Kubernetes

Kubernetes, or K8s, is an open-source tool that helps run and manage apps packaged in containers. It handles tasks like starting, stopping, and scaling these containers automatically. Containers are grouped into small units called pods, which make managing them easier.

Key features include:

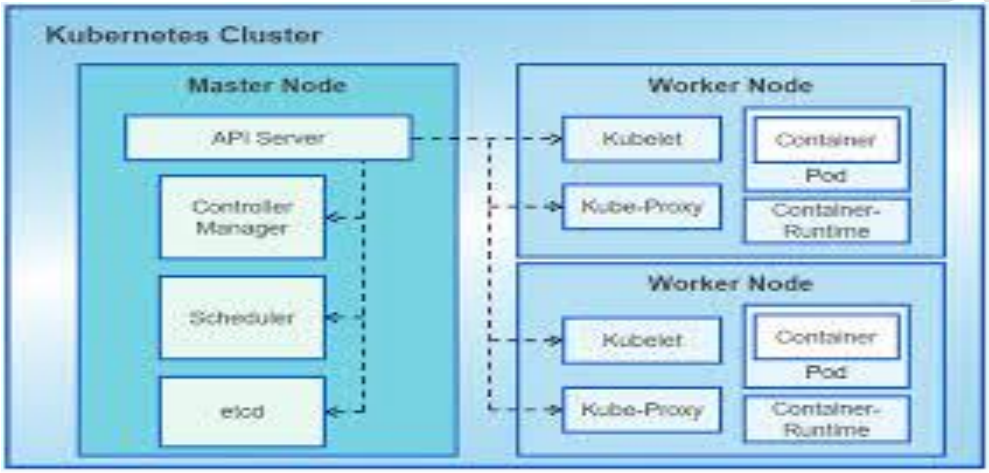
1) **Restart or Recreate Containers (Self-Healing):**
   - Kubernetes automatically restarts containers that fail, terminates unresponsive ones, and replaces containers when nodes fail.
2) **Autoscaling:**
   - Horizontal Pod Autoscaler (HPA): Scales the number of pods based on CPU/memory utilization or custom metrics.
   - Cluster Autoscaler: Adjusts the number of nodes in a cluster based on resource demands.
3) **Load Balancing:**
   - Kubernetes uses Services to distribute network traffic among pods, ensuring application reliability.
   - It provides internal and external load balancing for efficient traffic distribution.
4) **Networking Among Containers Across Hosts:**
   - Kubernetes uses a container network interface (CNI) to enable seamless communication between containers, even if they are on different hosts.
   - Each pod gets its unique IP, enabling pod-to-pod communication.
5) **Create Containers on Different Hosts:**
   - Kubernetes uses schedulers to distribute pods across nodes based on resource requirements, availability, and other constraints.
6) **Deployment of Different Versions of Applications:**
   - Kubernetes Deployments support rolling updates, allowing you to release new application versions without downtime.
7) **Rollback to Previous Versions:**
   - Deployments also enable rollback functionality, letting you revert to a stable previous version of your application if issues arise.
8) **Managing Containers:**
   - Kubernetes simplifies container lifecycle management through pods, replica sets, and deployments.
   - Tools like the kubectl command-line interface and Kubernetes Dashboard provide easy container management.

Kubernetes cluster

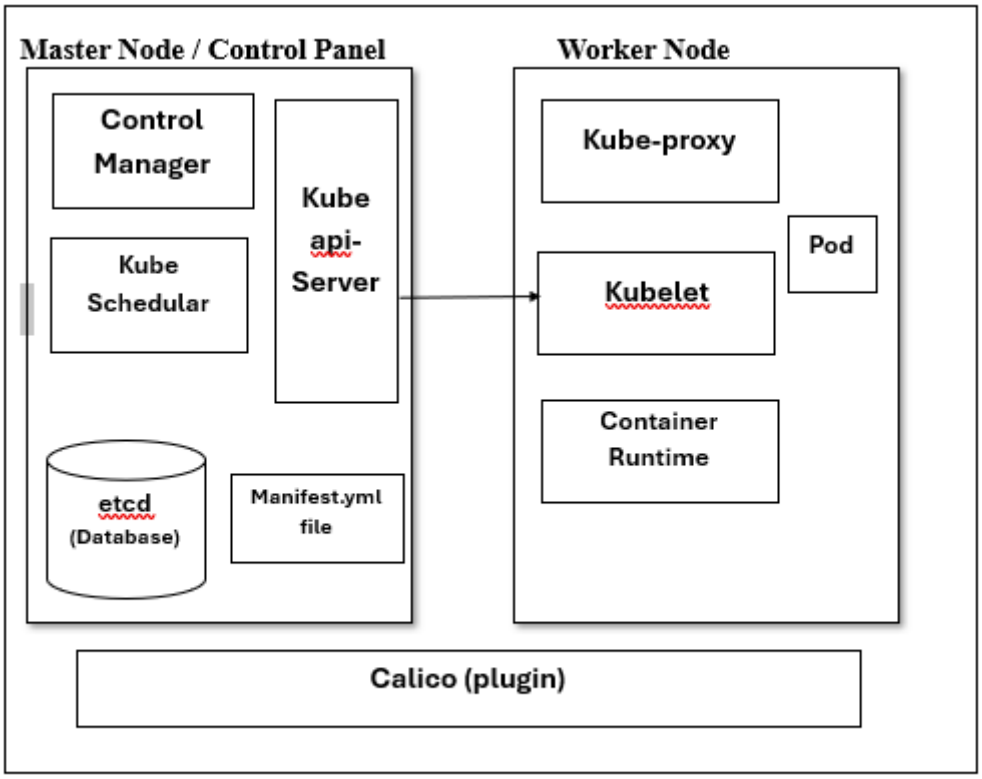
A Kubernetes cluster is a set of machines (nodes) that work together to run containerized applications. It has two main components:

1. **Control Plane:** Manages the cluster, handling tasks like scheduling, scaling, and ensuring apps are running as expected.
2. **Worker Nodes:** Run the actual applications inside containers (grouped in pods).

The cluster enables apps to run reliably across multiple machines while automating tasks like scaling, networking, and updates.



