10/27/2024

Mastering DevOps with Kubernetes

-By Vaibhav Upare

History:

- google develop an internal system called 'Borg' (later named as omega) to deploy and manage thousands of google application and services on their cluster
- in 2014, google introduce k8s as an open source platform written in Golang and later donated to CNCF (cloud native computing foundation)

container orchestration platform

- play with k8s
- GKS google
- AKS azure
- EKS amazon
- kubernetes
- Docker Swarm
- Apache Mesos
- Dokku

Problem with scaling up the container:

- can't communicate with each other
- autoscaling
- load balancing
- container had to be manage carefully

Terms to know:

- monolithic application: single stone application, every
- Microservice: each task is deploy in diff-2 services, connect with each other via API
- Orchestration tool = container management tool

Kubernetes = k8s

- k8s is an open source container orchestration platform
- It is used to automates deployment , scaling, load balancing, management of containerized applications.
- all top cloud provider support k8s

feature of k8s:

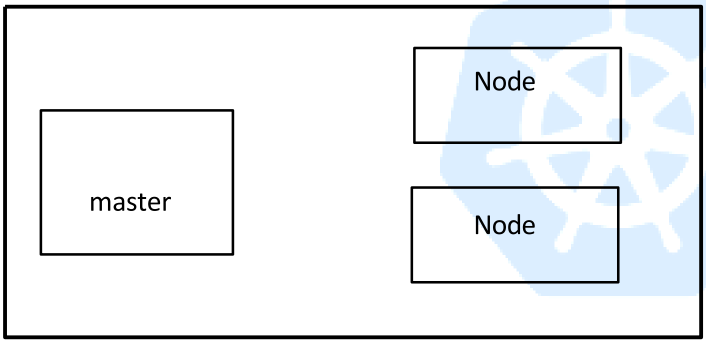
- Orchestration (clustering of any no of cluster running on different n/w)
- Auto scaling or high Performance
- load balancing
- platform independent (cloud / virtual / physical)
- fault tolerance (node / pod failure)
- rollback
- health monitoring of pod
- batch execution
- High Availability or no Downtime

Comparation between k8s and docker swarm:

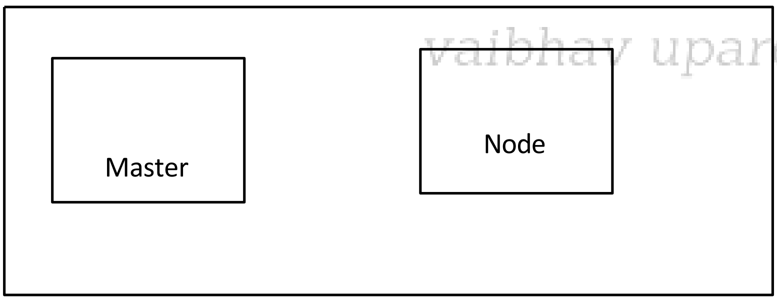
Feature	K8S	Docker Swarm
installation & cluster configuration	Complicated & time consuming	Fast & Easy
Supports	Work with all type of container like Rocket, Docker, ContainerD	Only work with docker
GUI	available	Not Available
Data Volumes	Only shared with containers in same pod	Can be shared with any other container
Update & Rollback	Process schedule to maintain services while updating	Progressive update and Service health monitoring while update
Autoscaling	Available	Not Available
Monitoring	Inbuilt tool	3rd party tool

There are 3 type of Architecture: very high level

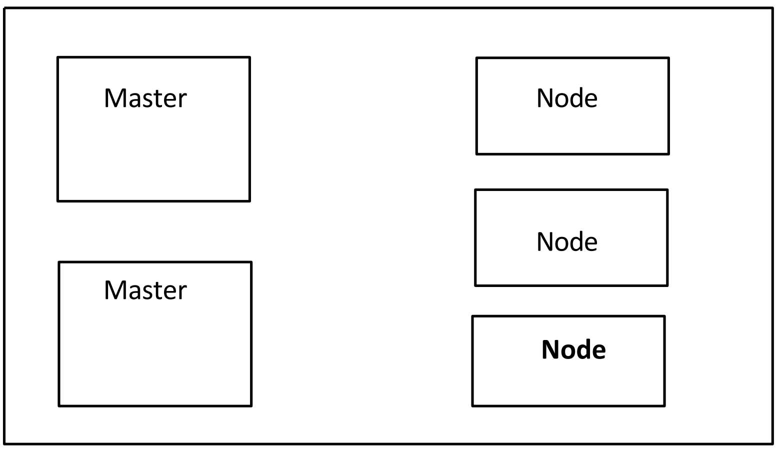
1. **Single master multiple nodes**

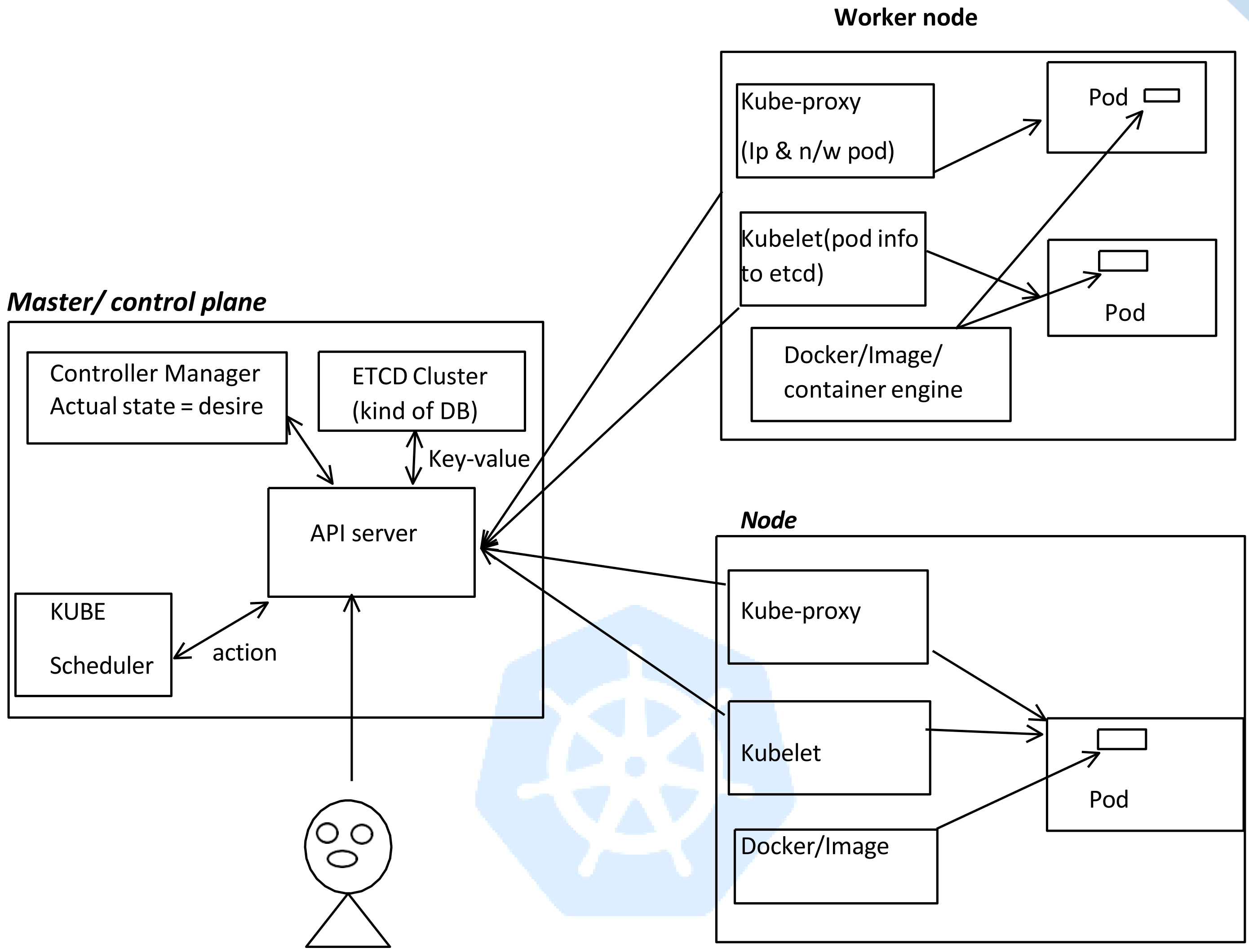


2. **Single Master single node**



3. **Multiple master multiple nodes**





Request flow High Level Diagram

Working:

1. Create the manifest file for k8s objects (json/yml/yaml)
2. Apply these files to cluster (to master) to bring into desired state
3. Pods runs on node which is controlled by master

Role Of Master:

Kubernetes cluster runs on VM/ BareMetal / cloud or mix

1. K8s having one or more master and one or more workers
2. The master is now going to run set of k8s process . These process with insure smooth functioning of master these process are called control plane
3. Can be multi master for high availability
4. Master runs control plane to run cluster smoothly

Components Of master plane:

1. API Server
2. Kube Scheduler
3. Controller Manager
4. ETCD (not part of k8s but without this k8s won't work so consider this also a part of k8s)

1. API Server:

a. API Server front end of kubernetes, provide the interface
b. It meant to scale automatically according to load or request load

2. ETCD:

a.
b. Store metadata or status of
c. cluster Consistent and high availability Store data in key value form

Features:

a. Fully replicated: entire state is available on every node of cluster
b. Secure: implements TLS with optional client-certificate authentication 7
c. Fast: benchmark at 10,000 writes per second

3. Kube Scheduler:

a. It can be assign the newly created pod
b. Check the resource availability of worker node then schedule the pod

4. Controller Manager:

a. Make sure actual state equals to desired state of maniface file
b. If k8s on cloud then "cloud controller manager"
c. If k8s on non-cloud "kube-controller manager"

Controller Components:

a. Node Controller: for checking of nodes that has detect in cloud after it's stop responding.
b. Route Controller: Responsible to setting up n/w, route
c. Service Controller: Responsible for load balancing
d. Volume Controller: Managing Volumes

Components Of Worker Node:

1. Kube Proxy
2. Kubelet
3. Pods
4. Container Engine

1. Kube Proxy:

a. It is responsible for networking and responsible to allocate the IP for pods
b. It's runs on each node
c. It's communicate to master via the API Server

2. Kubelet:

a. Agent running on node
b. Listen the k8s master (pod creation
c. request) Provide pod information to etcd
d. via API Server use port 10255
e. Send success / failure status to control plane

3. Pods:

a. smallest unit of k8s
b. One pod can contains multiple container but recommend only one container in one pod
c. Pod having it's IP address but container don't have
d. Cluster has at least one master node and one worker node
e. K8s cannot start container without pods
f. Auto scaling and auto healing by default not provided by pod for this high level k8s object required
g. Pod crashed is also one more limitation but fix this by high level objects

4. Container Engine:

a. Work with kubelet
b. Pulling image
c. Start/stop container
d. Expose port which is specified in manifest

