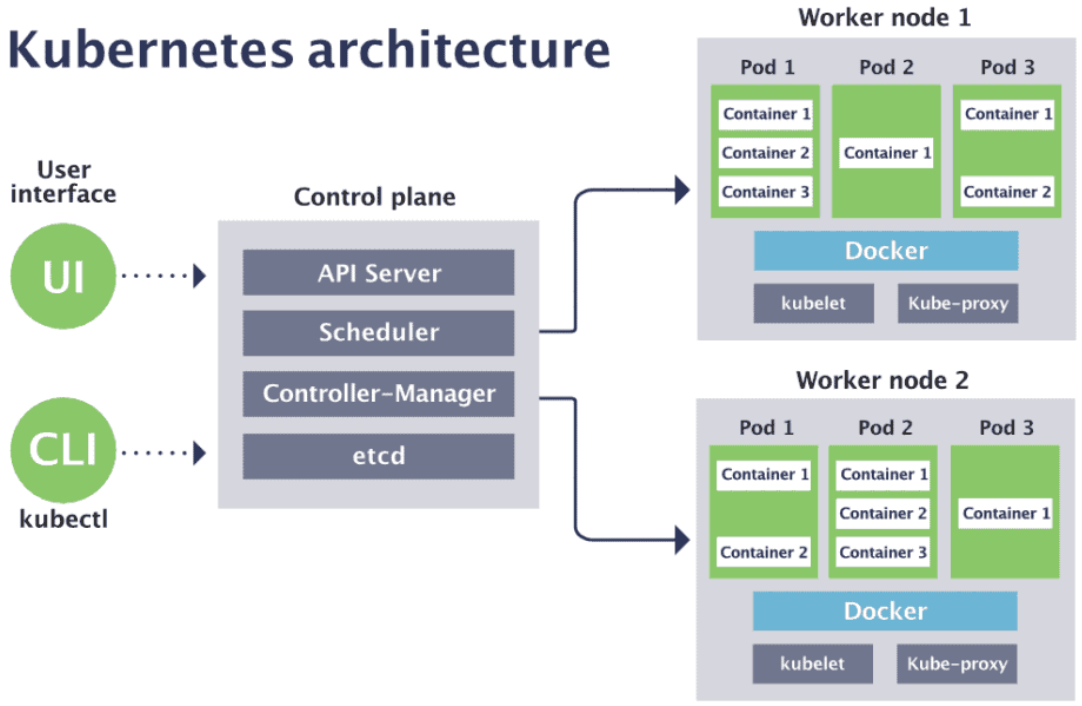
Kubernetes is an open-source container orchestration platform that automates the deployment, management, scaling, and networking of containers.

Container orchestration is the process of automating the deployment, management, scaling, and networking tasks of containers in a distributed environment.

Kubernetes was developed by Google using the Go Programming Language, and this amazing technology has been open-source since 2014.

Prerequisites
Familiarity with the Linux Terminal
Familiarity with Docker

Kubernetes Architecture



In the world of Kubernetes, a node can be either a physical or a virtual machine with a given role. A collection of such machines or servers using a shared network to communicate between each other is called a cluster.

Each server in a Kubernetes cluster gets a role. There are two possible roles:

control-plane — Makes most of the necessary decisions and acts as sort of the brains of the entire cluster. This can be a single server or a group of server in larger projects.

node — Responsible for running workloads. These servers are usually micro managed by the control plane and carries out various tasks following supplied instructions.
kubectl run mynginx --image=nginx
kubectl get pods.

Control Plane Components

The control plane in a Kubernetes cluster consists of **five** components. These are as follows:

1. **kube-api-server:** This acts as the entrance to the Kubernetes control plane, responsible for validating and processing requests delivered using client libraries like the `kubectl` program.

2. **etcd:** This is a distributed key-value store which acts as the single source of truth about your cluster. It holds configuration data and information about the state of the cluster.

3. **kube-controller-manager:** The controllers in Kubernetes are responsible for controlling the state of the cluster. When you let Kubernetes know what you want in your cluster, the controllers make sure that your request is fulfilled.
4. **kube-scheduler:** Assigning task to a certain node considering its available resources and the requirements of the task is known as scheduling.
5. **cloud-controller-manager:** In a real world cloud environment, this component lets you wire-up your cluster with your cloud provider's (GKE/EKS) API. This way, the components that interact with that cloud platform stays isolated from components that just interact with your cluster. In a local cluster like `minikube`, this component doesn't exist.

