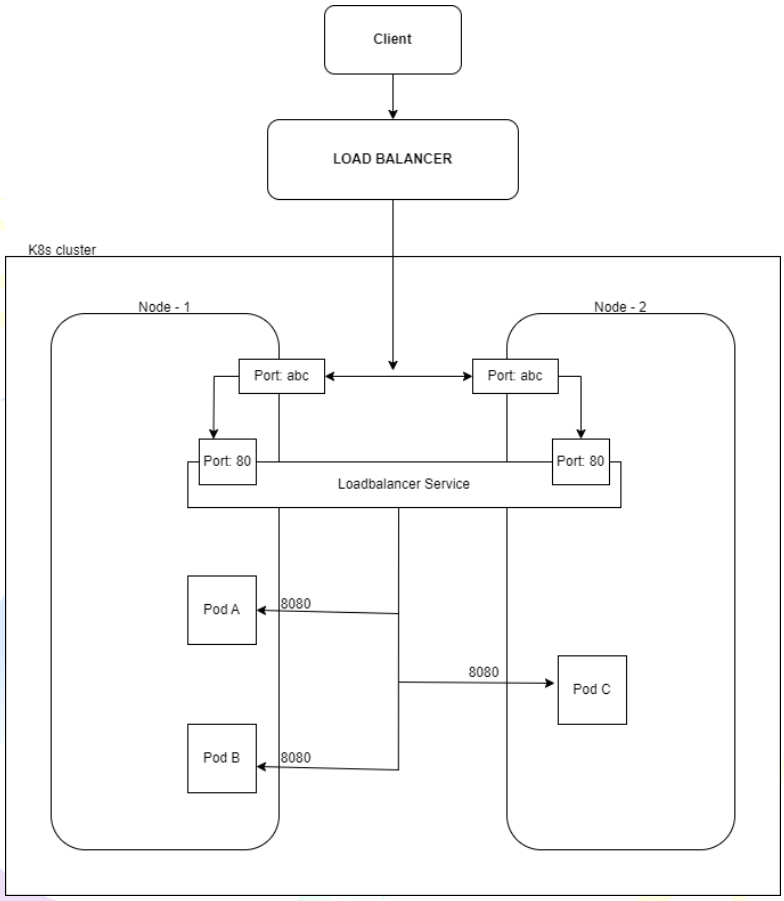
1. You define a service in Kubernetes with the type: NodePort specification.
2. Kubernetes allocates a random port from the predefined range and assigns it to the service.
3. Each node in the cluster opens that port and starts forwarding traffic to the service.
4. When an external client connects to any node's IP address on the allocated port, the traffic is directed to the service.
5. Kubernetes internally load balances the traffic to the pods associated with the service, using labels and selectors.

NodePort services are commonly used when you need to expose a service to external clients or when you have external dependencies that need to access the service. However, keep in mind that the allocated port is exposed on all nodes, which can be a security concern. It's also important to note that the NodePort range can be customized if needed.

Load balancer service

In Kubernetes, a LoadBalancer service is a type of service that automatically provisions and configures an external load balancer to distribute traffic across the pods of the service. It allows you to expose your service externally and distribute traffic in a balanced manner.

When you create a LoadBalancer service, Kubernetes interacts with the underlying cloud provider (such as AWS, Azure, or GCP) to provision a load balancer and assign it an external IP address. The load balancer then distributes incoming traffic across the pods associated with the service.



Here's a high-level overview of how a LoadBalancer service works:

1. You define a service in Kubernetes with the type: LoadBalancer specification.
2. Kubernetes communicates with the cloud provider's API to request the creation of a load balancer.
3. The cloud provider provisions a load balancer and assigns it an external IP address.
4. The load balancer listens for incoming traffic on a specific port (usually port 80 for HTTP or port 443 for HTTPS).
5. External clients connect to the load balancer's IP address and send traffic to the service.
6. The load balancer evenly distributes the traffic across the pods associated with the service, using labels and selectors.
7. Kubernetes continuously monitors the health of the pods and updates the load balancer's configuration to reflect any changes.