Important Notes:

a. If we are using single cloud then command will use "kubectl"
b. If we are using on premise then command will use "kubeadm"
c. If we are using on hybrid/federated then command will use "kubefed"
d. https://docs.aws.amazon.com/eks/latest/userguide/what-is-eks.html read this and now easily you can understan

a. **basic objects:** Pods, Service, Volumes, Namespace, etc., which are independent and don't require

i. other objects

b. **high-level objects (controllers):** Deployments, Replication Controllers, ReplicaSets, StatefulSets, Jobs, etc., which are built on top of the basic objects

All Kubernetes Object list :

1. Pod : A thin wrapper around one or more containers
2. Service: Maps a fixed IP address to a logical group of pods
3. Volume: a directory with data that is accessible across multiple containers in a Pod
4. Namespace: a way to organize clusters into virtual sub-clusters
5. ReplicaSets: Ensures a defined number of pods are always running
6. Replica Controller : Ensures a defined number of pods are always running
7. Secrets: an object that contains a small amount of sensitive data such as a password, a tok1en, or a key
8. Config Maps: an API object that lets you store configuration for other objects to use
9. Deployments : Details how to roll out (or roll back) across versions of your application
10. StatefulSets: the workload API object used to manage stateful applications
11. Jobs: Ensures a pod properly runs to completion and stop after process complete it's execution
12. Daemon Sets: Implements a single instance of a pod on all (or filtered subset of) worker node(s)
13. Label: Key/Value pairs used for association and filtering

Relationship Between k8s Objects

- pods maintains container
- ReplicaSets manage pods
- Deployment is a higher-level concept that manages ReplicaSets and provides declarative updates to Pods along with a lot of other useful features
- Service expose the pod process to the outside world
- Config Map and Secrets Both are store the data the same way, with key/value pairs, but ConfigMaps are meant for plain text data, and secrets are meant for data that you don't want anything or anyone to know about except the application
- the replication controller only supports equality-based selectors whereas the replica set supports set- based selectors
- Replica Set is the next generation of Replication Controller. Replication controller is kind of imperative, but replica sets try to be as declarative as possible.

Steps to Install minikube and run pod

Go to Aws account and create & launch instance
Ubuntu18.04 -> t2.medium (minimum 2 CPU required)
Do SSH and after that run following command

1. sudo su
2. Now install docker
   a. sudo apt update && apt -y install docker.io
3. install Kubectl
   a. curl -LO https://storage.googleapis.com/kubern... -s https://storage.googleapis.com/kubern... && chmod +x ./kubectl && sudo mv ./kubectl /usr/local/bin/kubectl
4. install Minikube
   a. curl -Lo minikube https://storage.googleapis.com/miniku... && chmod +x minikube && sudo mv minikube /usr/local/bin/
5. minikube dashboard

Fundamental Of Kubernetes Object Pod:

- A *Pod* is a smallest utile of kubernetes, Representing group of one or more containers, with shared storage and network resources.
- When a Pod creation happen then master will automatically decide that on which node it should create until you have not specified the node.
- Pods remain on node until is not terminated or until node failure not happen , until pod is not deleted, lack of required resource of pod creation
- If node die then after a timeout period pod will also get delete
- If a pod get deleted then same pod (id) cannot restart always start a new pod with different unique ID
- Volume inside pod also die if the pods die
- Controller can manage pod autoscaling, self-healing etc.

Example OF Pod Yaml File:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

- Defaults to Always kubectl apply -f pod1.yml
- List Pods: kubectl get pods
- List Pods wide: kubectl get pods -o wide
- Describe Pod: kubectl describe pod <pod_name>
- Logs of Pod: kubectl logs <pod_name>
- Port Forwarding: kubectl port-forward <pod_name> <local_port>:<remote_port>
- Attach to Pod: kubectl attach <pod_name> -c <container_name>

Pod Example with annotation:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  annotations:
    description: "this is demo"
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

- Defaults to Always kubectl apply -f pod1.yml

Pod Example with multiple containers:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: container1
    image: ubuntu
    command: ["/bin/bash", "-c", "while true; do echo c1; sleep 5 ; done"]
  - name: container2
    image: nginx
    Ports:
        -containerPort: 80
```

Pod Example with environment variable

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: demo
spec:
  containers:
  - name: container1
    image: ubuntu
    command: ["/bin/bash", "-c", "while true; do echo c1; sleep 5 ; done"]
    env:
    - name: myname
      value: vaibhav
```

- Defaults to Always kubectl apply -f pod1.yml

Create a pod from yaml file

The resource same name already exits, create command will return an error.
Command: Kubectl create -f filename.yaml

Update a pod from yaml file

Create command always create new resource and Apply can create or update an existing resources.
Command: Kubectl apply -f filename.yaml

Those are two different approaches:

1. **Imperative Management**
   [Imperative Management](#) create Kubernetes resource direct command line **kubectl create** you want to create, replace or delete

2. **Declarative Management**
   [Declarative Management](#) define the resource within yaml file and then **kubectl apply**

Show log Of a running Pod:

- kubectl logs my-pod #dump pod logs (stdout)
- *kubectl logs -l name=myLabel # dump pod logs, with label name=myLabel (stdout)*
- *kubectl logs my-pod -c my-container# dump pod container logs (stdout, multi-container case)*
  kubectl logs -l name=myLabel -c my-container # dump pod logs, with label name=myLabel (stdout)

- *kubectl logs -f my-pod # stream pod logs (stdout)*
- *kubectl logs -f my-pod -c my-container # stream pod container logs (stdout, multi-container case)*

- *kubectl logs -f -l name=myLabel --all-containers # stream all pods logs with label name=myLabel (stdout)*

- *kubectl logs my-pod --previous # dump pod logs (stdout) for a previous instantiation of a container*
- *kubectl logs my-pod -c my-container --previous # dump pod container logs (stdout, multi-container case) for a previous instantiation of a container*

Labels:

1. **labels** are key-value pairs attached to resources (such as Pods, Deployments, and Services) to organize, select, and manage them.Labels can be attached to objects at creation time
2. Labels are similar to tag in AWS and GIT
3. Labels help in filtering kubernetes resource

Valid label value:

- must be 63 characters or less (can be empty),
- unless empty, must begin and end with an alphanumeric character ([a-z0-9A-Z]),
- could contain dashes (-), underscores (_), dots (.), and alphanumeric between.

Apply label on pod and remove imperative method

a. This command displays the current labels of the pod **Kubectl label pod my-pod --show-labels**
b. This command adds or updates the label `app=dev` on the pod **Kubectl label pod my-pod app=dev**
c. This command removes the `app` label from the pod **Kubectl label pod my-pod app -**

Example of Declarative :

```yaml
apiVersion: v1
kind: Pod
metadata:
 name: my-pod
 labels:
    app: dev
spec:
 containers:
 - name: nginx
  image: nginx:1.14.2
  ports:
  - containerPort: 80
```

Note: *there is 3 way to delete an object*

- *From yaml file*
- *Kubectl delete object object_name*
- *Kubectl delete object -l env=dev*

Selector:

selector is a tool used to filter and identify a specific set of resources based on their labels. Labels are simply key-value pairs attached to Kubernetes resources (like pods, deployments, services)

Types of Selectors

Equality-Based Selectors: These selectors use = or != operators to match resources with specific label values.

- kubectl get pods -l app=myapp

This command retrieves all pods with the label app=myapp.

- kubectl get pods -l env!=prod

This command retrieves all pods that do *not* have the label env=prod.

Set-Based Selectors: These selectors allow matching resources based on inclusion in or exclusion from a set of values, using operators like `in`, `notin`, and `exists`.

- kubectl get pods -l 'env in (dev, staging)'

This command retrieves pods that have the label env with values either dev or staging.

- kubectl get pods -l 'env notin (prod, test)'

This command retrieves pods that do not have env set to prod or test.

- kubectl get pods -l 'app'

This command retrieves pods that have the app label, regardless of the value

1. You can constrain a Pod so that it can only run on particular set of node(s).
2. You can use any of the following methods to choose where Kubernetes schedules specific Pods:
   a. **Node-Selector**
   b. **Nodename**

NodeSelector:

-

- **Node Labels**: kubectl label nodes node-1 app=v1
- kubectl get node –show-labels

You can add the nodeSelector field to your Pod specification and specify the node labels you want the target node to have. Kubernetes only schedules the Pod onto nodes that have each of the labels you specify.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
  nodeSelector:
    app: v1
```

NodeName:

`nodeName` field in a pod specification is used to directly assign a specific pod to a particular node.
Example:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
  nodeName: node0
```

Replication controller:

- Replication Controller in Kubernetes is a resource used to ensure a specific number of pod replicas are running at all times. If pods or nodes fail, the Replication Controller automatically replaces them to maintain the desired state.
- Replication controller version is V1
- Support the rolling update but cant rollout, rollback
- It is a equality base selector only support one label

Example:

```yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx-controller
spec:
  replicas: 3
  selector:
    app: rc-pod
  template:
    metadata:
      name: rc-pod
      labels:
        app: rc-pod
    spec:
      containers:
      - name: rc-pod
        image: ubuntu
        command: ["/bin/bash", "-c", "while true; do echo c1; sleep 5 ; done"]
```


Interactive method to scaleup and scale down number of pods from RC

Command: *kubectl scale rc nginx-controller --replicas=3*

`--cascade=false` option is used in Kubernetes when you delete a resource like a **ReplicationController**. By default, when you delete a resource (such as a ReplicationController), Kubernetes also deletes any dependent resources that are associated with it, such as **pods**. This is called a "cascading delete."

- kubectl delete rc <replication-controller-name> --cascade=fals

ReplicaSets:

- Replica Sets is a next generation Replica Controller
- Replica Sets version is apps/v1
- The Replica Sets supports equality based selector and set based selector
- does not rolling-update future

How a ReplicaSet works:

- In replicasets there is a lot of properties which helps replicasets to work properly.
- There is selector properties which helps replicasets to recreate the pods based on label when pod failure happen, also helped to group the same label pods.
- There is one more properties called template that help RS to create pod according to the given pod template
- Replicas properties is used to provide the number of desired pods to create

Example:

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: book
    tier: frontend
spec:
  replicas: 3      # Modify the number of replicas as per your requirement
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
      - name: php-redis
        image: gcr.io/google_samples/gb-frontend:v3
```


Deployments:

- Deployment version is apps/v1
- The Deployment supports equality based selector and set based selector
- Support rolling-update rollback future
- **Self-Healing**: If a pod fails or is deleted, the Deployment ensures that a new pod is created to replace it and maintain the desired number of replicas.

imperative way to create a **Deployment** in Kubernetes.

- kubectl create deployment my-deploy --image=nginx
- kubectl scale deployment my-deploy --replicas=3
- kubectl get deployments
- kubectl delete deployment my-deploy

Failed Deployment:

Your Deployment may get stuck trying to deploy its newest ReplicaSet without ever completing. This can occur due to some of the following factors:

- Insufficient quota
- Readiness probe failures
- Image pull errors
- Insufficient permissions
- Limit ranges
- Application runtime misconfiguration

Example:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: my-deployment
 labels:
  app: nginx
spec:
 replicas: 3
 selector:
  matchLabels:
   app: nginx
 template:
  metadata:
   labels:
    app: nginx
  spec:
   containers:
   - name: nginx
    image: nginx:1.16.1
    ports:
    - containerPort: 80
```


Updating a Deployment

Follow the steps given below to update your Deployment:

1. Let's update the nginx Pods to use the nginx:1.16.1 image instead of the nginx:1.14.2 image
   - kubectl set image deployment/my-deployment <container_name>=nginx:1.14.2

2. See the rollout status, run: *kubectl rollout status deployment/my-deployment*

3. Run *kubectl get rs* to see that the Deployment updated the Pods by creating a new ReplicaSet and scaling it up to 3 replicas, as well as scaling down the old ReplicaSet to 0 replicas

Checking Rollout History of a Deployment

Follow the steps given below to check the rollout history:

1. First, check the revisions of this Deployment:
   - *kubectl rollout history deployment/my-deployment*

2. To see the details of each revision, run:
   - *kubectl rollout history deployment/my-deployment --revision=2*

Rolling Back to a Previous Revision

Follow the steps given below to rollback the Deployment from the current version to the previous version, which is version 2.