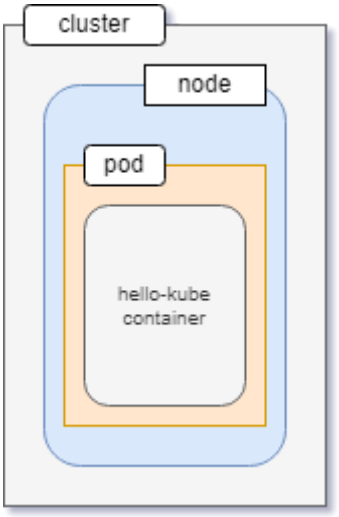
Node Components

Compared to the control plane, nodes have a very small number of components. These components are as follows:

1. **kubelet:** This service acts as the gateway between the control plane and each of the nodes in a cluster. Every instruction from the control plane towards the nodes, passes through this service. It also interacts with the `etcd` store to keep the state information updated.
2. **kube-proxy:** This small service runs on each node server and maintains network rules on them. Any network request that reaches a service inside your cluster, passes through this service.
3. **Container Runtime:** Kubernetes is a container orchestration tool hence it runs applications in containers. This means that every node needs to have a container runtime like Docker or rkt or cri-o.

Pods: Pods are the smallest deployable units of computing that you can create and manage in Kubernetes.

A pod usually encapsulates one or more containers that are closely related sharing a life cycle and consumable resources.



Usually, you should not manage a pod directly. Instead, you should work with higher level objects that can provide you much better manageability. You'll learn about these higher level objects in later sections.

Service: A service in Kubernetes is an abstract way to expose an application running on a set of pods as a network service.

Install and Set Up kubectl on Windows

https://kubernetes.io/docs/tasks/tools/install-kubectl-windows/
https://dl.k8s.io/release/v1.25.0/bin/windows/amd64/kubectl.exe

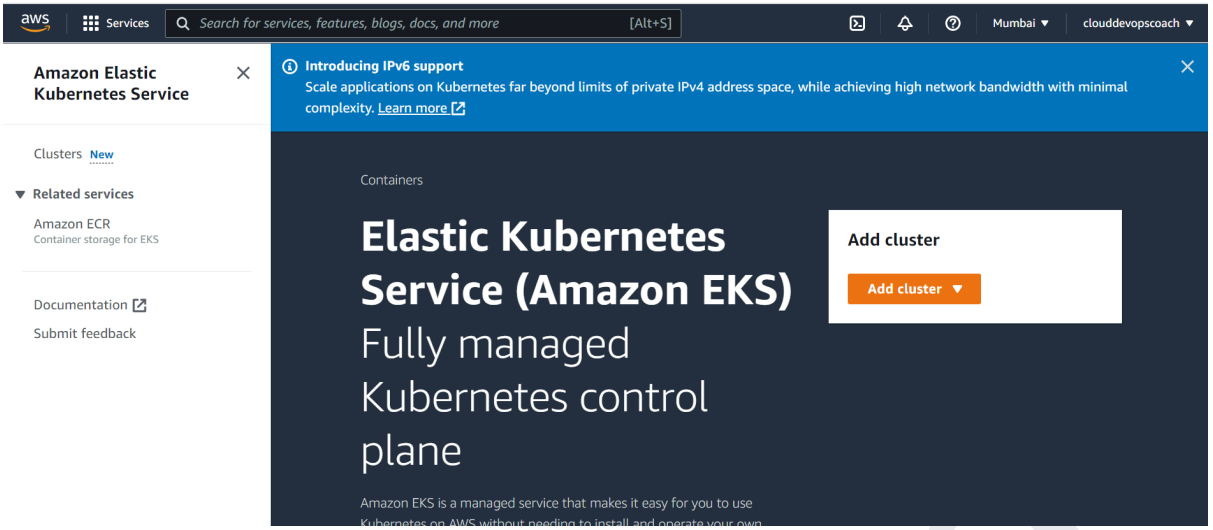
minikube start

https://minikube.sigs.k8s.io/docs/start/
https://storage.googleapis.com/minikube/releases/latest/minikube-installer.exe