Architecture of K8s



Cluster

Master Node (Control Plane): is the brain

1. **etcd:** A storage system that holds all the cluster's data, like configurations and statuses.
2. **Kube api-server:** The entry point for interacting with the cluster through APIs.
3. **kube-scheduler:** Decides where to place new pods based on available resources.
4. **Controller Manager:** Manages different controllers to ensure the cluster stays in the desired state.
5. **Manifest Files:** Configuration files (in YAML or JSON) that define how applications and their components (pods, services) should behave.

Worker Node: are the hands

1. **kubelet:** A tool that runs on each worker node to manage containers and make sure pods are running.
2. **Container Runtime:** Software like Docker or containerd that runs and manages containers.
3. **kube-proxy:** Manages network rules and load balancing between services.
4. **Pod(s):** The smallest unit in Kubernetes, which can contain one or more containers.

Calico Plugin:

A tool for managing networking and security policies between pods.

Calico ensures secure and smooth communication between pods.

Kubernetes setup

(Run on both nodes)

1. **Update and install Docker:**
   a. sudo apt-get update
   b. sudo apt-get install docker.io
2. **Install required packages for Kubernetes:**
   a. sudo apt-get install -y apt-transport-https ca-certificates curl gnupg
3. **Add Kubernetes key and repository:**
   a. sudo mkdir /etc/apt/keyrings
   b. curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.30/deb/Release.key | sudo gpg --dearmor -o /etc/apt/keyrings/kubernetes-apt-keyring.gpg
   c. sudo chmod 644 /etc/apt/keyrings/kubernetes-apt-keyring.gpg
4. **Add Kubernetes repository to sources list:**
   a. echo 'deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] https://pkgs.k8s.io/core:/stable:/v1.30/deb/ /' | sudo tee /etc/apt/sources.list.d/kubernetes.list
   b. sudo chmod 644 /etc/apt/sources.list.d/kubernetes.list
5. **Install kubectl, kubeadm, and kubelet:**
   a. sudo apt-get update
   b. sudo apt-get install -y kubectl kubeadm kubelet

On Master

1. **Initialize the Kubernetes control plane:**
   a. sudo kubeadm init --ignore-preflight-errors=all
2. **Configure kubectl for the user:**
   a. mkdir -p $HOME/.kube
   b. sudo cp -i
   c. /etc/kubernetes/admin.conf $HOME/.kube/config
   d. sudo chown $(id -u):$(id -g) $HOME/.kube/config
3. **Install Calico network plugin:**
   a. kubectl apply -f https://raw.githubusercontent.com/projectcalico/calico/v3.26.0/manifests/calico.yaml
4. **Generate the join command for worker nodes:**
   a. kubeadm token create --print-join-command
5. **[add port 6443 in both master and worker node]**

On Worker:

1. **Join the cluster using the token command from the master node:**
   - Use kubeadm join command with token (copy the command from master and paste on worker)
   - **Example**- sudo kubeadm join 172.31.81.224:6443 --token zx8cxk.9jl0lu4rhylqe5qa --discovery-token-ca-cert-hash sha256:e0c52061ecb64af117433a9909ba51ff49e41dcd1bc9dd8bb2f0a309b2e10ad3a57f9

On Master:

1. **Verify that the nodes are connected:**

   kubectl get nodes

```
root@ip-172-31-81-224:/home/ubuntu# kubectl get nodes
NAME                STATUS   ROLES           AGE   VERSION
ip-172-31-81-224    Ready    control-plane   14h   v1.30.7
ip-172-31-84-244    Ready    <none>          14h   v1.30.7
root@ip-172-31-81-224:/home/ubuntu#
```

Kubernetes Commands

1) **List all Pods in the current namespace:**
   - kubectl get pods # or kubectl get pod or kubectl get po
2) **Create a Pod using the specified image:**
   - kubectl run pod-name --image=nginx
3) **Delete a specific Pod:**
   - kubectl delete pod pod-name
4) **Delete all Pods in the current namespace:**
   - kubectl delete pod --all
5) **List Pods with additional details:**
   - kubectl get pod -o wide
6) **Access a Pod interactively:**
   - kubectl exec -it mynginx -- /bin/bash
7) **Create Nginx Pod:**
   - **nano manifest.yml**

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod1
spec:
  containers:
   - image: nginx
     ports:
      - containerPort: 80
     name: mynginx
```

   - **kubectl apply -f manifest.yml**
   - **kubectl exec -it mynginx – bin/bash**
8) **Create Mysql Pod:**
   - **nano myfile.yml**

```
apiVersion: v1
kind: Pod
metadata:
  name: mydbpod
spec:
  containers:
   - image: mysql
     name: mydbpod
     ports:
      - containerPort: 3306
        hostPort: 3306
     env:
      - name: MYSQL_ROOT_PASSWORD
        value:  Pass@123
      - name:  MYSQL_DATABASE
        value:  Mydb
```

- **kubectl apply -f myfile.yml**
- **kubectl exec -it mynginx – bin/bash**
- **mysql -u root -p** # and add password

Add label

1. **Create nginx pod**
   - **nano label.yml**

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
  labels:
    app: facebook
spec:
  containers:
   - image: nginx
     ports:
      - containerPort: 80
     name: mynginx
```

   - **kubectl apply -f label.yml**
   - **kubectl exec -it podname – bash**
   - **cd /var/www/html**
   - **add html page**
   - **exit**

2) **Forward a port from a Pod to your local machine:**
   - kubectl port-forward podname 8080:80
   - curl pod-ip-address:8080 # or curl localhost:8080

3) **Label a Pod (add or overwrite a label):**
   - kubectl label pod podname env=labelname –overwrite

4) **List all Pods with their labels:**
   - kubectl get pods --show-labels

5) **List Pods with a specific label key:**
   - kubectl get pods -l env --show-labels

6) **List Pods with a specific label key-value pair:**
   - kubectl get pods -l env=testing --show-labels

7) **Add a label to a node:**
   - kubectl label node ip-172-31-0-133 gpu=true

8) **Verify the label:**
   - kubectl describe node ip-172-31-0-133

Node Selector

Adding nodeSelector in the Pod Spec

- **Add a label to a node:**

  kubectl label node ip-172-31-0-133 gpu=true

- **Nano nodeselector.yml**

```
apiVersion: v1
kind: Pod
metadata:
  name: myngpod
  labels:
    app: facebook
spec:
  nodeSelector:
     gpu: "true"
  containers:
    - image: nginx
      name: mynginx
      ports:
       - containerPort: 80
```