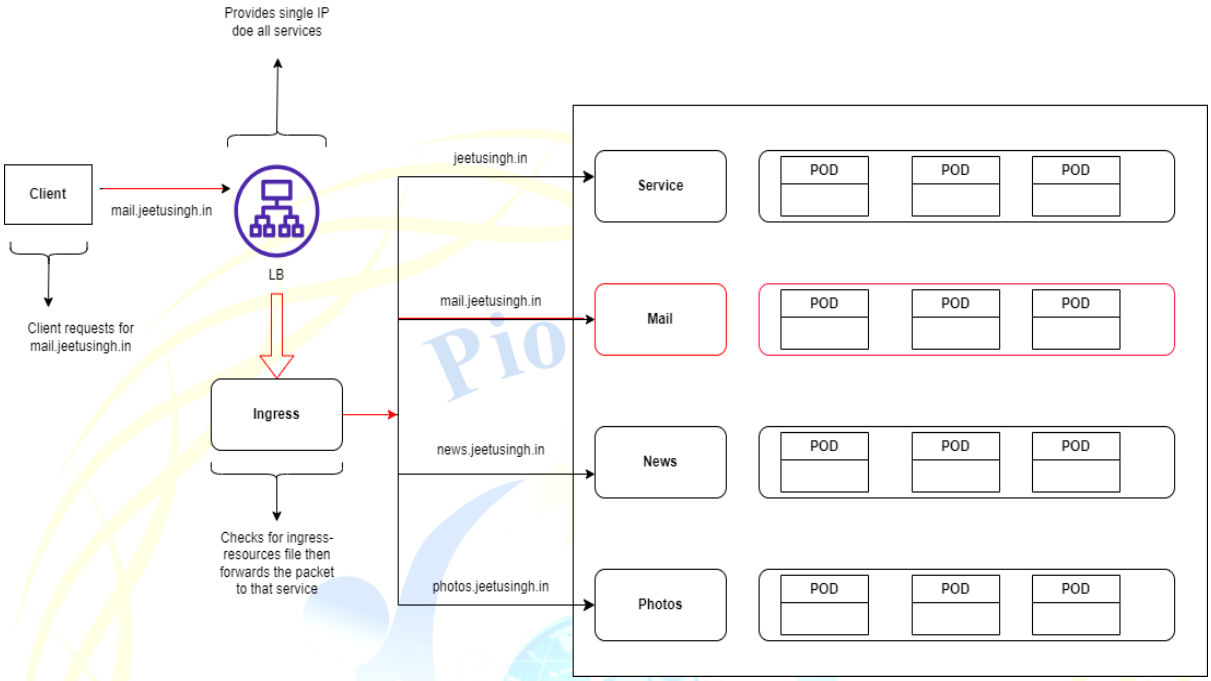
LoadBalancer services are ideal when you require a simple and efficient way to expose your service to the internet or external clients. The underlying cloud provider handles the load balancer setup and configuration, making it easier for you to expose your application.

It's important to note that not all Kubernetes deployments have native integration with cloud providers. In such cases, LoadBalancer services might not work out of the box. However, there are alternative solutions, such as using an external load balancer or configuring Ingress controllers, which can achieve similar functionality.

Additionally, it's worth mentioning that LoadBalancer services can have an additional cost associated with the provisioned load balancer, depending on the cloud provider's pricing model.

Ingress service

In Kubernetes, an Ingress is an API object that provides a way to expose HTTP and HTTPS routes from outside the cluster to services within the cluster. It acts as a reverse proxy, routing external traffic to the appropriate services based on rules defined in the Ingress resource.



The Ingress resource itself is not a service type like NodePort or LoadBalancer. Instead, it serves as a configuration object that defines rules for routing and load balancing traffic to services. To make use of the Ingress resource, you need an Ingress controller—a separate component or software that understands and enforces the Ingress rules.

Here's a high-level overview of how Ingress works:

- You define an Ingress resource that specifies the rules for routing and load balancing traffic.
- An Ingress controller, which is usually a separate component running in the cluster, watches for changes to Ingress resources and configures itself accordingly.
- External clients send requests to the Ingress controller using the configured hostnames and paths.
- The Ingress controller evaluates the rules defined in the Ingress resource and forwards the traffic to the appropriate service.
- The service then routes the traffic to the corresponding pods based on labels and selectors.