1. Now you've decided to undo the current rollout and rollback to the previous revision:
   - *kubectl rollout undo deployment/my-deployment*

   Alternatively, you can rollback to a specific revision by specifying it with --to-revision:
   - *kubectl rollout undo deployment/my-deployment --to-revision=2*
2. Check if the rollback was successful and the Deployment is running as expected, run:
   - *kubectl get deployment my-deployment*
3. Get the description of the Deployment:
   - *kubectl describe deployment my-deployment*

Scaling a Deployment

1. You can scale a Deployment by using the following command:
   - *kubectl scale deploymentmy-deployment --replicas=10*
2. Assuming horizontal Pod autoscaling is enabled in your cluster, you can setup an autoscaler for your Deployment and choose the minimum and maximum number of Pods you want to run based on the CPU utilization of your existing Pods.
   - *kubectl autoscale deployment/my-deployment --min=10 --max=15 --cpu-percent=80*
3. Proportional scaling

   RollingUpdate Deployments support running multiple versions of an application at the same time. When you or an autoscaler scales a RollingUpdate Deployment that is in the middle of a rollout (either in progress or paused), the Deployment controller balances the additional replicas in the existing active ReplicaSets (ReplicaSets with Pods) in order to mitigate risk. This is called *proportional scaling*.

Pausing and Resuming a rollout of a Deployment

When you update a Deployment, or plan to, you can pause rollouts for that Deployment before you trigger one or more updates. When you're ready to apply those changes, you resume rollouts for the Deployment. This approach allows you to apply multiple fixes in between pausing and resuming without triggering unnecessary rollouts

1. Get the Deployment details:

- ➢ *kubectl get deploy*

2. Pause by running the following command:
   - ➢ *kubectl rollout pause deployment/my-deployment*
3. Then update the image of the Deployment:
   - ➢ *kubectl set image deployment/my-deployment nginx=nginx:1.14.2*
4. Notice that no new rollout started:
   - ➢ *kubectl rollout history deployment/my-deployment*
5. Get the rollout status to verify that the existing ReplicaSet has not changed: kubectl get rs
6. You can make as many updates as you wish, for example, update the resources that will be used:
   - ➢ *kubectl set resources deployment/my-deployment -c=nginx --limits=cpu=200m,memory=512Mi*
7. Eventually, resume the Deployment rollout and observe a new ReplicaSet coming up with all the new updates:
   - ➢ *kubectl rollout resume deployment/my-deployment*
8. Watch the status of the rollout until it's done.
   - ➢ *kubectl get rs -w*

StatefulSet:

- StatefulSet are kubernetes resource that allow us to deploy and manage stateful applications.
- **Stable Network Identity**: Each pod gets a unique DNS name.
- **Persistent Storage**: Retains storage across pod restarts using PersistentVolumeClaims (PVCs).
- **Ordered Operations**: Ensures deployment, scaling, and deletion happen in a specific sequence.
- **Headless Service**: Required for stable DNS (use `clusterIP: None`).
- Pods are added or removed sequentially (e.g., Pod-0, Pod-1, Pod-2).
- Each pod gets its own **PersistentVolumeClaim (PVC)**, independent of other pods. Data is not lost even if a pod restarts or reschedules.
- PVCs are **not deleted** automatically (must be removed manually).
- Databases (e.g., MySQL, PostgreSQL).

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  selector:
    matchLabels:
      app: nginx  # must match .spec.template.metadata.labels
  serviceName: "nginx"
  replicas: 3  # By default, replicas is 1
  minReadySeconds: 10  # By default, it's 0
  template:
    metadata:
```

```
      labels:
        app: nginx  # must match .spec.selector.matchLabels
    spec:
      terminationGracePeriodSeconds: 10
      containers:
      - name: nginx
        image: k8s.gcr.io/nginx-slim:0.8
        ports:
        - containerPort: 80
      volumeMounts:
      - name: www
        mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: www
    spec:
      accessModes: ["ReadWriteOnce"]
      storageClassName: "my-storage-class"
      resources:
        requests:
          storage: 1Gi
```

Cluster Networking:

Networking is a central part of Kubernetes, but it can be challenging to understand exactly how it is expected to work. There are 4 distinct networking problems to address:

- Highly-coupled container-to-container communications: this is solved by Pods and localhost communications.
- Pod-to-Pod communications.
- Pod-to-Service communications.
- External-to-Service communications.

1. **Container to container communication inside pod Example:**

```
apiVersion: v1
kind: Pod
metadata:
  name: testpod
spec:
  containers:
  - name: c0
    image: ubuntu
    command: ["/bin/bash", "-c", "while true; do echo Hello-sagar; sleep 5; done"]
  - name: c1
    image: httpd
    ports:
    - containerPort: 80
```

Commands:

- Kubectl apply -f filename.yaml
- kubectl logs -f testpod -c abc
- kubectl exec -it testpod -c abc /bin/bash
- Apt update && apt install curl
- Curl localhost:80
- Now it will show the output of application which is running inside 2nd containers

2. ***Pod to Pod Communication within same node Example:***

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: testpod1
spec:
  containers:
  - name: v1
    image: ubuntu
    command: ["/bin/bash", "-c", "while true; do echo Hello-sagar; sleep 5; done"]
---
apiVersion: v1
kind: Pod
metadata:
  name: testpod2
spec:
  containers:
  - name: v2
    image: httpd
    ports:
    - containerPort: 80
```

a. Pod to communication will happen via the Ips.
b. By default pod Ip will not accessible outside the node.

Commands:

- Kubectl apply -f filename.yaml
- Kubectl get all
- kubectl exec -it pod/testpod1 -- /bin/bash
- apt install && apt install curl
- Ping IPaddressOFPod2:80

ClusterIP:

1. This kubernetes service used to internal communication, its provide the IP address
2. Mainly used to communicate between components of microservices
3. Also used to communicate between pods on different-2 nodes & default service type.

Example:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: httpddeployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: httpddeployment
  template:
    metadata:
      labels:
        app: httpddeployment
    spec:
      containers:
      - name: c00
        image: httpd
        ports:
        - containerPort: 80
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ubuntudeployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: ubuntudeployment
  template:
    metadata:
      labels:
```

```yaml
      app: ubuntudeployment
    spec:
     containers:
     - name: c01
       image: ubuntu
       command: ["/bin/bash", "-c", "while true; do echo Hello-sagar; sleep 5; done"]
---
apiVersion: v1
kind: Service
metadata:
  name: demoservice
spec:
  selector:
    app: httpddeployment
  ports:
  - port: 80
    targetPort: 80
  type: ClusterIP      # Specifies the service type i.e ClusterIP or NodePor
```

Commands:

- kubectl apply -f filename.yaml
- To see all resource is running or not: kubectl get all
- kubectl get pod -o wide
- Copy the httpddeployment pod IP i.e. 172.17.0.7
- Now go inside the ubuntu pod
- And run apt update && apt install curl -y
- Run "curl 172.17.0.7:80" and you will get output.
- Run "curl clusterIP:80 i.e. curl 10.104.84.124:80 "
- Now you hit via the pod IP and it's working
- But if someone delete the pod or due to any reason the pod terminated
- Then pod will get new IP and if you hit the command again with old pod IP then it will not give any output
- Now we copy the Cluster Ip and again go inside the ubuntu pod
- kubectl exec -it pod/ubuntudeployment-594f56844c-4w6sk -- /bin/bash
- Now run "curl clusterIP:80 i.e curl 10.104.84.124:80 "
- It will work same
- So now we not need to worry about POD IP.

NodePort:

- Make your service available outside your cluster
- Expose the service on the same port of each selected node in the cluster using NAT.
- This is top level of service of clusterIP
- Here only port we need to know and instead of IP we will use Public Ip of our host.
- If you set the type field to NodePort, the Kubernetes control plane allocates a port from a range specified by --service-node-port-range flag (default: 30000-32767).
- When you only pass .spec.type to NodePort then it will take any random unique port from the default range

Example:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: httpddeployment
spec:
  replicas: 1
  selector: # Tells the controller which pods to watch/belong to
    matchLabels:
      app: httpddeployment
  template:
    metadata:
      labels:
        app: httpddeployment
    spec:
      containers:
      - name: c00
        image: httpd
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: demoservice
spec:
  selector:
```

```
    app: httpddeployment   # Apply this service to any pods which   have the specific label
  ports:
  - port: 80         # Container port exposed
    targetPort: 80     # Pod's port
  type: NodePort    # Specifies the service type i.e ClusterIP or NodePort
```

Commands:

- kubectl apply -f filename.yaml
- Kubectl get all
- minikube service list

Scenario:

You want the `NodePort` service to bind only to a specific IP range, e.g., `192.168.1.0/24`.

```
apiVersion: v1
kind: Service
metadata:
  name: example-nodeport
spec:
  type: NodePort
  ports:
  - port: 80                # Service Port
    targetPort: 8080        # Pod Port
    nodePort: 30001         # Custom NodePort
  selector:
    app: example-app
```


kube-proxy Configuration:

Update the `kube-proxy ConfigMap` to restrict NodePort access to the specific IP range. Edit the `kube-proxy` ConfigMap:

- `kubectl -n kube-system edit configmap kube-proxy`

Add the `nodePortAddresses` field in the configuration:

```
apiVersion:  kubeproxy.config.k8s.io/v1alpha1
kind:  KubeProxyConfiguration
nodePortAddresses:
        -  192.168.1.0/24       # Restrict NodePort to this subnet
mode:  iptables
```

Apply the updated configuration by restarting the `kube-proxy` daemonset:

- `kubectl -n kube-system rollout restart daemonset/kube-proxy`

Validate the NodePort service is listening only on the specified IP range:

➢ netstat -tuln | grep 30001

LoadBlancer:

- service is used to expose your application outside of the Kubernetes cluster
- Automatically provisions an external load balancer (e.g., AWS ELB, Azure LB, GCP LB).
- Distributes incoming traffic evenly across backend pods to improve availability and performance.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: httpddeployment
spec:
 replicas: 1
 selector:
  matchLabels:
   name: httpddeployment
 template:
  metadata:
   name: testpod1
   labels:
    name: httpddeployment
  spec:
   containers:
   - name: c00
    image: httpd
    ports:
    - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
 name: demoservice
spec:
 ports:
 - port: 80       # Exposes port 80 on the service
  targetPort: 80   # Redirects to container's port 80
 selector:
  name: httpddeployment  # The service will route traffic to pods with this label
 type: LoadBalancer
```


Commands:

➢ kubectl apply -f filename.yaml
➢ Kubectl get all
➢ Kubectl get svc
➢ Copy the loadBlancer Ip with port
➢ Run on browser and you can able to get the pod which is running inside the cluster

Headless Service:

Headless Service is a type of service that does not assign an IP address to the service itself. Instead, it enables direct communication with the underlying pods without load balancing, making it useful for certain types of applications, such as stateful or clustered applications where each pod needs to be directly accessible.

Note: example file you will get on next page and copy that file and run.

- Kubectl apply -f filename.yaml
- Kubectl get all
- Now you can see there is there is 1 service running
- Service is ClusterIP type but without IP and here this also called headless service.
- Now go inside the ubuntu pod by using the below command
  kubectl exec -it pod/pod_name -- /bin/bash
- Now install the curl and nslookup to test the headless service benefits by using the below command
  - apt update&& apt install curl -y && apt install dnsutils -y
- Now run the command " nslookup headlessservice "
- And you get the dns and then run the below command and you get the desired result curl headlessservice.default.svc.cluster.local:80

Example:

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: httpddeployment
spec:
 replicas: 1
 selector:
  matchLabels:
   name: httpddeployment
 template:
  metadata:
   name: testpod1
   labels:
    name: httpddeployment
  spec
   containers:
   - name: c00
    image: httpd
    ports:
    - containerPort: 80
---
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: ubuntudeployment
spec:
 replicas: 1
 selector:
  matchLabels:
   name: ubuntudeployment
 template:
  metadata:
   name: testpod2
   labels:
    name: ubuntudeployment
  spec:
   containers:
   - name: c01
    image: ubuntu
    command: ["/bin/bash", "-c", "while true; do echo Hello-sagar; sleep 5 ; done"]
---
apiVersion: v1
kind: Service
metadata:
 name: headlessservice
spec:
 clusterIP: None
 ports:
 - port: 80       # Exposes container port
  targetPort: 80   # Redirects to Pod's port
 selector:
  name: httpddeployment  # Routes traffic to Pods with this label
```

Namespaces: In Kubernetes, *namespaces* provide a logical isolating groups of resources within a single cluster

Type of Namespaces

- Default Namespace
- Custom or User-defined Namespaces
- kube-system : controller , api-server, etcd database, kube-proxy.

- kubectl -n kube-system get pods
- kubectl get namespaces
- kubectl create namespace <project-1>

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

- kubectl run podname --image=nginx *#is used default namespace*
- kubectl get pods
- kubectl -n project-i run podname --image=nginx
- kubectl -n project-1 get all
- kubectl -n project-i delete deployment < deployment-name >

Deployments Strategies:

strategies for managing application updates and rolling out new versions while maintaining availability and minimizing downtime

Deployments Strategies type:

- **Rolling Deployment** : RollingUpdate strategy updates your application by gradually create new replica set and replacing old versions of your pods with new ones. minimal downtime with controlled rolling updates.

```
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxSurge: 2
    maxUnavailable: 0
```

- **maxSurge** : mention the number of new Pods created then old pod determined
- **maxUnavailable :** old pod determined and create new pod

- **Canary Deployment , blue green, red black** : can be implemented with tools like Argo Rollouts for gradual traffic shifting.
- **Recreate Deployment** : Recreate strategy terminates all existing pods before creating new ones

```
strategy:
  type: Recreate
```

Pause Container:

special container automatically created by Kubernetes within each Pod to manage **namespace ,ip,**

- run this command worker-node : docker container ls   # show pause container

Init Containers:

An Init Container is a special container in Kubernetes that runs before the main application container in a Pod. Init containers are primarily used to perform tasks like initializing data, checking dependencies, or configuring the environment.

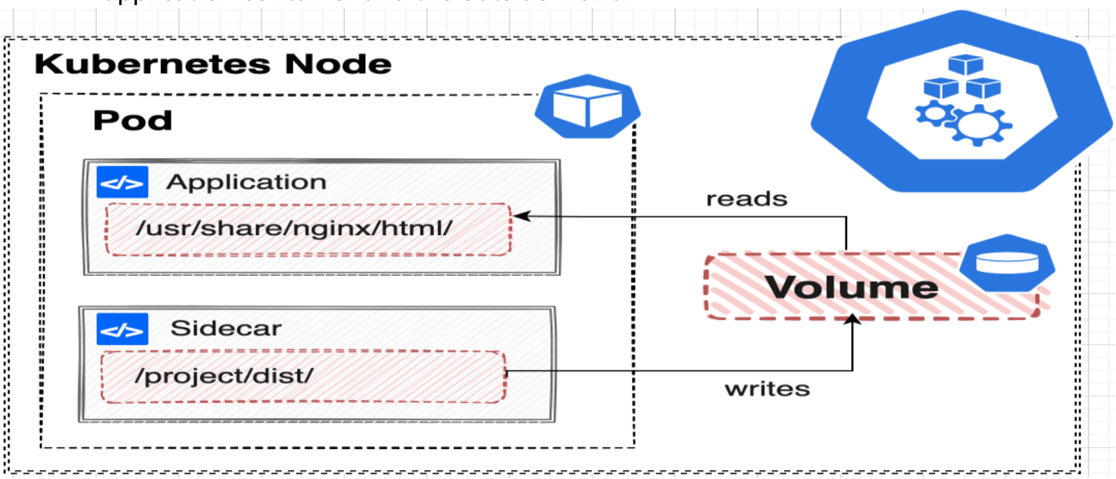
```yaml
apiVersion: v1
kind: Pod
metadata:
  name: init-container-demo
spec:
  initContainers:
  - name: init-container
    image: busybox
    command: ["/bin/sh", "-c", "echo 'Hello from Init Container' > /tmp/message"]
  containers:
  - name: main-container
    image: busybox
    command: ["/bin/sh", "-c", "cat /tmp/message && sleep 3600"]
  restartPolicy: Never
```


- kubectl get pods
- kubectl describe pod init-container-demo
- kubectl exec -it init-container-demo -c main-container -- /bin/bash

Sidecar Container:

A Sidecar Container is a helper container that runs alongside the main application container in the same Pod.

- **Adapter Containers:** An adapter container is used to transform data from one format to another
- **Ambassador Containers:** An ambassador container acts as a proxy or a gateway between the application container and the outside world.




```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-with-sidecar
spec:
  containers:
  - name: nginx
    image: nginx:alpine
    volumeMounts:
    - name: html
      mountPath: /usr/share/nginx/html
    ports:
    - containerPort: 80
  - name: ubuntu
    image: ubuntu:latest
    volumeMounts:
    - name: html
      mountPath: /usr/share/nginx/html
    command: ["/bin/sh", "-c"]
    args:
    - echo "hii my name is vaibhav" > /usr/share/nginx/html/index.html && while true; do sleep 3600;
      done
  volumes:
  - name: html
    emptyDir: {}
```


- kubectl apply -f nginx-with-sidecar.yaml
- kubectl get pods
- kubectl port-forward pod/nginx-with-sidecar 8080:80
- Open a browser and go to http://localhost:8080.

taints *and* tolerations

In Kubernetes, **taints** and **tolerations** are mechanisms to control which pods can be scheduled on particular nodes.

Taints are applied to nodes

A taint consists of three components:

- ✓ **Key**: Identifier for the taint.
- ✓ **Value**: A value paired with the key.

- ✓ **Effect**: The action taken when a pod doesn't tolerate the taint. Effects can be:
- ❖ `NoSchedule`: Pods without matching tolerations are not scheduled on the node.
- ❖ `PreferNoSchedule`: avoid scheduled pods on the node
- ❖ `NoExecute`: Existing pods delete , and new pods are not scheduled.

To add a taint to a node, use the following command:

```
kubectl taint nodes <node-name> <key>=<value>:<effect>
```

To remove a taint from a node:

```
kubectl taint nodes <node-name> <key>:<effect>-
```

Tolerations

Tolerations allow pods to "tolerate" nodes with specific taints.

```
tolerations:
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoSchedule"
```

Kubernetes Probes

Kubernetes uses probes to check if your app (running in a container) is healthy and working properly. There are **three types of probes**:

1. **Liveness Probe**: Checks if the app is still running. Kubernetes restarts the container.
2. **Readiness Probe**: Checks if the app is ready to handle requests. doesn't restart it.
3. **Startup Probe**: Checks if the app has started successfully. Kubernetes restarts the container.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: probe-example
spec:
  containers:
  - name: my-app
    image: my-app-image
    ports:
    - containerPort: 8080

    startupProbe:
      httpGet:
        path: /start
        port: 8080
      initialDelaySeconds: 15
      periodSeconds: 10
      failureThreshold: 30

    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
      initialDelaySeconds: 10
      periodSeconds: 5

    readinessProbe:
      httpGet:
        path: /ready
        port: 8080
      initialDelaySeconds: 5
      periodSeconds: 5
```

Kubernetes Volume Access Modes

1. **ReadWriteOnce:** the volume can be mounted as read-write by a single node.
2. **ReadOnlyMany:** the volume can be mounted as read-only by many nodes.
3. **ReadWriteMany:** the volume can be mounted as read-write by many nodes.

Kubernetes Reclaim Policy

A **Reclaim Policy** defines the behavior of a **PersistentVolume (PV)** after the associated **PersistentVolumeClaim (PVC)**

Retain:
When the PVC is deleted, the PV automatically detach, but this PV **not available for reuse** and the data is not deleted automatically. You can delete manually.

Recycle:
When the PVC is deleted, the PV automatically detach, this PV **available for reuse** and the data is not deleted automatically. You can delete manually.

Delete:
When the PVC is deleted, the PV automatically deleted.

`emptyDir` Volume - Temporary Storage

- emptyDir volume is a temporary storage. you can create pode inside create volume directory, Pod is delete, the emptyDir volume is also deleted.
- If the Pod is deleted or restarted, the data is lost, `emptyDir` volume is **not persistent volume**.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: emptydir-example
spec:
  containers:
  - name: app-container-1
    image: nginx:alpine
    volumeMounts:
    - mountPath: /data          # Data will be stored under /data in the container
      name: shared-storage       # The shared volume name
  volumes:
  - name: shared-storage
    emptyDir: {}              # Define emptyDir volume type
```

- ***kubectl exec -it emptydir-example -- /bin/sh***
- echo "This is a test file" > /data/testfile.txt

Kubernetes `hostPath` Volume

- The volume mounts a **specific path** from the ***host machine*** <node> into the ***container.*** You can delete pod, container your data can not lose
- **Not Portability**: Since the data is stored on the host node, it is not portable across nodes. If the Pod is rescheduled to another node, the data in `hostPath` will not be available unless the same path exists on the new node.
- Any data written by the container to the mounted directory will be stored on the host machine.
- It's useful for scenarios where you want to persist data on the host system or access host-specific files, such as logs or configuration files.

```yaml
volumes:
- name: host-volume
  hostPath:
    path: /mnt/data          # Mount the /mnt/data directory from the host machine
```

Mount EBS Volume

- **Persistent**: EBS volumes are **persistent** even if the Pod is deleted.
- **Automatic Detachment**: When the Pod is deleted, the volume is automatically detached

Creating and Using EBS Volumes:

- **EBS Volume**: Persistent block storage provided by AWS for EC2 instances.
- **EBS Volume ID**: You need the **Volume ID** (e.g., `vol-0c1234567890abcdef`) to reference the volume in Kubernetes.

```yaml
volumes:

- name: ebs-storage

  awsElasticBlockStore:

    volumeID: vol-0c1234567890abcdef  # Use the EBS volume ID you created earlier

    fsType: ext4              # File system type (e.g., ext4)
```

- **gcePersistentDisk**: Mounts a Google Compute Engine persistent disk into a Pod.
- **azureDisk**: Mounts a Microsoft Azure Data Disk into a Pod

NFS (Network File System) Volume in Kubernetes

The **NFS (Network File System)** allows a Pod in Kubernetes to mount a remote file system that is accessible over the network. This can be useful when you need to share data across multiple Pods or even across multiple nodes.