- **kubectl apply -f nodeselector.yml**

Namespace

1) **List all namespaces in the cluster:**
   - Kubectl get ns
2) **Create a new namespace:**
   - kubectl create ns nsname
3) **List Pods in a specific namespace:**
   - kubectl get pods -n nsname
4) **Adding nodespace in the Pod Spec**
   - nano namespace.yml

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

   - kubectl apply -f namespace.yml
   - kubectl get pods -n myns

5) **Adding nodespace through command**
   - Nano filename.yml

```
apiVersion: v1
kind: Pod
metadata:
  name: mydbpod
spec:
  containers:
    - name: mydb
      image: mysql
      ports:
        - containerPort: 3306
      env:
        - name: MYSQL_ROOT_PASSWORD
          value: pass123
        - name: MYSQL_DATABASE
          value: mydb
```

   - kubectl apply -f filename.yml -n nsname
   - kubectl get pods -n nsname
6) **Set the default namespace:**
   - kubectl config set-context –current –namespace=nsname
7) **To revert back to the default namespace:**
   - kubectl config set-context --current --namespace=default

Replication-Controller

1) **Adding ReplicationController in the Pod Spec**
   - nano replication.yml

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: mypod
  labels:
    app: myapp
spec:
  replicas: 2
  selector:
    app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: mycontainer
        image: nginx
        ports:
         - containerPort: 80
```

   - kubectl apply -f replication.yml
   - kubectl get pods

2. **through command**
   - kubectl scale rc podname --replicas=20
   - kubectl get pods
3. **Force delete a ReplicationController:**
   - kubectl delete -f replication.yml –force
4. **List all ReplicationControllers in the cluster:**
   - kubectl get rc
5. **Watch the status of ReplicationControllers:**
   - kubectl get rc –watch
6. **View detailed information about a specific ReplicationController:**
   - kubectl describe rc <rc-name>

Health check Probes

Health check probes in Kubernetes are like regular "checkups" for your containers. They make sure the application inside a container is working properly and can handle requests. If something goes wrong, Kubernetes can restart the container or stop sending traffic to it until it recovers.

Types of Probes:

1. **Liveness Probe**
   - **Purpose:** Checks if the container is still working. If it isn't, Kubernetes restarts it.
   - **Use Case:** Helps fix containers that are stuck or not working anymore.
2. **Readiness Probe**
   - **Purpose:** Checks if the container is ready to take requests. If it isn't, Kubernetes stops sending traffic to it.
   - **Use Case:** Makes sure only fully ready containers handle requests.
3. **Startup Probe**
   - **Purpose:** Checks if the application inside the container has started. If it doesn't start, Kubernetes restarts it. After it starts, other probes (Liveness and Readiness) take over.
   - **Use Case:** Useful for apps that take a long time to start.

Ways to Set Up Probes:

1. **HTTP Probe:** Sends an HTTP request (like opening a webpage) to check if the container is working.
2. **TCP Probe:** Tries to connect to the container on a specific port to see if it's responding.
3. **Command Probe (Exec):** Runs a command inside the container. If the command succeeds, the container is healthy.

Liveness Probe

Examples

1. HTTP-Based Probes

```
livenessProbe:
  httpGet:
    path: /index.html
    port: 80
  initialDelaySeconds: 3
  periodSeconds: 5
  failureThreshold: 4
```

- **httpGet:** Specifies the URL path and port to check if the container is healthy (like visiting a webpage).
- **initialDelaySeconds:** How long to wait after the container starts before starting the health check.

- **periodSeconds:** How often the health check is performed (time between checks).

2. TCP-Based Probes

```
livenessProbe:
  tcpSocket:
    port: 3306
  initialDelaySeconds: 5
  periodSeconds: 10
  failureThreshold: 4
```

- **tcpSocket**: Establishes a TCP connection on port 3306.

3. Command (Exec) Probes livenessProbe:

```
  exec:
    command:
    - cat
    - /tmp/healthy
  initialDelaySeconds: 5
  periodSeconds: 10
```

- **exec:** Executes a command inside the container. If the command exits with a non-zero status, the probe fails.

Readiness Probe

HTTP-Based Probes

```
readinessProbe:
  httpGet:
    path: /index.html
    port: 80
  initialDelaySeconds: 3
  periodSeconds: 5
  failureThreshold: 4
```

- **httpGet:** Specifies the URL path and port to check if the container is healthy (like visiting a webpage).
- **initialDelaySeconds:** How long to wait after the container starts before starting the health check.
- **periodSeconds:** How often the health check is performed (time between checks).

Startup Probe

HTTP-Based Probes

```
startupProbe:
  httpGet:
    path: /index.html
    port: 80
  initialDelaySeconds: 3
  periodSeconds: 5
  failureThreshold: 4
```

- **httpGet:** Specifies the URL path and port to check if the container is healthy (like visiting a webpage).
- **initialDelaySeconds:** How long to wait after the container starts before starting the health check.
- **periodSeconds:** How often the health check is performed (time between checks).

ReplicaSet

- **Purpose:** A ReplicaSet makes sure that a certain number of identical Pods are always running in your cluster. If any Pod stops or crashes, the ReplicaSet will create a new one to keep the count the same.
- **How it works:**
  A **ReplicaSet** makes sure that a certain number of identical Pods are always running. If one Pod crashes or gets deleted, the ReplicaSet automatically creates a new one to keep the number of Pods the same as you set.
- **Use Case:**
  A **ReplicaSet** is used when you want to make sure there are multiple copies of the same Pod running at all times, so if one fails, others can still handle the traffic. You don't need to worry about which node the Pods are running on, just that there are enough copies available.
- **Key Points:**
  - **How many Pods:** You specify the number of Pods you want by setting spec.replicas in the ReplicaSet.
  - **Matching Pods:** The ReplicaSet uses labels to make sure it keeps the correct Pods running.
  - **Used with Deployments:** ReplicaSets are often used together with Deployments to easily update Pods without downtime (e.g., rolling updates).
- **Example YAML:**

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: my-replicaset
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: nginx
        image: nginx
```

  - replicas: 3: Ensures that there are always 3 Pods of nginx running.
- **Lists all the Pods in the current replicaset:**
  - kubectl get pods
- **Lists all ReplicaSets in the current replicaset:**
  - kubectl get rs
- **Provides detailed information about a specific ReplicaSet:**
  - kubectl describe rs myapp-rs