Ingress provides more advanced features than simple load balancing. To use Ingress in Kubernetes, you need to deploy an Ingress controller into your cluster. Popular Ingress controllers include Nginx Ingress Controller, Traefik, and HAProxy Ingress. The specific steps for installing and configuring an Ingress controller depend on the chosen implementation.

Overall, Ingress provides a powerful way to manage external access to services in your Kubernetes cluster, allowing you to define routing rules and leverage advanced features for traffic handling.

STORAGE

In Kubernetes, a storage volume is a directory or filesystem that is accessible to containers in a pod. It provides a way to persist and share data between containers, as well as to retain data across pod restarts or rescheduling.

Kubernetes offers various types of storage volumes to meet different use cases. Here are some commonly used volume types:

- **EmptyDir**: An EmptyDir volume is created when a pod is scheduled and exists for the lifetime of that pod. It is initially empty and can be used for temporary storage or sharing files between containers within the same pod.
- **HostPath**: A HostPath volume mounts a file or directory from the host node's filesystem into the pod. It allows containers to access and share files directly on the host node, but it is specific to the node where the pod is scheduled.
- **PersistentVolumeClaim** (PVC): A PersistentVolumeClaim is an abstraction layer that allows pods to request storage resources from a PersistentVolume (PV). PVs are provisioned by administrators and can be backed by different storage solutions like network-attached storage (NAS), cloud provider disks, or local storage. PVCs provide a way for pods to request a specific amount and type of storage, and Kubernetes handles binding a suitable PV to the PVC.
- **CSI Volumes**: Container Storage Interface (CSI) is a standardized interface for connecting external storage systems to Kubernetes. CSI volumes allow you to integrate with various storage providers and enable advanced features like dynamic provisioning, snapshotting, and encryption.
- **Cloud Provider-Specific Volumes**: Kubernetes integrates with cloud providers to offer volume types specific to their storage solutions. For example, AWS Elastic Block Store (EBS), Azure Disk, or Google Cloud Persistent Disk. These volumes are provisioned and managed by the cloud provider and provide features such as snapshots and encryption.
- **NFS**: NFS (Network File System) volumes allow you to mount network-shared file systems into pods. It provides a way to access and share files across multiple pods in a distributed manner.
- **GlusterFS**: GlusterFS is a distributed file system that can be used as a volume in Kubernetes. It provides scalable and highly available network storage for pods.
- **iSCSI**: iSCSI volumes enable the use of block-level storage in Kubernetes. It allows you to connect and mount block devices from storage arrays to pods.

EmptyDir

EmptyDir is a type of storage volume in Kubernetes that is created when a pod is scheduled and is tied to the pod's lifecycle. It provides a temporary directory for containers within the pod to read from and write to.

Here are some key characteristics of EmptyDir volumes:

- Lifetime: An EmptyDir volume is created when the pod is scheduled and exists as long as the pod is running. It is tightly coupled with the pod's lifecycle. When the pod is terminated or deleted, the EmptyDir volume is also removed, along with any data stored within it.
- Data Sharing: EmptyDir volumes can be shared by multiple containers within the same pod. This enables containers to read from and write to the same directory, facilitating data exchange or sharing of files between containers.

- Data Persistence: The data stored in an EmptyDir volume is not persisted beyond the lifespan of the pod. If the pod restarts or is rescheduled to a different node, the EmptyDir volume is recreated, and any existing data is lost. Therefore, EmptyDir volumes should not be used for long-term or critical data storage.
- Size and Limitations: EmptyDir volumes do not have a predefined size. Instead, they consume disk space from the underlying node's filesystem. The available capacity depends on the available disk space on the node. It's important to note that if the node's disk becomes full, it can impact the stability and performance of the entire cluster.
- Use Cases: EmptyDir volumes are commonly used for scenarios that require temporary storage or data sharing between containers in the same pod. Examples include sharing configuration files, caching data, or creating temporary files during batch processing.