- ***Set up NFS Server:***

Install and configure an NFS server on a machine or an instance (this could be on-prem or in the cloud). Share a directory via NFS (e.g., `/mnt/nfs_share`).

```yaml
volumes:

- name: nfs-volume

  nfs:

    server: <NFS_SERVER_IP>   # IP address of the NFS server

    path: /mnt/nfs_share      # Path to the shared directory on the NFS server

    readOnly: false         # Set to true if you want the volume to be read-only
```

Persistent Volume (PV)

- A **Persistent Volume** is a piece of storage provisioned in the cluster or by an external storage system. Supports multiple backends (NFS, AWS EBS, GCP Persistent Disks, etc.).

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: example-pv
spec:
  capacity:
    storage: 1Gi               # Size of the volume
  accessModes:
    - ReadWriteOnce            # Access mode
  persistentVolumeReclaimPolicy: Retain  # Reclaim policy (Retain, Recycle, Delete)
  storageClassName: manual        # Storage class associated
  hostPath:
    path: /mnt/data            # Host machine path
```

- Check PVs: `kubectl get pv`

Persistent Volume Claim (PVC)

- A **Persistent Volume Claim** is a request for storage by a Pod. A PVC specifies the desired size, access mode, and optionally, the storage class.
- Binds automatically to an appropriate PV.

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: example-pvc
spec:
  accessModes:
    - ReadWriteOnce            # Must match PV's access mode
  resources:
    requests:
      storage: 1Gi             # Requested storage size
  storageClassName: manual        # Must match PV's storageClassName
```

- Check PVCs: `kubectl get pvc`

Storage Class

- A **Storage Class** defines the types of storage and provisioning methods available in a Kubernetes cluster. It enables dynamic provisioning of storage when a PVC is created.
- Supports dynamic storage provisioning.

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast
provisioner: kubernetes.io/aws-ebs    # Backend storage provider
parameters:
  type: gp2                  # EBS volume type
  fsType: ext4               # File system type
```

Dynamic vs Static Provisioning

- **Static Provisioning**: Administrators manually provision PVs and match them with PVCs.

- **Dynamic Provisioning**: PVs are created automatically based on StorageClass when a PVC is created.

Use the PVC in a Pod:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pvc-demo-pod
spec:
  containers:
  - name: app-container
    image: nginx
    volumeMounts:
    - mountPath: "/data"
      name: pvc-volume
  volumes:
  - name: pvc-volume
    persistentVolumeClaim:
      claimName: dynamic-pvc
```

Describe PV or PVC:

- `kubectl describe pv <pv-name>`
- `kubectl describe pvc <pvc-name>`

Changing the Default Pod Limit on a Node

By default, Kubernetes sets a limit of **110 Pods** per node. To increase or decrease this limit, you need to modify the kubelet configuration file (***config.yaml***)

- Vim /var/lib/kubelet/***config.yaml***

maxPods: 200 # Set the desired pod limit

- Check Current Node Pod Limit ***: kubectl describe node <node-name> | grep "Pods"***

Resource Quotas:

- Resource quotas in Kubernetes are used to limit the amount of compute resources (***CPU and memory***) and/or the number of objects resources (such as ***Pods, Services, Persistent Volume Claims, ConfigMaps, Secrets***) that can be created within a namespace.
- When creating a Pod in a namespace with a ***Resource Quota*** applied, it is mandatory to specify resource ***requests and limits*** (for CPU and memory). If these are not defined in the Pod's configuration, the Pod will ***fail to create***.
- To avoid this, you can set ***Default Limit Ranges*** in the namespace. This automatically applies default resource requests and limits to Pods that do not explicitly define them, allowing successful creation.

requests = not metion limit = metion requests = limit	limit = not metion requests = metion limit = 0
create	0 = unlimited not create

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: example-quota
  namespace: default
spec:
  hard:
    pods: "10"                          # Maximum number of Pods in the namespace
    requests.cpu: "2"                   # Total CPU requests allowed
    requests.memory: "1Gi"              # Total memory requests allowed
    limits.cpu: "4"                     # Total CPU limit allowed
    limits.memory: "2Gi"                # Total memory limit allowed
    persistentvolumeclaims: "5"         # Maximum number of PVCs allowed
```


Limit Ranges in Kubernetes

- **Limit Ranges** are a way to enforce default resource requests and limits for Pods and containers in a namespace.

```
apiVersion: v1
kind: LimitRange
metadata:
  name: resource-limits
  namespace: default
spec:
  limits:
  - type: Container
    default:
      cpu: "500m"        # Default CPU limit
      memory: "512Mi"      # Default memory limit
    defaultRequest:
      cpu: "250m"        # Default CPU request
      memory: "256Mi"      # Default memory request
    max:
      cpu: "1"          # Maximum CPU allowed
      memory: "1Gi"       # Maximum memory allowed
    min:
      cpu: "100m"        # Minimum CPU required
```

memory: "128Mi"   # Minimum memory required

- *kubectl get limitrange -n default*
- *kubectl describe limitrange resource-limits -n default*

- If a Pod requests or limits exceed the `max` values or fall below the `min`, the Pod creation fails.

- Example Pod (limit-range-demo.yaml)

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: limit-range-demo
  namespace: default
spec:
  containers:
  - name: nginx-container
    image: nginx
    resources: {}  # No explicit requests/limits
```

- kubectl describe pod limit-range-demo -n default

Secrets in Kubernetes

- Kubernetes **Secrets** are used to store and manage sensitive information such as passwords, OAuth tokens, SSH keys, etc.

Key Points:

- Secrets are stored in etcd in **base64-encoded** format.
- The maximum size of a single secret is **1MB**.
- Kubernetes **does not encrypt** secrets by default, but it is recommended to enable encryption at rest for secrets in etcd.

Types of Secrets in Kubernetes

- **Generic Secrets**: These are the most commonly used secrets, where you can store various types of sensitive information such as keys, passwords, and more.

Create a Secret from Literal Values

To create a secret from a literal value, use the following command:

- *kubectl create secret generic <secret-name> --from-literal=username=dbuser --from-literal=password=secretpass*

This command will create a secret named `<secret-name>` with the specified `username` and `password` values. These values are stored in base64 format in the secret.

Create a Secret from a File

You can also create a secret from the contents of a file, where the file will be encoded in base64 and stored:

- *kubectl create secret generic <secret-name> --from-file=<path-to-file>*

Create a Secret and Export to YAML File

To view the secret, you can export it to a YAML file:

- *kubectl get secrets <secret-name> -o yaml > secret-file.yaml*

Base64 Encoding

When creating a secret in Kubernetes, data is always stored in **base64-encoded** format. For example,

```
echo -n "vaibhav" | base64
# Output: YWRtaW4=
```

Secret YAML Definition Example

```
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
data:
  username: YWRtaW4=     # Base64 encoded value for 'admin'
  password: c2VjcmV0     # Base64 encoded value for 'secret'
```

Accessing Secrets in Pods

- To use the secret in a Pod, you can reference it in your pod's specification as environment variables or mount it as a file.

1. **Use Secrets as Environment Variables**

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


2. **Mount Secrets as Volumes**

```
apiVersion: v1
kind: Pod
metadata:
   name: secret-volume-pod
spec:
   containers:
   - name: nginx
     image: nginx
     volumeMounts:
     - name: secret-volume
       mountPath: "/etc/secrets"
       readOnly: true
   volumes:
```

```yaml
      - name: secret-volume
        secret:
          secretName: my-secret
```

In this case, the secrets `username` and `password` will be available in the `/etc/secrets/` directory of the container.

Important Commands

- To list all secrets in the current namespace: `kubectl get secrets`
- view a secret in base64-encoded format: `kubectl get secret <secret-name> -o yaml`
- To delete a secret: `kubectl delete secret <secret-name>`

ConfigMap in Kubernetes

- ConfigMaps store non-sensitive data like environment variables, command-line arguments, and application configurations.
- Data can be stored as **key-value pairs** or **files**.
- ConfigMaps are not designed for storing sensitive data. Use **Secrets** for sensitive information..