DaemonSet

- **Purpose:** Makes sure a Pod runs on every (or specific) node in the cluster.
- **How it works:**

  A DaemonSet runs one Pod on each node. If a new node is added, it automatically runs a Pod on that node. If a node is removed, the Pod on it is deleted.
- **Use Case:**
  - Logging (collecting logs from all nodes)
  - Monitoring (collecting metrics from all nodes)
  - Network proxies (handling communication between nodes)
- **Key Points:**
  - DaemonSets are useful for system-level services that need to run everywhere.
  - You can specify which nodes the Pods should run on using node selectors.
  - When nodes are added or removed, DaemonSets automatically create or remove Pods accordingly.
- **Example YAML:**

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: mydaemonset
spec:
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: myapp
        image: myapp-image
```

  - **selector**: Ensures that DaemonSet manages Pods with the correct labels.
  - DaemonSet will ensure that one Pod of myapp is running on each node in the cluster.
- **Lists all the Pods in the current Daemonset:**

  kubectl get pods
- **Lists all ReplicaSets in the current Daemonset:**

  kubectl get rs
- **Provides detailed information about a specific Daemonset:**

  kubectl describe rs myapp-rs

Services

ClusterIP:

- A service that exposes your application inside the Kubernetes cluster only. It gives your application a stable internal IP address that other services or pods within the cluster can use to communicate with it.
- **Key Idea:** It's for internal communication within the cluster.
- **Use Case:**
  - Imagine you have a backend service (like a database) that only needs to be accessed by other applications running inside the cluster.
- **How it works:**
  - When you create a ClusterIP service, Kubernetes assigns it a unique internal IP.
  - Other pods in the cluster can use this IP (or a DNS name) to access the service.

Example:

- nano clusterip.yml

```
apiVersion: v1
kind: Service
metadata:
  name: myservice
spec:
  type: ClusterIP
  selector:
    app: dev
  ports:
     port: 80
     targetPort: 80
     nodePort: 31000
```

- kubectl apply -f clusterip.yml
- kubectl get services
- kubectl describe service myservice
- nano test.yml

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: mypod
spec:
   replicas: 3
   template:
     metadata:
       labels:
         app: dev
     spec:
       containers:
        - image: nginx
          name: mycont
          ports:
            - containerPort: 80
```

- kubectl apply -f test.yml
- kubectl get pods
- kubectl describe service myservice
- kubectl exec -it podname – bash
- curl localhost # or curl podip or curl serviceip
- exit

NodePort:

- A service that exposes your application outside the cluster by opening a specific port on each node in the Kubernetes cluster.
- **Key Idea:** It allows external access to your application via <NodeIP>:<NodePort>.
- **Use Case:**
  - Imagine you are testing a small application, and you want to access it directly from your laptop.
- **How it works:**
  - Kubernetes assigns a port in the range 30000-32767 (you can also specify one manually).
  - The service is then accessible using the node's IP address and the assigned port.

Example:

- A web server running inside a pod is exposed on NodePort 31000.
  - Access it via http://<NodeIP>:31000.
  - nano nodeport.yml

```
apiVersion: v1
kind: Service
metadata:
  name: myservice
spec:
  type: NodePort
  selector:
    app: dev
  ports:
     port: 80
     targetPort: 80
     nodePort: 31000
```

  - kubectl apply -f nodeport.yml
  - kubectl get services
  - kubectl describe service myservice
  - nano test.yml

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: mypod
spec:
```

```
  replicas: 3
  template:
    metadata:
      labels:
        app: dev
    spec:
      containers:
       - image: nginx
         name: mycont
         ports:
           - containerPort: 80
```

- kubectl apply -f test.yml
- kubectl get pods
- kubectl describe service myservice
- Access it via http://<workerNodeIP>:31000.

LoadBalancer:

- A service that exposes your application to the internet (or external users) using a cloud provider's load balancer.
- **Key Idea:** It provides an external IP address or DNS name that users can access directly.
- **Use Case:**
  - If you are hosting a public website or API and need to allow internet users to access it.
- **How it works:**
  - Kubernetes communicates with the cloud provider (like AWS, GCP, or Azure).
  - It creates a load balancer for your service and assigns it an external IP or DNS.
  - The load balancer distributes traffic to the pods behind the service.
- **Example:**
  - nano loadbalance.yml

```
apiVersion: v1
kind: Service
metadata:
  name: myservice
spec:
  type: LoadBalancer
  selector:
    app: dev
  ports:
     port: 80
     targetPort: 80
     nodePort: 31000
```

  - kubectl apply -f loadbalance.yml
  - kubectl get services
  - kubectl describe service myservice
  - nano test.yml

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: mypod
spec:
    replicas: 3
    template:
      metadata:
        labels:
          app: dev
      spec:
        containers:
         - image: nginx
           name: mycont
           ports:
             - containerPort: 80
```

- kubectl apply -f test.yml
- kubectl get pods
- kubectl describe service myservice
- Create a Load Balancer in the AWS Management Console, ensure that port 31000 is open in the associated target group, configure the Load Balancer settings as required, and after the creation is complete, copy the DNS name of the Load Balancer and use it in a browser to access the application.

Ingress

- **Ingress Name:** my-ingress
- **Purpose:** Routes traffic based on the host.
- **Rules:**
  - **Host:** example.com
    - Path / routes traffic to Service myservice1 on port 80.
  - **Host:** example1.com
    - Path / routes traffic to Service myservice2 on port 80.
- **PathType:** Prefix ensures traffic matches the specified path and its sub paths.