HostPath

HostPath is a type of storage volume in Kubernetes that allows a pod to mount a directory or file from the host node's filesystem directly into the pod. With HostPath volumes, containers within the pod can access and manipulate files or directories on the host node.

Here are some key characteristics of HostPath volumes:

- Direct Access: HostPath volumes provide direct access to the host node's filesystem. This means that any modifications made to the files or directories within the HostPath volume will affect the corresponding files on the host node.
- Node-specific: HostPath volumes are specific to the node on which the pod is scheduled. If the pod is rescheduled or moved to a different node, it will no longer have access to the files or directories within the HostPath volume on the previous node.
- Data Persistence: HostPath volumes offer persistence, meaning that the files or directories within the volume remain intact even if the pod restarts. However, keep in mind that if the pod is deleted, the data within the HostPath volume is lost.
- Access Permissions: Containers within the pod can read from and write to the files or directories in the HostPath volume, assuming they have the necessary file permissions.
- Security Considerations: Using HostPath volumes requires caution, as it provides direct access to the host node's filesystem. It can potentially expose sensitive system files or allow containers to interfere with the host's operating system. Therefore, it's crucial to ensure that appropriate security measures are in place and only mount the necessary files or directories.

```
# mkdir /test-vol

# cat > /test-vol/dummy.txt
this is Jeetu.

# cat nginx-hostpath.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-hostpath
spec:
  containers:
    - name: nginx-container
      image: nginx
      volumeMounts:
```

```
    - mountPath: /test-mnt
      name: test-vol
  volumes:
  - name: test-vol
    hostPath:
      path: /test-vol


Deploy:
-------
# kubectl apply -f nginx-hostpath.yaml

get inside pod:
# kubectl exec -it nginx-hostpath -- /bin/bash

list mount points:
# df -h

list contents:
# ls -h /test-mnt/
```

PersistentVolume

In Kubernetes, a PersistentVolume (PV) is a storage abstraction that represents a piece of storage in the cluster. It provides a way to decouple storage resources from pods and allows for persistent storage that can be dynamically provisioned and managed.

Here are the key aspects of PersistentVolumes:

- **Provisioning**: PersistentVolumes can be provisioned in advance by cluster administrators or dynamically created by using PersistentVolumeClaims (PVCs). Administrators can pre-provision PVs and make them available for consumption by creating PV objects. Alternatively, PVCs can request specific storage requirements, and the cluster can dynamically provision the requested PV to fulfill the claim.
- **Capacity and Access Modes**: Each PersistentVolume has a defined capacity, which specifies the amount of storage available. PVs can also have access modes that define how the storage can be accessed, such as ReadWriteOnce (RWX), ReadOnlyMany (ROX), or ReadWriteMany (RWX). These access modes determine whether the PV can be mounted by a single node, multiple nodes, or read-only by multiple nodes.
- **Backing Storage**: PersistentVolumes can be backed by various storage solutions, such as network-attached storage (NAS), cloud provider disks, local storage on a node, or even external storage systems through the Container Storage Interface (CSI). The actual implementation and integration with storage providers depend on the Kubernetes cluster's setup and the available storage drivers.
- **Lifecycle and Reclamation**: PersistentVolumes have a lifecycle independent of the pods that use them. They persist data even if the associated pods are terminated. PVs can be manually deleted by administrators or released and reclaimed based on specified reclaim policies. The

reclaim policy determines what happens to the data when the PV is released, such as deleting the data or retaining it for future use.

- **PersistentVolumeClaims** (PVCs): PersistentVolumeClaims are requests made by pods for storage resources. PVCs specify the desired capacity, access mode, and other requirements. They can be bound to existing PVs or dynamically provisioned by the cluster. The PVC acts as a middle layer between pods and PVs, allowing pods to consume storage without directly referencing specific PVs.

PersistentVolumeClaim

In Kubernetes, a PersistentVolumeClaim (PVC) is an API object used by a pod to request storage resources from a PersistentVolume (PV). PVCs provide a way to dynamically provision and manage storage volumes, allowing pods to consume storage without directly specifying or knowing the details of the underlying PV.