Example YAML definition for a ConfigMap:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-configmap
data:
  version: "16"   # Configuration key-value pair
  name: "admin"
```

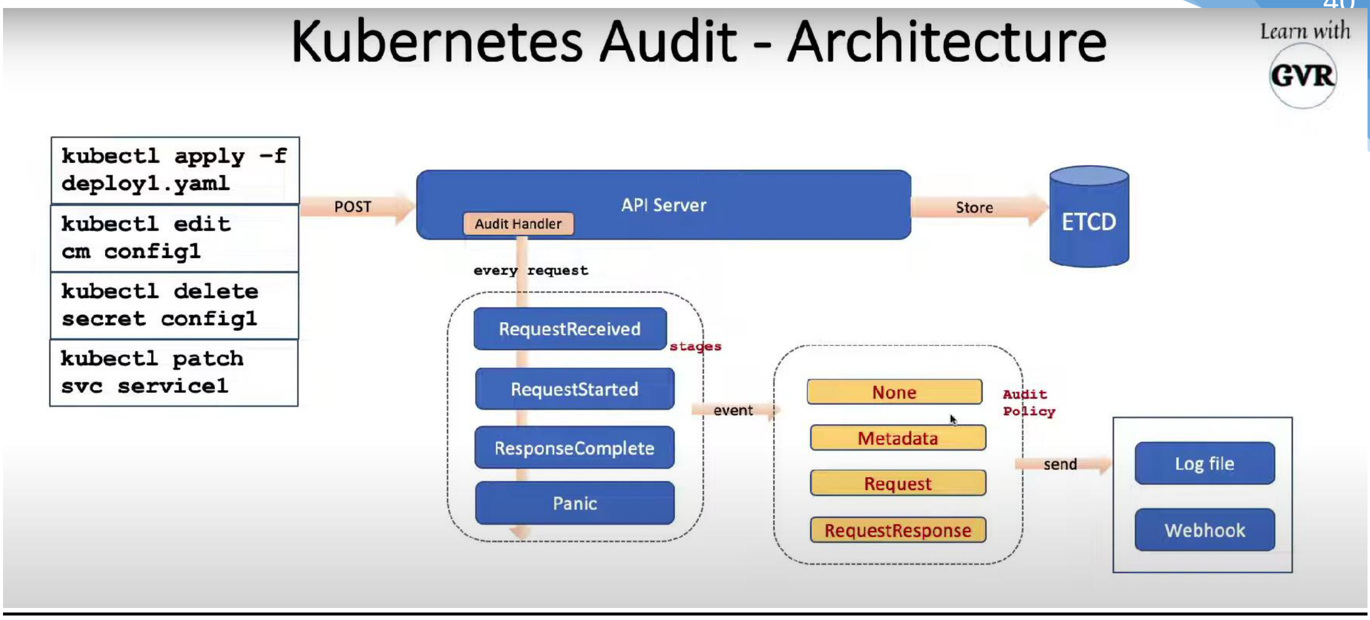
Accessing ConfigMaps in Pods

You can use ConfigMaps in Pods as **environment variables** or **mounted volumes**

Use ConfigMap as Mounted Volumes


```yaml
apiVersion: v1
kind: Pod
metadata:
  name: configmap-volume-pod
spec:
  containers:
  - name: nginx
    image: nginx
    volumeMounts:
    - name: config-volume
      mountPath: "/etc/config"
  volumes:
  - name: config-volume
    configMap:
      name: my-configmap
```

- **List ConfigMaps**: `kubectl get configmaps`
- **View ConfigMap Details**: `kubectl get configmap <configmap-name> -o yaml`
- **Delete ConfigMap**: `kubectl delete configmap <configmap-name>`
- **Edit ConfigMap**: `kubectl edit configmap <configmap-name>`



Kubernetes *Auditing* is an important security measure that can help you monitor and audit various activities in the cluster to ensure the security and compliance of the cluster.

Logs include details such as:

- *Timestamp*: When the request was made.
- *User identity*: Who made the request.
- *Resource*: What resource the request targeted (e.g., pods, services).
- *Action*: What action was performed (e.g., GET, POST, DELETE).

Audit Backends:

Kubernetes supports different output backends for audit logs:

- ***Log backend***: Writes logs to a file.
- ***Webhook backend***: Sends audit events to a remote server.

Audit Policy*:*

- **None**: Do not log events that match this rule.
- **Metadata**: Log basic details like user, time, resource, and action, but not the request or response content.
- **Request**: Log details and the request content, but not the response content.
- **RequestResponse**: Log everything — details, request content, and response content.

Stages:

Define at which point the request should be logged

- **RequestReceived**: Logged when the audit handler gets the request, before passing it further.
- **ResponseStarted**: Logged after response headers are sent but before the body, for long-running requests (e.g., watch).
- **ResponseComplete**: Logged when the response body is fully sent, and no more data will follow.
- **Panic**: Logged if a system panic (critical error) occurs

Check the supported audit policy versions in your Kubernetes cluster.

- `kubectl api-resources | grep audit`

Create an audit policy file ***/etc/kubernetes/audit-policy.yaml*** to define events and rules that need to be audited.

```
apiVersion: audit.k8s.io/v1
```

```yaml
kind: Policy
rules:
  # Log changes to Namespaces and Pods at the RequestResponse level.
  - level: RequestResponse
    resources:
      - group: "*"
        resources: ["pods", "namespaces"]
  # Log pod changes in specific namespaces (e.g., "dey" and "default") at the Request level.
  - level: Request
    resources:
      - group: "*"
        resources: ["pods"]
    namespaces: ["dey", "default"]
  # Log changes to ConfigMaps, Secrets, Services, Deployments, and ServiceAccounts at the
     Metadata level.
  - level: Metadata
    resources:
      - group: ""
        resources: ["secrets", "configmaps", "services", "deployments", "serviceaccounts"]
  # Catch-all rule - Log all requests at the Metadata level.
  - level: Metadata
```

Verify Kubernetes Policy File: kubectl apply -f /etc/kubernetes/audit-policy.yaml --dry-run=client

Enable Auditing of API Server

Edit the configuration file of the Kubernetes API Server (*usually* ***/etc/kubernetes/manifests/kube-apiserver.yaml***) and add audit configuration.

```yaml
- --audit-log-path=/var/log/k8-audit.log
- --audit-policy-file=/etc/kubernetes/audit-policy.yaml
- --audit-log-maxage=10
- --audit-log-maxbackup=5
- --audit-log-maxsize=100

  volumeMounts:
    - name: audit
      mountPath: /etc/kubernetes/audit
      readOnly: false
  volumes:
    hostPath:
    - name: audit
      path: /etc/kubernetes/audit
      type: DirectoryOrCreate
```

Parameter Description:

- `--audit-log-path`: The storage path of the audit log.
- `--audit` -log-format=json : Specify the format of the audit log.
- `--audit-log-maxage`: The maximum number of days to retain audit log files.
- `--audit-log-maxbackup`: Maximum number of backups of audit log files.
- `--audit-log-maxsize`: Maximum size of audit log files.
- `--audit-policy-file`: Path to the audit policy file.

➢ restart the API Server: ***sudo systemctl restart kubelet***

➢ The audit log will be recorded in the specified path. ***cat /var/log/kubernetes/audit.log***

Using audit logs, you can monitor sensitive operations that occur in the cluster, such as Pod creation and deletion.

➢ cat /var/log/kubernetes/audit.log | grep "CreatePod"

Role-Based Access Control (RBAC) in Kubernetes

RBAC is a secure and flexible way to manage authorization in Kubernetes. It allows administrators to define granular permissions for users, groups, or service accounts based on their roles within the cluster.

- **Verbs:** Actions allowed on resources (e.g., get, list, create, update, delete).
- **Resources:** Kubernetes API objects like pods, services, secrets, configmaps.
- **Subjects:** The entities that are assigned roles: users, groups, or service accounts.

Role:

A set of permissions (verbs and resources) defined within a particular namespace.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default                   # Role is scoped to this namespace
  name: pod-reader
rules:
- apiGroups: [""]                      # "" refers to the core API group
  resources: ["pods"]                  # Specifies Pods as the target resource
  verbs: ["get", "watch", "list"]      # Allowed actions
```

- To apply this role: *kubectl apply -f role.yaml*
- To check the created role*: kubectl get role*

RoleBinding:

Links a Role to a user, group, or service account within a specific namespace.

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

- To apply this role binding*: kubectl apply -f rolebinding.yaml*
- To check the created role binding*: kubectl get rolebinding*
- To check the permissions of the Vaibhav user: *kubectl auth can-i get pod --as Vaibhav*

Cluster Role

Similar to a Role but applies cluster-wide resource used to define permissions that are not limited to a single namespace.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: secret-reader
rules:
- apiGroups: [""]
```

```yaml
    resources: ["secrets"]
    verbs: ["get", "watch", "list"]
```

- To apply this cluster role: *kubectl apply -f clusterrole.yaml*
- To check the created cluster role: *kubectl get clusterrole*

Role Binding (Namespace-level)

The rolebinding.yaml file defines a role binding named read-secrets that binds the secret-reader cluster role to the user dev in the development namespace.

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-secrets
  namespace: development
subjects:
- kind: User
  name: dev                                   # Name of the user
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: secret-reader                          # Reference to the ClusterRole
  apiGroup: rbac.authorization.k8s.io
```

- To apply this role binding*: kubectl apply -f rolebinding.yaml*
- To check the created role binding: *kubectl get rolebinding*
- To check the permissions of the dev user in the development namespace:
- *kubectl auth can-i get secret --as dev -n development*

Cluster Role Binding

Links a ClusterRole to a user, group, or service account at the cluster level.

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: read-secrets-global
subjects:
- kind: User
  name: vaibhav
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: secret-reader
  apiGroup: rbac.authorization.k8s.io
```

- To apply this cluster role binding*: kubectl apply -f clusterrolebinding.yaml*
- To check the created cluster role binding*: kubectl get clusterrolebinding*
- To check the permissions of the vaibhav user across all namespaces:
- *kubectl auth can-i get secret --as vaibhav -A*

Service Account in Kubernetes

Service Accounts in Kubernetes allow you to authenticate and authorize applications and services running within a cluster. They provide a way to grant specific permissions and access control to pods and containers.. By default, every Pod in Kubernetes uses the `default` ServiceAccount

In this practical, we will cover the following steps:

- Creating a Service Account
- Creating a token for the Service Account
- Creating a Role to define permissions
- Creating a RoleBinding to associate the Role with the Service Account
- Using the Service Account in a Pod
- Verifying access permissions

To create a Service Account, use the following commands: `kubectl create sa my-service-account`

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: my-service-account
  namespace: default
```

To create a token "`my-service-account`" : `kubectl create token my-service-account`

Defining Permissions with Roles

To define permissions for the Service Account, we need to create a Role. Use the following YAML file:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups:
  - ''
  resources:
  - pods
  verbs:
  - get
  - watch
  - list
```

To associate the Role with the Service Account, create a ***RoleBinding.*** Use the following YAML file:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: ServiceAccount
  name: my-service-account
  namespace: default
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

To use the Service Account in a Pod

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  serviceAccountName: my-service-account
  containers:
```

```yaml
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

To verify the access permissions of the Service Account, use the following command:

```
kubectl auth can-i get pods --as=system:serviceaccount:default:my-service-account
```

Security Context in Kubernetes

Security Context in Kubernetes it is provide the additional layer of security for a Pod or container.

Types of Security Contexts

- **Pod-Level Security Context**: Applies settings to all containers within a Pod.
- **Container-Level Security Context**: Applies specific settings to an individual container.

Using root user perform any tasks

- Running process
- File access
- Network interface
- System configure

This YAML file demonstrates the implementation of security context at the ***pod level.***

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo
spec:
  securityContext:
    runAsUser: 1000
    runAsGroup: 3000
    fsGroup: 2000
  volumes:
  - name: demo-vol
    emptyDir: {}
  containers:
  - name: sc-demo
    image: busybox:1.28
    command: [ "sh", "-c", "sleep 1h" ]
    volumeMounts:
    - name: demo-vol
      mountPath: /data/demo
```

To apply this file, use the following command:

```
kubectl apply -f sc-demo-1.yaml
```

To access the shell inside the pod, run the following command:

```
kubectl exec -it security-context-demo -- /bin/sh
```

To check the user ID, group ID, and filesystem ID, run the following commands:

```
id
ps aux
```

The id command displays the user ID and group ID of the current user running inside the pod. The ps aux command shows the processes running inside the pod along with their details.