Example:

- Nano ingress.yml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my ingress
spec:
  rules:
   - host: tujanena.com
     http:
      paths:
       - path: /
         pathType: Prefix
         backend:
          service:
            name: myservice1
            port:
              number: 80
   - host: example.com
     http:
      paths:
       - path: /
         pathType: Prefix
         backend:
          service:
            name: myservice2
            port:
              number: 80
```

- kubectl apply -f ingress.yml
- kubectl get ingress
- kubectl get pods
- kubectl delete ingress myingress

Jobs

A Job in Kubernetes is used to run a one-time task or short-lived workloads that need to complete successfully. Unlike Deployments or Pods that typically run indefinitely, Jobs ensure a task is completed and can retry failed attempts.

One-time Execution: Runs tasks that finish after completion, like data processing or backups.

Retry Mechanism: Retries failed tasks automatically until successful or until a defined limit.

Completion Tracking: Tracks the number of successful completions for specified tasks.

Parallelism: Supports running multiple Pods in parallel for batch processing.

Example:

- nano job.yml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: myjob
spec:
  completions: 3 # The task will run successfully 3 times (total).
  parallelism : 2 # Up to 2 Pods will run the task simultaneously.
  template:
    metadata:
      labels:
        app: myapp
    spec:
      restartPolicy: OnFailure
      containers:
       - name: mycont
         image: luksa/batch-job
```

- kubectl apply job.yml # Creates or updates a Kubernetes Job from the job.yml file.
- kubectl get job # Lists all Jobs in the current namespace.
- kubectl get job – watch # Continuously watches and displays updates to the status of Jobs in real-time.

CronJob

A CronJob in Kubernetes is a way to schedule and automate tasks to run at specific times or intervals, similar to the cron system on Linux. It creates Jobs at scheduled times to perform tasks like backups.

Key Features of CronJob

1. **Automated Scheduling:** Runs tasks automatically at predefined times.
2. **Based on Jobs:** Each scheduled task creates a Kubernetes **Job** to handle the execution.
3. **Retry on Failure:** If a task fails, it can retry depending on the configuration.
4. **Periodic or Time-Based:** Tasks can run every minute, daily, weekly, or any custom interval.

Components of a CronJob

1. **Schedule:**
   - Defined using **cron syntax** (e.g., */5 * * * * to run every 5 minutes).
2. **Job Template:**
   - Specifies what the task will do (container image, commands, etc.).
3. **Pod:**
   - The actual unit where the task runs, created by the Job.

How It Works

1. Define a CronJob with a schedule and task (using YAML).
2. Kubernetes runs the task at the specified schedule.
3. A **Job** is created at each scheduled time to complete the task.
4. Pods are created to execute the work, and they shut down after completion.

Example:

- nano cronjob.yml

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: myjob
spec:
  schedule: "*/5 * * * *"  # run every 5 minutes
  jobTemplate:
    spec:
      template:
        metadata:
          labels:
            app: myapp
        spec:
          restartPolicy: OnFailure
          containers:
            - name: mycont
              image: luksa/batch-job
```

- kubectl apply -f cronjob.yml
- kubectl get pod # Lists all Pods in the current namespace.
- kubectl get cronjob # Lists all CronJobs in the current namespace.
- kubectl describe jobs myjob # Displays detailed information about the specified Job (myjob)

Deployment

A **Deployment** in Kubernetes is a way to manage your app automatically. It makes sure your app is always running the right number of instances, can scale up or down based on demand, can be updated easily without downtime, and will fix itself if something goes wrong.

Example1:

- Nano deployment.yml

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

- kubectl apply -f deployment.yml
- kubectl get pods # List all Pods in the current namespace
- kubectl get deployment / kubectl get deploy # List all Deployments in the current namespace.
- kubectl get deployment mydeployment / kubectl get deployment/mydeployment / kubectl get deploy/mydeployment # Show details of a specific Deployment named mydeployment.
- kubectl set image deployment mydeployment mycont=php # Updates the container named mycont in the mydeployment Deployment to use the php image
- kubectl rollout status deployment mydeployment # Check the progress/status of the Deployment update.
- kubectl rollout history deployment mydeployment # View the update history of the Deployment.
- kubectl get deployment mydeployment -o yaml # Retrieves the detailed configuration of the mydeployment Deployment in YAML format.
- kubectl annotate deployments.apps mydeployment kubernetes.io/change-cause="v1.1" # Adds an annotation to mydeployment with the key kubernetes.io/change-cause and value v1.1, typically to track the version or reason for a deployment change.

Example2:

- nano deployment.yml

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

```
  name: mydeployment
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 5
      maxUnavailable: 2
  selector:
    matchLabels:
      app: test
  template:
    metadata:
      labels:
        app: test
    spec:
      containers:
      - image: nginx
        name: mycont
        ports:
          - containerPort: 80
```

- kubectl apply -f department.yml
- kubectl get deployment
- kubectl set image deploy mydeployment mycont=httpd
- kubectl get deployment mydeployment –watch
- kubectl get pod
- kubectl get deploy/mydeployment
- kubectl rollout undo deploy mydeployment --to-revision=2 # Rolls back the mydeployment Deployment to revision 2, undoing any changes made after that revision.
- kubectl rollout history deployment mydeployment

StatefulSet

- **StatefulSet** in Kubernetes is used to manage stateful applications.
- It ensures **stable network identities** and **persistent storage** for each pod.
- Pods are created with unique names and can be restarted without losing data.
- **Example**
  - Nano statefulset.yml

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mypod3
spec:
  replicas: 3
  serviceName: mypod-service
  selector:
    matchLabels:
      app: dev
  template:
    metadata:
      labels:
        app: dev
    spec:
      containers:
      - image: nginx
        name: mycont
        ports:
        - containerPort: 80
```

  - Kubectl apply -f statefulset.yml
  - Kubectl get pod

Volumes

1) **Empty Directory:**
   - A temporary storage space inside a pod in Kubernetes.
   - Data is wiped out when the pod stops or restarts.
   - Useful for caching or temporary files shared among containers in the same pod.
2) **Git Repo:**
   - A repository for storing code and tracking its changes using Git.
   - Can be hosted on platforms like GitHub, GitLab, or locally.
   - Kubernetes can pull files from a Git repo into a pod if configured.
3) **Host Path:**
   - Mounts a file or directory from the node (host machine) into the pod.
   - Allows pods to access files directly from the host machine.
   - Used cautiously as it tightly couples pods with specific nodes.
4) **NFS (Network File System):**
   - A shared storage system that allows multiple pods or servers to access the same files over the network.
   - Useful for persistent shared storage across pods.
5) **Persistent Volume (PV):**
   - A storage resource in Kubernetes.
   - Can be backed by local disks, cloud storage, or NFS.
   - Provides long-term storage for data that persists beyond the life of a pod.
6) **Persistent Volume Claim (PVC):**
   - A request for storage by a user in Kubernetes.
   - Binds to a PV to provide the requested storage to a pod.
   - Ensures the pod gets the necessary storage resources.
7) **Confirm:**
   - Stores non-sensitive configuration data as key-value pairs.
   - Example: Application settings like URLs or environment variables.
   - Decouples configuration from application code.
8) **Secret:**
   - Stores sensitive data like passwords, API keys, or certificates.
   - Encrypted and more secure than ConfigMaps.
   - Can be injected into pods as environment variables or mounted files.