Here are the key aspects of PersistentVolumeClaims:

- **Requesting Storage**: When a pod requires storage, it creates a PVC object that specifies the desired storage capacity, access mode, and other requirements. The PVC serves as a request for a suitable PV that can fulfill the specified criteria.
- **Binding to PersistentVolumes**: PVCs can be bound to either pre-existing PVs or dynamically provisioned PVs. If a PVC specifies specific requirements that match the characteristics of an available PV, it can be bound to that PV. Alternatively, if dynamic provisioning is enabled, the cluster automatically creates a new PV that satisfies the PVC's requirements.
- **Access Modes**: PVCs define the desired access mode for the storage. The access modes determine how the storage can be accessed by the pods. The available modes are ReadWriteOnce (RWX), ReadOnlyMany (ROX), and ReadWriteMany (RWX). RWX allows read-write access by a single node, ROX allows read-only access by multiple nodes, and RWX allows read-write access by multiple nodes.
- **Lifecycle and Binding**: PVCs have a lifecycle independent of pods. They can be created, bound to PVs, used by pods, and eventually deleted. When a PVC is created, it enters a Pending state until a suitable PV is bound to it. Once a binding occurs, the PVC is said to be Bound and becomes available for the pod to use.
- **Dynamic Provisioning**: Kubernetes supports dynamic provisioning of PVs based on PVCs. Dynamic provisioning allows PVCs to automatically create and bind to a PV that matches the specified requirements, eliminating the need for manual PV provisioning.

CSI Volumes

CSI (Container Storage Interface) Volumes in Kubernetes refer to storage volumes provisioned and managed using the Container Storage Interface standard. CSI is a standardized interface that allows storage vendors to integrate their storage solutions with Kubernetes, providing advanced storage features and capabilities.

Here are the key aspects of CSI Volumes:

- **CSI Integration**: CSI Volumes enable seamless integration with third-party storage solutions. Storage vendors implement CSI drivers that communicate with Kubernetes through the CSI interface. This allows for the dynamic provisioning, attachment, and management of storage volumes provided by external storage systems.

- **Dynamic Provisioning**: CSI Volumes support dynamic provisioning, allowing storage volumes to be dynamically created based on PersistentVolumeClaims (PVCs). When a PVC is created, the CSI driver interacts with the external storage system to provision the requested volume that matches the specified capacity, access mode, and other requirements.
- **Advanced Storage Features**: CSI Volumes provide access to advanced storage features supported by the underlying storage solution. This may include features like volume expansion, snapshotting, cloning, encryption, performance tuning, and other capabilities specific to the storage provider.
- **Vendor Flexibility**: CSI Volumes enable flexibility in choosing storage vendors and solutions. As long as a CSI driver is available, you can use any storage solution that implements the CSI standard. This allows for multi-cloud and hybrid cloud deployments, as well as the ability to integrate with different storage backends based on specific requirements.
- **Lifecycle and Management**: CSI Volumes have a lifecycle independent of pods. They persist data even if pods are terminated or rescheduled. The management of CSI Volumes is typically handled by the storage provider and their respective CSI driver, ensuring proper attachment, detachment, and resource management.
- **Configuration and Deployment**: To use CSI Volumes, you need to deploy the appropriate CSI driver provided by your storage vendor. This involves installing the CSI driver components in the cluster and configuring the necessary parameters and credentials. Once the driver is deployed, you can create PVCs that specify the CSI storage class, allowing pods to consume the dynamically provisioned CSI Volumes.

Cloud Provider-Specific Volumes

Cloud Provider-Specific Volumes in Kubernetes refer to storage volumes that are specifically designed and integrated with the storage solutions provided by various cloud providers. These volume types are available in Kubernetes clusters running on specific cloud platforms, such as AWS, Azure, Google Cloud, or other cloud providers.