Since a volume is mounted, you can navigate to the mounted directory and perform file operations:

```
cd /data/demo
echo hello devops >> myfile
```

- ls -l

These commands change the directory to "/data/demo", appends the text "hello devops" to a file named "myfile", and lists the files in the directory.

File 2: sc-demofile-2.yaml

This YAML file demonstrates the implementation of security context at both the pod and container level.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo-2
spec:
  securityContext:
    runAsUser: 1000
  containers:
  - name: sc-demo-2
    image: gcr.io/google-samples/node-hello:1.0
    securityContext:
      runAsUser: 2000
```

To apply this file, use the following command:

- kubectl apply -f sc-demofile-2.yaml

To access the shell inside the pod, run the following command:

- kubectl exec -it security-context-demo-2 -- /bin/sh

This command executes an interactive shell inside the pod named "security-context-demo-2".
To check the applied security context on the container, run the following commands:

- ps aux
- id

The ps aux command displays the running processes inside the container. The id command shows the user ID and group ID of the current user running inside the container.

File 3: sc-demo-3.yaml

This YAML file demonstrates the addition of the NET_ADMIN capability to a container.


```yaml
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo-3
spec:
  containers:
  - name: sc-demo-3
    image: ubuntu
    command: [ "sh", "-c", "sleep 1h" ]
    securityContext:
      capabilities:
        add: ["NET_ADMIN"]
        drop: ["ALL"]                # Drops all other capabilities
```

To apply this file, use the following command:

- kubectl apply -f sc-demo-3.yaml

To access the shell inside the pod, run the following command:

- kubectl exec -it security-context-demo-3 -- /bin/bash

This command executes an interactive bash shell inside the pod named "security-context-demo-3".
To install the required package, run the following commands inside the pod:

- apt update
- apt install iproute2

These commands update the package lists and install the "iproute2" package, which provides advanced IP routing and network devices configuration.
To check the network interfaces, run the following command:

- ip link show

This command displays the network interfaces and their details, such as name, state, and MAC address.
To add an IP address to the eth0 network interface, use the following command:

➢ `ip addr add 192.168.0.10/24 dev eth0`

This command assigns the IP address "192.168.0.10" with a subnet mask of "/24" to the "eth0" network interface.

To check the added IP address, run the following command:

➢ `ip addr show eth0`

This command displays the details of the "eth0" network interface, including the assigned IP address.

Key Fields in Security Context

Field	**Description**
`runAsUser`	Specifies the user ID for running the container process.
`runAsGroup`	Specifies the group ID for the container process.
`fsGroup`	Defines the file system group ID for shared storage.
`allowPrivilegeEscalation`	Prevents processes inside the container from gaining additional privileges.
`privileged`	Allows or disallows running the container in privileged mode.
`readOnlyRootFilesystem`	Ensures the root file system is read-only.
`capabilities`	Adds or removes specific Linux capabilities from the container.
`seLinuxOptions`	Configures SELinux labels for the Pod or container.

Static Pods

Static Pods are special types of Pods that are managed directly by the **kubelet** on a specific node

Create a YAML file named static-web.yaml with the following contents:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: static-web
spec:
  containers:
  - name: nginx
    image: nginx
```

Locate the folder on the Kubernetes node that contains the static pod's YAML files, typically ***/etc/kubernetes/manifests/.***

Copy the static-web.yaml file into the folder using the command:

➢ ***sudo cp static-web.yaml /etc/kubernetes/manifests/***

➢ ***kubectl get pods -n kube-system***

Kubernetes Network Policy

A **Network Policy** in Kubernetes is a way to control traffic flow between pods, namespaces, or external services.

Default Behavior

- Without a Network Policy:
  All pods in a cluster can freely communicate with each other.
- With a Network Policy:
  You define rules to **allow** or **deny** traffic, making the cluster more secure.

Network Policy Controllers

Kubernetes doesn't enforce Network Policies on its own. You need a **network policy agent** (CNI plugin) installed in the cluster to apply these policies. Popular options include:
Calico, **Antrea**, **Cilium**, **Weave Net**, **Kube-Router**, **Flannel** (lacks full Network Policy support by default).

Components of a Network Policy

1. Ingress: Controls incoming traffic to a pod.
2. Egress: Controls outgoing traffic from a pod.
3. Namespace and Pod Selection: You can specify:
4. Ephemeral Block (IPBlock): Restricts access based on IP ranges

Network Policy

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-frontend-to-backend
  namespace: default
spec:
  podSelector:
    matchLabels:
      app: backend
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: frontend
      namespaceSelector:
        matchLabels:
          environment: frontend-ns
    ports:
    - protocol: TCP
      port: 8080
    - protocol: TCP
      port: 443
```


➢ kubectl apply -f network-policy.yaml

Restrict Outgoing Traffic

Allow pods with `app=web` to access only a specific IP range (e.g., 192.168.1.0/24).

```
policyTypes:
- Egress
egress:
- to:
  - ipBlock:
      cidr: 192.168.1.0/24
```

Jobs and CronJobs in Kubernetes:

Kubernetes provides **Jobs** and **CronJobs** to handle batch processing and scheduled tasks

1. Jobs

A Kubernetes **Job** ensures that a specified number of pods complete their tasks successfully. It is typically used for short-lived, one-time workloads.

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: example-job
spec:
  completions: 3  # Total successful completions needed
  parallelism: 2  # Pods to run in parallel
  template:
    spec:
      containers:
      - name: example-task
        image: busybox
        command: ["sh", "-c", "echo 'Hello, Kubernetes!' && sleep 10"]
      restartPolicy: OnFailure  # Restart only on failure
```


Explanation

- **`completions:`** The number of successful completions required.
- **`parallelism:`** Controls how many pods can run concurrently.
- **`restartPolicy`**: Ensures that failed pods are restarted.

2. CronJobs

A Kubernetes **CronJob** schedules Jobs to run periodically based on a specified **cron schedule**. It is ideal for recurring tasks like backups or report generation.

Explanation

- **`schedule:`** Specifies the cron schedule (`*/5 * * * *` runs every 5 minutes).
- **`jobTemplate`**: Defines the pod template for the task.
- **`successfulJobsHistoryLimit`** **and** **`failedJobsHistoryLimit`**: Limit how many past job records are kept.

```yaml
apiVersion: batch/v1
kind: CronJob
metadata:
  name: example-cronjob
spec:
```

```
schedule: "*/5 * * * *"  # Every 5 minutes
jobTemplate:
 spec:
  template:
   spec:
    containers:
    - name: periodic-task
     image: busybox
     command: ["sh", "-c", "echo 'This task runs every 5 minutes!'"]
    restartPolicy: OnFailure
successfulJobsHistoryLimit: 3  # Retain the last 3 successful runs
failedJobsHistoryLimit: 2      # Retain the last 2 failed runs
```

- kubectl get pod
- kubectl logs pods/cpu-mem-monitor-cronjob-28892885-knqs9

Kubernetes DNS:

Kubernetes DNS enables seamless service-to-service communication within the cluster by resolving names to their corresponding IPs. Key components include **CoreDNS** (default since Kubernetes v1.12) and **Kube-DNS** (legacy). Services and Pods can be accessed using **Fully Qualified Domain Names (FQDNs)** or their simple names.

DNS Record Types in Kubernetes

1. **Service DNS Records**:
   - **A Records**:
     - Normal Service: `svc.namespace.svc.cluster.local` → Resolves to ClusterIP.
     - Headless Service: Resolves to Pod IPs, no load balancing.
   - **CNAME Records**: Points to other hostnames (useful for cross-cluster service discovery).
   - **SRV Records**:
     - For named ports, e.g., `_port._protocol.svc.namespace.svc.cluster.local`.
     - Includes priority, weight, port, and target.
2. **Pod DNS Records**:
   - **A Records**: `pod-ip.namespace.pod.cluster.local` → Resolves to Pod IP.
   - **Custom Hostnames/Subdomains**:
     - `hostname.subdomain.namespace.svc.cluster.local` if configured.

Combined Pod and Service YAML

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  labels:
    app: my-app
spec:
  containers:
    - name: my-container
      image: nginx:latest  # Replace with your desired image
      ports:
        - containerPort: 80  # The port your application listens on
---
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: my-app  # Matches the label of the pod
  type: NodePort  # Expose the service externally
```

```yaml
    ports:
     - protocol: TCP
       port: 80        # Service port
       targetPort: 80    # Pod container port
       nodePort: 30007   # External port on the node (optional)
```

- kubectl apply -f pod-service.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod-1
spec:
  containers:
   - name: my-container
     image: nginx:latest  # Replace with your desired container image
     ports:
      - containerPort: 80  # Port your application listens on inside the container
```

- kubectl apply -f pod.yaml
- kubectl get pods
- **kubectl exec -it my-pod-1 -- /bin/bash**
- **curl my-service**

To create a Pod directly from diff namespace the command line:

- kubectl run my-pod --image=nginx --restart=Never -n my-namespace
- kubectl get pods -n my-namespace

Access the Service via its DNS name:

- curl http://my-service.default.svc.cluster.local

Debugging DNS

Check DNS Resolution:

- Use nslookup:
- kubectl exec -it <pod-name> -- nslookup <service-name>
- Verify /etc/resolv.conf:
- kubectl exec <pod-name> cat /etc/resolv.conf

Check DNS Components:

- **DNS Pods**:
- kubectl get pods -n kube-system
- **DNS Service**:
- kubectl get svc kube-dns -n kube-system
- **DNS Endpoints**:

➢ `kubectl get ep kube-dns -n kube-system`

"🔐Secure Kubernetes Secrets with Confidence: A Complete Guide to Sealed Secrets"

Sealed Secrets in Kubernetes

Sealed Secrets is a Kubernetes-native way to securely manage secrets using encryption, even before they are applied to the cluster. It ensures that sensitive information is encrypted and safely stored in version control systems like Git.

How Sealed Secrets Work

1. **Encryption**: A `kubeseal` CLI tool encrypts sensitive data using a public key from a `Sealed Secrets Controller`.
2. **Storage**: The encrypted secret (sealed secret) is stored as a Kubernetes custom resource (CRD) in the cluster or version control.
3. **Decryption**: The controller running in the cluster decrypts the sealed secret using its private key and creates a standard Kubernetes Secret.

Components of Sealed Secrets

1. **kubeseal CLI**:
   - A client-side tool used to encrypt secrets.
   - Encrypts data with the public key of the Sealed Secrets controller.
2. **Sealed Secrets Controller**:
   - A Kubernetes operator running in the cluster.
   - Manages the private key for decryption and converts sealed secrets into Kubernetes Secrets.
3. **Custom Resource Definition (CRD)**:
   - The encrypted secret is stored as a `SealedSecret` resource in the cluster.