Empty Directory:

- An empty directory is a temporary storage volume in Kubernetes that is created when a pod is assigned to a node. It's empty when the pod starts.
- How it works:
  - The empty directory is tied to the lifecycle of the pod. If the pod is deleted, the empty directory is also deleted.
  - It can be used for storing temporary data that is shared between containers in the same pod.
- Use Case:
  Suppose you have a pod with multiple containers, and they need to share logs or temporary data. You can use an empty directory for this purpose.
- Key Point:
  The data is not persistent. It's gone when the pod stops or is removed.
- **Example:**
  - Nano empty.yml

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: mypod
spec:
   replicas: 5
   selector:
     matchLabels:
       app: test
   template:
     metadata:
       labels:
        app: test
     spec:
       containers:
       - image: nginx
         name: mycont1
         ports:
           - containerPort: 80
         volumeMounts:
          - name: myvolume
            mountPath: /usr/share/nginx/html/
       - image: nginx
         name: mycont2
         ports:
           - containerPort: 90
         volumeMounts:
          - name: myvolume
            mountPath: /usr/share/nginx/html/
      volumes:
       - name: myvolume
         emptyDir: {}
```

  - kubectl apply -f empty.yml
  - kubectl get rs
  - kubectl describe rs mypod
  - kubectl get pods

Git Repository Volume:

- A Git repository volume in Kubernetes allows you to clone a Git repository and use its contents as a volume in your pod.
- How it works:
  - Kubernetes clones a specified Git repository to a directory inside the pod when the pod starts.
  - The files in the repository are available to the containers in the pod.
- Use Case:
  If your application needs to use configuration files or static content stored in a Git repository, this volume can help.
- Key Point:
  It's deprecated in Kubernetes, as managing code versions and deployments is better handled by CI/CD pipelines.
- **Example:**
  - nano gitrepo.yml

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: mypod
spec:
    replicas: 5
    selector:
      matchLabels:
        app: test
    template:
      metadata:
        labels:
          app: test
      spec:
        containers:
        - image: nginx
          name: mycont1
          ports:
            - containerPort: 80
          volumeMounts:
          - name: myvolume
            mountPath: /usr/share/nginx/html/
        volumes:
        - name: myvolume
          gitRepo:
          repository: https://github.com/vishalLavare/volume.git
          revision: master
          directory: .
```

  - kubectl apply -f gitrepo.yml
  - kubectl get rs
  - kubectl describe rs mypod
  - kubectl get pods

Host Path:

- A host path volume mounts a directory or file from the host node's filesystem into the pod.
- How it works:
  - You specify the path on the host machine that you want to mount.
  - Containers in the pod can then read from or write to this path as if it's a local directory.
- Use Case:
  - Accessing logs stored on the host.
  - Running pods that need access to specific files or devices on the host, like Docker sockets.
- Key Point:
  Be careful when using host paths, as they can make the pod dependent on the specific node.
- **Example:**
  - create a directory on Worker node(Host) – mydata and also create a file 'index.html' in it to check
  - nano hostpath.yml

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: mypod
spec:
    replicas: 3
    selector:
      matchLabels:
        app: test
    template:
      metadata:
        labels:
          app: test
      spec:
        containers:
        - image: nginx
          name: mycont1
          ports:
            - containerPort: 80
          volumeMounts:
           - name: myvolume
             mountPath: /usr/share/nginx/html/
        volumes:
         - name: myvolume
           hostPath:
             path: /home/ubuntu/mydata
             type: Directory
```

  - kubectl apply -f hostpath.yml
  - kubectl get pods
  - kubectl exec -it pod-name -- /bin/bash

NFS (Network File System):

- NFS is a distributed file system protocol that allows you to share directories across a network. In Kubernetes, you can use an NFS volume to share data between multiple pods.
- How it works:
  - An NFS server hosts the shared directory.
  - Kubernetes pods mount the shared directory using the NFS protocol.
  - Any changes made by one pod are visible to all other pods using the same NFS volume.
- Use Case:
  If you need persistent shared storage across pods (e.g., for storing user-uploaded files in a web application).
- Key Point:
  NFS is persistent and can be used for sharing data between pods across different nodes.
- **Example:**
  - create an efs
  - add sg of worker node in sg of efs at protocol NFS
  - create a directory on WN as myvolume
  - sudo apt-get update
  - sudo apt-get install nfs-common -y
  - copy mount command from efs → attach
  - run that command on WN with replacing efs from command - ~/myvolume
  - eg.: sudo mount -t nfs4 -o nfsvers=4.1,rsize=1048576,wsize=1048576,hard,timeo=600,retrans=2,noresvport fs-04afa8ea9234b32d1.efs.ap-south-1.amazonaws.com:/ ~/myvolume
  - go inside the myvolume and create a directory test
  - sudo mkdir test
  - cd test
  - sudo nano index.html
  - **nano nfs.yml**

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: mypod
spec:
    replicas: 3
    selector:
      matchLabels:
        app: test
    template:
      metadata:
        labels:
          app: test
      spec:
        containers:
        - image: nginx
          name: mycont1
          ports:
            - containerPort: 80
          volumeMounts:
```

```
        - name: myvolume
          mountPath: /usr/share/nginx/html/
      volumes:
       - name: myvolume
         nfs:
          server: fs-04afa8ea9234b32d1.efs.ap-south-1.amazonaws.com
          path: /test
```

- kubectl apply -f nfs.yml
- kubectl get rs
- kubectl describe rs mypod
- kubectl get pods

Persistent Volume (PV):

- A PV is storage provided by an admin in a Kubernetes cluster, representing physical storage (like a disk, NFS share, or cloud storage).
- Think of it as:

  A storage unit in Kubernetes that exists independently of any pod and can be reused.
- **Key Points:**
  - Provides 20Gi of storage from an NFS server (fs-050913897e56d3b25.efs.ap-south-1.amazonaws.com).
  - Uses the path /test on the NFS server.
  - Accessible in ReadWriteOnce (RWO) mode (one node can write at a time).
  - The Retain policy ensures the storage is not deleted automatically after release.