Here are the key aspects of Cloud Provider-Specific Volumes:

- **Native Integration**: Cloud Provider-Specific Volumes leverage the native storage solutions offered by the cloud provider. This integration allows Kubernetes clusters to seamlessly utilize the storage services provided by the underlying cloud platform, taking advantage of their unique features and capabilities.
- **Storage Offerings**: Each cloud provider offers its own set of storage services, such as AWS Elastic Block Store (EBS), Azure Disk, Google Cloud Persistent Disk, or similar offerings from other providers. These services typically provide block-level storage that can be provisioned and attached to Kubernetes pods as persistent volumes.
- **Integration with Kubernetes**: Cloud Provider-Specific Volumes are integrated with Kubernetes through specific storage plugins and drivers. These plugins enable Kubernetes clusters to interact with the cloud provider's storage API and manage the lifecycle of the storage volumes. They handle operations such as volume provisioning, attachment, detachment, and management.
- **Dynamic Provisioning**: Cloud Provider-Specific Volumes often support dynamic provisioning, allowing storage volumes to be dynamically created based on PersistentVolumeClaims (PVCs). When a PVC is created, the cluster communicates with the cloud provider's storage API to provision the requested volume that matches the specified capacity, access mode, and other requirements.

- **Performance and Features**: Cloud Provider-Specific Volumes often offer performance optimizations and additional features specific to the cloud platform. These features may include features like storage snapshots, encryption, automatic backup and recovery, replication, availability zones, and other cloud-specific capabilities.
- **Configuration and Authentication**: To use Cloud Provider-Specific Volumes, proper configuration and authentication with the cloud provider are required. This involves setting up credentials, access control, and any necessary configuration parameters to enable the communication between the Kubernetes cluster and the cloud provider's storage services.

NFS

NFS (Network File System) is a distributed file system protocol that allows file sharing and remote access to files over a network. In the context of Kubernetes, NFS can be used as a storage solution to provide persistent volumes for pods.

Here are the key aspects of NFS in Kubernetes:

- **Shared File System**: NFS enables multiple servers or clients to share files over a network. It allows clients to mount and access remote directories as if they were local file systems. The server exporting the file system makes the files available for access by the clients.
- **Persistent Volume Provider**: In Kubernetes, NFS can be used as a persistent volume provider. It allows pods to request storage using PersistentVolumeClaims (PVCs) and mount remote NFS directories as persistent volumes.
- **Network-Based**: NFS relies on network connectivity between the server exporting the file system and the clients that mount and access the files. The server exports specific directories or file systems, and the clients can access them over the network using NFS protocol.
- **Read-Write Access**: NFS supports both read and write access to the shared file system. Pods can read from and write to the mounted NFS directories, allowing for persistent storage and data sharing across multiple pods.
- **Configuration**: To use NFS as a persistent volume provider in Kubernetes, you need to have an NFS server set up and running. The NFS server should export the directories that you want to use as persistent volumes.
- **Flexibility and Scalability**: NFS provides flexibility and scalability in storage management. You can have multiple pods mount the same NFS directory, enabling data sharing between pods. Additionally, NFS can support dynamic provisioning by utilizing tools like dynamic NFS provisioners, allowing for on-demand creation of NFS persistent volumes.

GlusterFS

GlusterFS is an open-source distributed file system that allows the creation of scalable and highly available network-attached storage (NAS) solutions. It is designed to aggregate multiple storage servers into a unified global namespace, providing a single point of access for storing and retrieving files.

Here are the key aspects of GlusterFS:

- **Distributed File System**: GlusterFS works by creating a distributed file system that spans multiple storage servers, known as "bricks." Each brick contributes a portion of the overall storage capacity. By combining these bricks, GlusterFS creates a scalable and distributed storage infrastructure.

- **Scalability and Performance**: GlusterFS is designed to scale horizontally, allowing the addition or removal of storage servers without disrupting the overall system. This scalability enables GlusterFS to handle large amounts of data and serve high-performance workloads.
- **Redundancy and High Availability**: GlusterFS provides built-in redundancy and fault tolerance through data replication and distribution. Data can be replicated across multiple storage servers to ensure data availability in case of hardware failures. This redundancy enhances the system's resilience and provides high availability for stored data.
- **Global Namespace**: GlusterFS presents a single global namespace, allowing users or applications to access files through a unified path. Regardless of which storage server the file resides on, the namespace ensures consistent and transparent access.
- **Data Consistency**: GlusterFS supports different modes of data consistency, including strong consistency and eventual consistency. Strong consistency ensures that all clients see a consistent view of the file system, while eventual consistency relaxes the consistency requirements, allowing for higher performance in certain scenarios.
- **Integration with Kubernetes**: GlusterFS can be integrated with Kubernetes as a storage solution. It enables the creation of PersistentVolumes (PVs) and PersistentVolumeClaims (PVCs) that leverage GlusterFS as the underlying storage backend for pods in a Kubernetes cluster.
- **Flexibility and Use Cases**: GlusterFS is versatile and can be used for various storage use cases, such as shared file storage, distributed data analytics, content delivery networks (CDNs), backup and archiving, and more. Its flexible architecture and support for different storage configurations make it adaptable to different application requirements.