Installation

1. **Install the Controller**::

➢ `helm repo add sealed-secrets https://bitnami-labs.github.io/sealed-secrets`
➢ `helm install sealed-secrets sealed-secrets/sealed-secrets`

Using YAML:

➢ `kubectl apply -f` https://github.com/bitnami-labs/sealed-secrets/releases/download/v0.27.3/controller.yaml

2. **Install `kubeseal` CLI**:

➢ Download from the [Sealed Secrets GitHub Releases](https://github.com/bitnami-labs/sealed-secrets/releases): https://github.com/bitnami-labs/sealed-secrets/releases
➢ curl -OL "https://github.com/bitnami-labs/sealed-secrets/releases/download/v0.27.3/kubeseal-0.27.3-linux-amd64.tar.gz"
➢ tar -xvzf kubeseal-0.27.3-linux-amd64.tar.gz kubeseal
➢ sudo install -m 755 kubeseal /usr/local/bin/kubeseal

Usage

1. Create a Secret

Create a standard Kubernetes Secret manifest: `my-secret.yaml`

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
  namespace: default
type: Opaque
data:
  username: dXNlcm5hbWU=  # base64 encoded value
  password: cGFzc3dvcmQ=  # base64 encoded value
```

2. Encrypt the Secret

Use the `kubeseal` CLI to encrypt the secret:

- `kubeseal --format=yaml <my-secret.yaml >my-sealed-secret.yaml`

- The output is a *cat* ***SealedSecret*** resource:

```yaml
apiVersion: bitnami.com/v1alpha1

kind: SealedSecret

metadata:

 name: my-secret

 namespace: default

spec:

 encryptedData:

  username: <encrypted-data>

  password: <encrypted-data>
```

```yaml
---
apiVersion: bitnami.com/v1alpha1
kind: SealedSecret
metadata:
  creationTimestamp: null
  name: my-secret
  namespace: default
spec:
  encryptedData:
    password: AgCB0iCWL0tbMFHvUt4DnsWUIkioiiROd9wUFD7cOywo9h1sv2/c2ENnjzYErwsRozJaH8PZhE9B88yWn0WKgO+YLMMFB6yNpu8mSbDpx1FHwOAbXFhc2fQargExCRqlh
    username: AgAVF4SbpHH4rrwdBE0m5j8b59Y1Hv3tr1u8L/5+13rzg7dtMfWvY/+myGsZYZPiMuZACQUlzG+RNiGVAqflySaCEziCfRZyiMzddMFOSueiYVHdsxAC3+5yUBjDyazwW
  template:
    metadata:
      creationTimestamp: null
      name: my-secret
      namespace: default
    type: Opaque
```

Apply the SealedSecret: Use `kubectl` to apply the SealedSecret to your Kubernetes cluster.

- ➢ `kubectl apply -f sealedsecret.yaml`

Verify the Secret Creation: After applying, the Sealed Secrets controller will create a decrypted Secret in the specified namespace (`default`).

- ➢ `kubectl get secret my-secret -n default`

You can inspect the Secret (decoded) if needed:

- ➢ `kubectl get secret my-secret -n default -o yaml`

```
controlplane $ kubectl get secret my-secret -n default -o yaml
apiVersion: v1
data:
  password: cGFzc3dvcmQ=
  username: dXNlcm5hbWU=
kind: Secret
metadata:
  creationTimestamp: "2024-12-09T12:53:26Z"
  name: my-secret
  namespace: default
  ownerReferences:
  - apiVersion: bitnami.com/v1alpha1
    controller: true
    kind: SealedSecret
    name: my-secret
    uid: ae6209bc-384c-4c1a-89eb-f8bf28a2cef1
  resourceVersion: "3959"
  uid: 3c8fca93-7ab0-4d90-8086-7299150e8334
type: Opaque
```

Decode them locally:

- ➢ `echo "dXNlcm5hbWU=" | base64 --decode`

4. Decrypt and Create a Secret

The Sealed Secrets controller automatically decrypts the sealed secret and creates the corresponding Kubernetes Secret in the specified namespace.

Benefits of Sealed Secrets

1. **GitOps-Friendly**: Securely store and manage secrets in Git repositories.
2. **Asymmetric Encryption**: Ensures only the cluster can decrypt the secret using the private key.
3. **Namespace/Cluster Binding**: Secrets can be restricted to specific namespaces or clusters.
4. **Automation**: Works seamlessly with CI/CD pipelines.

"Protecting Your Secrets: How to Encrypt Kubernetes Secrets in etcd"

Kubernetes: Encrypting Secrets in etcd

In Kubernetes, secrets are stored in the `etcd` key-value store, which serves as the cluster's primary data storage. By default, secrets are only Base64-encoded, not encrypted, which makes them potentially vulnerable if `etcd` is compromised. Encrypting secrets at rest provides an additional layer of security to protect sensitive data.

Steps to Enable Secrets Encryption in etcd

Step 1: Generate an Encryption Key

To generate a secure Base64-encoded encryption key, use:

- head -c 32 /dev/urandom | base64

Example generated key:

o556O5o4A2mSFccVEJcQdRiJ+YiYT23H8uGZYqPt+JM=

Step 2: Create an Encryption Configuration File

Define the encryption provider and specify the encryption key in a new file called `encryption-config.yaml`. Example:

```
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  - resources:
      - secrets
    providers:
      - aescbc:  # Encryption provider (AES-CBC)
          keys:
            - name: key1
              secret: r7uZiIe8cYNTubuljC1GPaSbIemrCfp680LM1ZDJOdY=
      - identity: {}  # Fallback for non-encrypted data
```

Step 3: Copy the Configuration File to the Correct Location

Copy the `encryption-config.yaml` file to the `/etc/kubernetes/pki/` directory:

- sudo cp encryption-config.yaml /etc/kubernetes/pki/

Step 4: Verify the File Location

Check if the file has been copied successfully:

- ls -l /etc/kubernetes/pki/encryption-config.yaml

Step 5: Update the API Server Configuration

Modify the `kube-apiserver` manifest to include the encryption configuration. The manifest file is typically located at `/etc/kubernetes/manifests/kube-apiserver.yaml`.

Add the following argument:

```
- --encryption-provider-config=/etc/kubernetes/pki/encryption-config.yaml
```

Step 6: Restart the API Server

After saving the changes, the `kube-apiserver` pod will automatically restart. You can verify this by checking the pods in the `kube-system` namespace:

- `kubectl get pods -n kube-system`

Step 7: Re-encrypt Existing Secrets

Newly created secrets will be encrypted automatically. However, existing secrets will remain unencrypted. To re-encrypt them, you can:

- **Backup and Restore** secrets using a script or tool:
  - `kubectl get secrets --all-namespaces -o yaml | kubectl replace -f -`

Step 8: Verify Encryption

After enabling encryption at rest, verify that secrets are indeed encrypted in `etcd`:

```
ETCDCTL_API=3 etcdctl get /registry/secrets/default/my-secret \
  --cert=/etc/kubernetes/pki/etcd/server.crt \
  --key=/etc/kubernetes/pki/etcd/server.key \
  --cacert=/etc/kubernetes/pki/etcd/ca.crt
```

Custom Resource Definition (CRD)

Overview

- CRD is a Kubernetes feature that allows you to extend Kubernetes capabilities by defining your own custom resources.
- Custom resources are extensions of Kubernetes API that allow you to manage and store custom application configurations or domain-specific objects.

Key Concepts

1. Custom Resource (CR):

- A resource created using CRD.
- Example: `kind: MyCustomResource` can define resources like `Database` or `Cache`.

2. Custom Resource Definition (CRD):

- A Kubernetes object that defines the schema and behavior of your custom resources.
- Acts as a template for creating custom resources.

Why Use CRDs?

- Enable the management of application-specific configurations alongside Kubernetes-native resources.
- Automate operational tasks using Kubernetes declarative model.
- Integrate third-party systems into Kubernetes workflows.

Components of CRD

1. apiVersion:

- Specifies the API version (e.g., `apiextensions.k8s.io/v1`).

2. kind:

- Always set to `CustomResourceDefinition`.

3. metadata:

- Contains metadata like the name of the CRD.

4. spec:

- **group**: Logical grouping for the custom resource (e.g., `example.com`).
- **names**: Plural, singular, and kind names of the resource.
- **scope**: Namespace or cluster-scoped resource.
- **versions**: Versioning for the custom resource (e.g., `v1`, `v2`).
- **schema**: Defines the structure and validation rules for the custom resource.

Basic Example: CRD YAML

CRD Definition

```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: databases.example.com
spec:
  group: example.com
  names:
    plural: databases
    singular: database
    kind: Database
    shortNames:
      - db
  scope: Namespaced
  versions:
    - name: v1
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                username:
                  type: string
                password:
                  type: string
                databaseName:
                  type: string
```

Custom Resource (CR) Example

```yaml
apiVersion: example.com/v1
kind: Database
metadata:
  name: my-database
spec:
  username: admin
  password: mypassword
  databaseName: mydb
```

Creating a CRD

1. Apply the CRD
- kubectl apply -f crd.yaml

2. Verify the CRD
- kubectl get crds

3. Create Custom Resources
- kubectl apply -f custom-resource.yaml

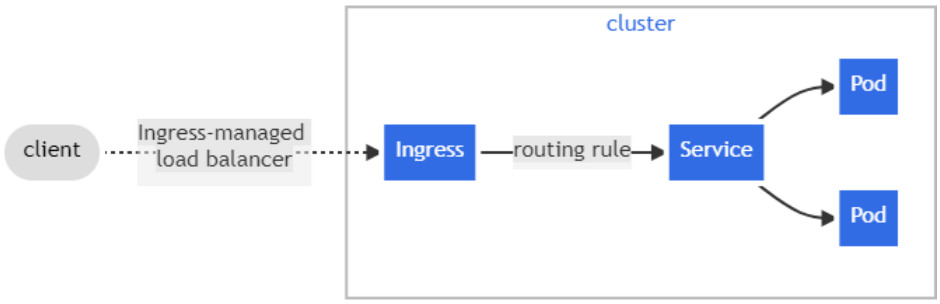
4. Verify Custom Resources
- kubectl get databases

Useful Commands

Command	Description
kubectl explain <CRD-name>	Displays schema and fields of the CRD.
kubectl get <CR-kind>	Lists custom resources of the specified kind.
kubectl describe <CRD-name>	Provides detailed information about the CRD.
kubectl delete -f <crd.yaml>	Deletes the CRD and associated custom resources.

Introduction:

1. An API object that manages external access to the services in a cluster
2. typically it works on http request
3. Ingress may provide load balancing, SSL termination and name-based virtual hosting
4. Ingress exposes HTTP and HTTPS routes from outside the cluster to services within the cluster.
5. Traffic routing is controlled by rules defined on the Ingress resource.



6. An Ingress may be configured to give Services externally-reachable URLs, load balance traffic, terminate SSL / TLS, and offer name-based virtual hosting
7. usually ingress provide a load balancer, though it may also configure your edge router or additional frontends to help handle the traffic
8. An Ingress does not expose arbitrary ports or protocols.
9. Exposing services other than HTTP and HTTPS to the internet typically uses a service of type Service.Type=NodePort or Service.Type=LoadBalancer

10. Ingress mostly apply on service so we can say that it's high level object of service.

KUBERNETES

vaibhav upare