Persistent Volume Claim (PVC):

- A PVC is a request for storage by a pod, specifying how much storage is needed and the access type (e.g., read/write).
- Think of it as:

  A storage request form that pods use to claim a PV.
- **Key Points:**
  - Requests 10Gi of storage from a PV with the label app: dev.
  - Matches the PV named mypv because both share the label app: dev.
  - Binds to a matching PV, enabling pods to use the storage.

Pod Using PVC

- A pod can mount a PVC to access the storage provided by a PV.
- **Key Points:**
  - The pod named mypod runs an nginx container.
  - Mounts the PVC (mypvc) to the container at /usr/share/nginx/html.
  - Allows the container to read and write to the storage provided by the NFS server.

Limitations

1. Access Modes:

   Only one node can write to the volume at a time (due to RWO mode).
2. Storage Requests:

   If the PVC requests more storage than the PV provides, it won't bind.
3. Dependency on NFS:

   Requires a running NFS server. If the server goes down, the storage is unavailable.

Key Takeaways

- PV: Provides external storage like NFS or cloud disks.

- PVC: Requests storage from a PV with matching criteria.
- Pod: Uses a PVC to access the storage.
- Data Persistence: Even if the pod is deleted, data in the PV persists (due to the Retain policy).

Analogy

- PV = A storage locker available to rent.
- PVC = A rental agreement form filled by the tenant (pod) to claim the locker.

Example1: using NFS

- create an efs
- add sg of worker node in sg of efs at protocol NFS
- create a directory on WN as myvolume
- sudo apt-get update
- sudo apt-get install nfs-common -y
- copy mount command from efs → attach
- run that command on WN with replacing efs from command - ~/myvolume
- eg.: sudo mount -t nfs4 -o nfsvers=4.1,rsize=1048576,wsize=1048576,hard,timeo=600,retrans=2,noresvport fs-04afa8ea9234b32d1.efs.ap-south-1.amazonaws.com:/ ~/myvolume
- go inside the myvolume and create a directory test
- sudo mkdir test
- cd test
- sudo nano index.html
- nano pv.yml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mypv
  labels:
    app: dev
spec:
 capacity:
   storage: 20Gi
 accessModes:
    - ReadWriteOnce
 persistentVolumeReclaimPolicy: Retain
 nfs:
  server: fs-050913897e56d3b25.efs.ap-south-1.amazonaws.com
  path: /test
```

- kubectl apply -f pv.yml

- kubectl get pv
- nano pvc.yml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mypvc
  labels:
    app: dev
spec:
 accessModes:
    - ReadWriteOnce
 resources:
  requests:
   storage: 10Gi
 selector:
  matchLabels:
    app: dev
```

- kubectl apply -f pv.yml
- kubectl get pvc
- nano mypod.yml

```
apiVersion: v1
kind: Pod
metadata:
 name: mypod
 labels:
  app: dev
spec:
  containers:
   - image: nginx
     name: mycont
     ports:
      - containerPort: 80
     volumeMounts:
       - name: my-volume
         mountPath: /usr/share/nginx/html
  volumes:
    - name: my-volume
      persistentVolumeClaim:
        claimName: mypvc
```

- kubectl apply -f mypod.yml
- kubectl describe pod mypod

Example2: using HostPath

- create a directory on Worker node(Host) – mydata and also create a file 'index.html' in it to check

- nano pvhostpath.yml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mypv1
  labels:
    app: dev
spec:
 capacity:
   storage: 256Mi
 accessModes:
    - ReadWriteOnce
 persistentVolumeReclaimPolicy: Recycle
 hostPath:
          path: /home/ubuntu/mydata
          type: Directory
```

- kubectl ap ply -f pvhostpath.yml
- kubectl get pv
- nano pvchostpath.yml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mypvc1
  labels:
    app: dev
spec:
 accessModes:
    - ReadWriteOnce
 resources:
  requests:
   storage: 256Mi
 selector:
   matchLabels:
     app: dev
```

- kubectl apply -f pvhostpath.yml
- kubectl get pvc
- nano mypod.yml

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
  labels:
    app: dev
spec:
   containers:
     - image: nginx
       name: mycont
       ports:
        - containerPort: 80
       volumeMounts:
         - name: my-volume
           mountPath: /usr/share/nginx/html
   volumes:
      - name: my-volume
        persistentVolumeClaim:
          claimName: mypvc1
```

- kubectl apply -f mypod.yml
- kubectl describe pod mypod

configMap:

- **ConfigMap**: It is a way to store configuration data separately from the application code.
- **Purpose**: By using a ConfigMap, you can make configuration changes without altering the container image or redeploying the application.
- **Flexibility**: Update configuration without rebuilding your container.
- **Separation of Concerns:** Keep configurations separate from the application code.
- **Dynamic Updates**: ConfigMaps can be updated, and the new values can be picked up by the application (if it supports live reload).

configMap for using command:

- kubectl create configmap mysql-pass --from-literal MYSQL_ROOT_PASSWORD=Pass123
- kubectl get configMap
- kubectl get cm
- kubectl describe cm mysql-pass
- nano configpod.yml

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
   containers:
     - image: mysql
       name: mycont
       ports:
        - containerPort: 3306
       env:
        - name: MYSQL_ROOT_PASSWORD
          valueFrom:
            configMapKeyRef:
              name: mysql-pass
              key: MYSQL_ROOT_PASSWORD
```

- kubectl apply -f configpod.yml
- kubectl get pods

configMap for using a file:

- nano configmap1.yml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: myconfig
data:
  MYSQL_ROOT_PASSWORD: Pass@123
  MYSQL_DATABASE: mydb
```

- kubectl apply -f configmap1.yml
- kubectl get configMap
- kubectl get cm
- kubectl describe cm myconfig
- nano configpod1.yml

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod1
spec:
```

```
  containers:
    - image: mysql
      name: mycont
      ports:
       - containerPort: 3306
      env:
       - name: MYSQL_ROOT_PASSWORD
         valueFrom:
           configMapKeyRef:
             name: myconfig
             key: MYSQL_ROOT_PASSWORD
       - name: MYSQL_DATABASE
         valueFrom:
           configMapKeyRef:
             name: myconfig
             key: MYSQL_DATABASE
```

- kubectl apply -f configpod1.yml
- kubectl get pods

Secret:

1. Generic Secret Volume

- This is a general-purpose secret created and mounted into a pod.
- It stores arbitrary data (e.g., passwords, API tokens) that your application might need.
- Secrets are referenced in the pod's volume configuration and mounted as files.
- **For Using Commands**
  - kubectl create secret generic mysecret --from-literal MYSQL_ROOT_PASSWORD=Pass@123
  - kubectl get secret
  - kubectl describe secret mysecret
  - kubectl get secret mysecret -o yaml
  - nano secret_pod1.yml

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
   containers:
     - image: mysql
       name: mycont
       ports:
        - containerPort: 3306
       env:
        - name: MYSQL_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysecret
              key: MYSQL_ROOT_PASSWORD
```

  - kubectl apply -f secret_pod1
  - kubectl get pods
- **Using yml file**
  - Nano secret.yml

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret1
data:
  MYSQL_ROOT_PASSWORD: U3dhdGlAMTIz  # password add encrypted
type:
  Opaque
```

  - kubectl apply -f secret.yml
  - kubectl get secret
  - nano secret_pod2.yml

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod1
spec:
   containers:
     - image: mysql
```

```
    name: mycont
    ports:
     - containerPort: 3306
    env:
     - name: MYSQL_ROOT_PASSWORD
       valueFrom:
         secretKeyRef:
           name: mysecret1
           key: MYSQL_ROOT_PASSWORD
```

- kubectl apply -f secret_pod2

2. Docker Registry Secret

- Used to authenticate with private Docker registries for pulling container images.
- Created using kubectl create secret docker-registry.
- Referenced in the pod's imagePullSecrets section.
- **For Using Commands**
  - kubectl create secret docker-registry mydockersecret --docker-username=vishal09 --docker-password=dckr_pat_HpJNGtjgdJj3edlstb8dqdnJ6wsdAyo --docker-server=hub.docker.com --docker-email=vishal@gmail.com
  - kubectl get secret
  - kubectl describe secret mydockersecret
  - kubectl get secret mydockersecret-o yaml
  - nano mydockersecret_pod1.yml

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - name: mycont
      image: mysql
      ports:
        - containerPort: 3306
      env:
        - name: MYSQL_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mydockersecret
              key: MYSQL_ROOT_PASSWORD
  imagePullSecrets:
    - name: mydockersecret
```

  - kubectl apply -f mydockersecret_pod1
  - kubectl get pods
- **Using yml file**
  - nano mydockersecret1.yml

```
apiVersion: v1
kind: Secret
metadata:
  name: mydockersecret1
```

```
  namespace: default
data:
  .dockerconfigjson:
eyJhdXRocyI6eyJodHRwczovL2luZGV4LmRvY2tlci5pby92MS8iOnsicGFzc3dvcmQiOiJkY2tyX3BhdF9IcEpOR3RqZ2RKajNkPiJ9fQ==
type:
  kubernetes.io/dockerconfigjson
```

- kubectl apply -f mydockersecret1.yml
- kubectl get secret
- nano mydockersecret_pod2.yml

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod1
spec:
  containers:
    - name: mycont
      image: mysql
      ports:
        - containerPort: 3306
      env:
        - name: MYSQL_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mydockersecret1
              key: MYSQL_ROOT_PASSWORD
  imagePullSecrets:
    - name: mydockersecret1
```

- kubectl apply -f mydockersecret_pod2

3. TLS Secret

- Specifically for storing TLS certificates and keys.
- Used for securing communication (e.g., HTTPS).
- Created using kubectl create secret tls.
- to create key
  - openssl genrsa -out mykey.key 2048
- to create certificate
  - openssl req -new -x509 -key mykey.key -out mycert.crt -days 365 -subj /CN=oleoleapp.in
    or
  - openssl req -new -x509 -key mykey.key -out mycert.crt -days 365 -subj "/CN=oleoleapp.in"

- kubectl get secrets
- kubectl describe secret my-tls-secret
- nano tls_pod.yml

```
apiVersion: v1
kind: Pod
metadata:
  name: tls-pod
```

```
spec:
  containers:
    - name: nginx
      image: nginx
      ports:
        - containerPort: 443
      volumeMounts:
        - name: tls-volume
          mountPath: "/etc/tls"
          readOnly: true
  volumes:
    - name: tls-volume
      secret:
        secretName: mytls-secret
```

- kubectl apply -f tls_pod.yml
- kubectl get pods

HPA

1. **Definition**:
   - HPA stands for **Horizontal Pod Autoscaler** in Kubernetes.
   - It automatically adjusts the number of pods in a deployment or replica set.
2. **Purpose**:
   - Ensures your application has enough resources to handle traffic.
   - Prevents resource wastage during low traffic.
3. **How it Works**:
   - **Monitors Metrics**: Continuously checks metrics like CPU usage, memory usage, or custom metrics.
   - **Decides Scaling**: Adds or removes pods based on defined thresholds for these metrics.
   - **Applies Changes**: Automatically increases or decreases the number of pods in real-time.
4. **Benefits**:
   - Handles traffic spikes by scaling up (adding more pods).
   - Reduces costs by scaling down (removing unused pods) during low activity.
5. **Example**:
   - If CPU usage exceeds 80%, HPA scales up by adding more pods.
   - If CPU usage drops below 50%, HPA scales down by reducing the number of pods.
6. **Usage**:
   - Commonly used to ensure applications remain responsive and cost-efficient.
   - Helps manage workloads dynamically without manual intervention.
7. **HorizentalPodAutoscaling**
   - Required metrics server install
   - Connect Master server

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

   - nano metrics-server-components.yaml
   - (Able to see file)—in this file add arg for tls

```
  metadata:
    labels:
      k8s-app: metrics-server
  spec:
    containers:
    - args:
      - --kubelet-insecure-tls
      - --cert-dir=/tmp
      - --secure-port=10250
      - --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname
      - --kubelet-use-node-status-port
      - --metric-resolution=15s
      image: registry.k8s.io/metrics-server/metrics-server:v0.7.0
      imagePullPolicy: IfNotPresent
      livenessProbe:
        failureThreshold: 3
        httpGet:
          path: /livez
-- INSERT --
```

- After add arg save file.
- nano hpa.yml

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: nginx-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: replica
  minReplicas: 3
  maxReplicas: 7
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 50
```

- kubectl get hpa
- nano deployment.yml

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

- kubectl apply –f deployment.yml
- kubectl get pod