iSCSI

iSCSI (Internet Small Computer System Interface) in Kubernetes is a storage networking protocol that allows the use of block-level storage over an IP network. It enables the attachment and remote access of block devices, typically provided by storage systems or SAN (Storage Area Network), to Kubernetes pods as persistent volumes.

Here are the key aspects of iSCSI in Kubernetes:

- **Block-Level Storage**: iSCSI operates at the block level, meaning it allows direct access to storage devices or disks rather than accessing files through a file system. It provides a way to present block devices over the network, allowing pods to use them as persistent volumes.
- **IP-Based Protocol**: iSCSI uses standard IP networks for communication between the storage system (iSCSI target) and the Kubernetes nodes (iSCSI initiators). It leverages TCP/IP as the transport protocol and encapsulates SCSI (Small Computer System Interface) commands within IP packets.
- **Storage Provider**: iSCSI requires a storage system or SAN that supports the iSCSI protocol to provide the block devices. This storage system, known as the iSCSI target, exposes the block devices to the Kubernetes cluster for consumption.
- **Initiator and Target**: In the context of iSCSI, the Kubernetes nodes act as initiators, while the storage systems providing the block devices are the targets. The initiator (Kubernetes node) sends iSCSI commands to the target (storage system) to establish a connection and access the block devices.
- **iSCSI Initiator Configuration**: To use iSCSI in Kubernetes, each node (initiator) needs to have the necessary iSCSI initiator software installed and configured. The iSCSI initiator software

enables the node to discover and connect to the iSCSI targets, authenticate, and mount the block devices as persistent volumes.

- **Persistent Volume Configuration**: In Kubernetes, you define iSCSI-based persistent volumes (PVs) and corresponding PersistentVolumeClaims (PVCs) to request and consume the iSCSI block devices. The PVs contain the necessary information, such as the target's IP address, port, iSCSI initiator name, and authentication details, to establish a connection and mount the block device to the pod.
- **Multipathing and High Availability**: iSCSI supports multipathing, allowing multiple network interfaces on the Kubernetes node to connect to the iSCSI target for increased redundancy and bandwidth. This enhances availability and provides load balancing capabilities for the storage connections.

MONITORING

Tracking all/any resources for efficiently working of cluster.

Why monitoring?

1. Reliability
2. Insights
3. Cost control

PRE-REQ: Installing Metrics-Server (Required for "kubectl top" command)

a. Download

git clone https://github.com/kubernetes-sigs/metrics-server.git

b. Install

kubectl apply -k metrics-server/manifests/test

c. Validate

kubectl get deployment metrics-server -n kube-system

kubectl get pods -n kube-system | grep metrics

kubectl get apiservices | grep metrics

kubectl top pods

kubectl top nodes

Note: Give it a minute if nothing shows up in top command output.

1. NODEs: Ensure all nodes are "Healthy" and "Ready". Monitor its resource usage (CPU & Memory):

kubectl run nginx-pod --image=nginx

kubectl run redis-pod --image=redis

kubectl top nodes

kubectl top nodes --sorty-by cpu

kubectl top nodes --sorty-by memory

kubectl top nodes --sorty-by memory > mem-out.txt

2. PODs: Ensure all Pods are "Running" successfully and Monitor its resource usage (CPU & Memory):

In Current Namepsace:

kubectl top pods

kubectl top pods --sorty-by cpu

kubectl top pods --sorty-by memory

kubectl top pods --sorty-by memory > mem-out.txt

In Specific NameSpace:

kubectl top pods -n [NAME-SPACE]

kubectl top pods -n [NAME-SPACE] --sorty-by cpu

kubectl top pods -n [NAME-SPACE] --sorty-by memory

kubectl top pods -n [NAME-SPACE] --sorty-by memory > mem-out.txt

In Across Namespaces:

kubectl top pods -A

kubectl top pods -A --sorty-by cpu

kubectl top pods -A --sorty-by memory

kubectl top pods -A --sorty-by memory > mem-out.txt

When Multi-Container Pod:

kubectl top pod [POD-NAME] --containers

3. Cluster Components: Ensure all K8s Cluster Components are "Healthy" and "Running" status:

If Cluster configured with "Kubeadm"

kubectl get pods -n kube-system

If cluster configured Manually (the hard-way)

systemctl status kube-apiserver

systemctl status kube-controller-manager

systemctl status kube-scheduler

Ensure, following components are in "Running" status on all nodes including Master(Control-Plane) node.

systemctl status docker

systemctl status kubelet

Logging

Counter for the logging

```
# cat counter.yaml
apiVersion: v1
kind: Pod
metadata:
  name: one-counter-pod
spec:
  containers:
  - name: counter-container
    image: busybox
    args: [/bin/sh, -c,
          'i=0; while true; do echo "$i: $(date)"; i=$((i+1)); sleep 1; done']
---
apiVersion: v1
kind: Pod
metadata:
  name: two-counter-pod
spec:
  containers:
  - name: counter-1
    image: busybox
    args: [/bin/sh, -c,
          'i=0; while true; do echo "From Counter-ONE: $i: $(date)"; i=$((i+1)); sleep 1; done']
  - name: counter-2
    image: busybox
    args: [/bin/sh, -c,
          'i=0; while true; do echo "From Counter-TWO:  $i: $(date)"; i=$((i+1)); sleep 1; done']
```

Display Container Logs using "kubectl logs" command:

```
# kubectl logs [POD-NAME]                    # dump pod logs (stdout)
# kubectl logs -f [POD-NAME]                 # stream pod logs (stdout)
# kubectl logs [POD-NAME] —-since=5m         # view logs for last 5 mins (h for hours)
# kubectl logs [POD-NAME] --tail=20          # Display only more recent 20 lines of output in pod
# kubectl logs [POD-NAME] --previous         # dump pod logs (stdout) for a previous instantiation
of a container
# kubectl logs [POD-NAME] > [FILE-NAME].log  # Save log output to a file
# kubectl logs [POD-NAME] -c [CONTAINER-NAME]  # dump pod container logs (stdout, multi-
container case)
# kubectl logs [POD-NAME] --all-containers=true  # dump logs of all containers inside nginx pod
# kubectl logs -l [KEY]=[VALUE]              # dump pod logs, with label  (stdout)
```

Display Container Logs using "docker logs" command from respective worker node

```
kubectl get pods -o wide
docker ps | grep [KEY-WORD]
docker logs [CONTAINER-ID]
```

Using Journalct
journalctl #Display all messages journalctl -r #Display newest log entries first (Latest to Old order) journalctl -f #Enable follow mode & display new messages as they come in journalctl –n 3 #Display specific number of RECENT log entries journalctl –p crit #Display specific priority – “info”, “warning”, “err”, “crit”, “alert”, “emerg” journalctl –u docker #Display log entries of only specific systemd unit journalctl –o verbose #Format output in “verbose”, “short”, “json” and more journalctl –n 3 –p crit #Combining options journalctl --since "2019-02-02 20:30:00" --until "2019-03-31 12:00:00" # Display all messages between specific duration

K8s Cluster Component Logs
journalctl –u docker journalctl –u kubelet

If K8s Cluster Configured using "kubeadm
kubectl logs kube-apiserver-master -n kube-system | more kubectl logs kube-controller-manager-master -n kube-system kubectl logs kube-scheduler-master -n kube-system kubectl logs etcd-master -n kube-system

If K8s Cluster Configured using "Hard-way (Manual)"
journalctl –u kube-apiserver journalctl –u kube-scheduler journalctl –u etcd journalctl –u kube-controller-manager