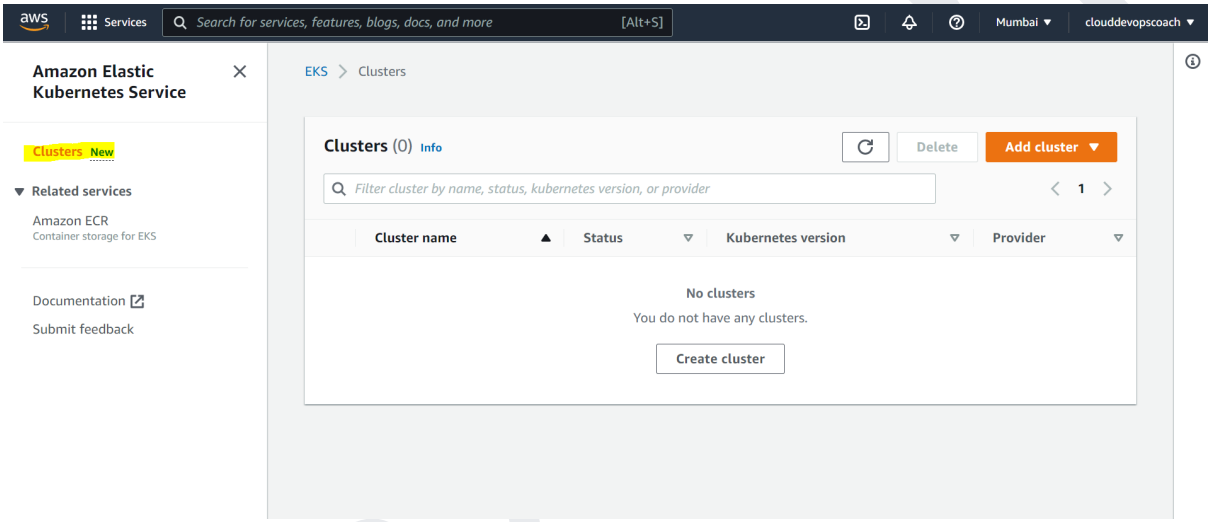
Creating EKS Cluster and Connecting to it.

Login to AWS Management Console

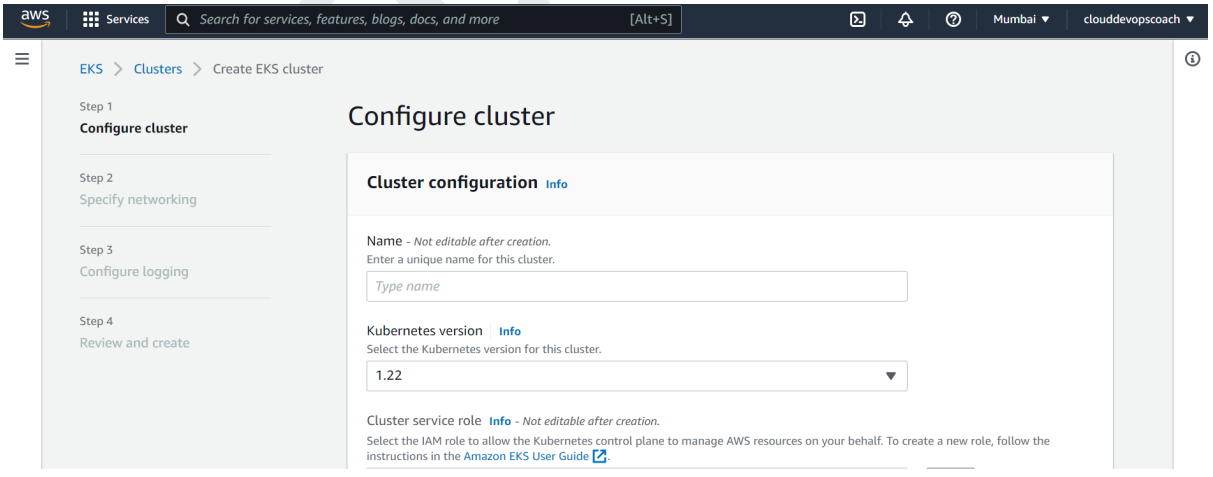
Go to Elastic Kubernetes Service



Go to Clusters



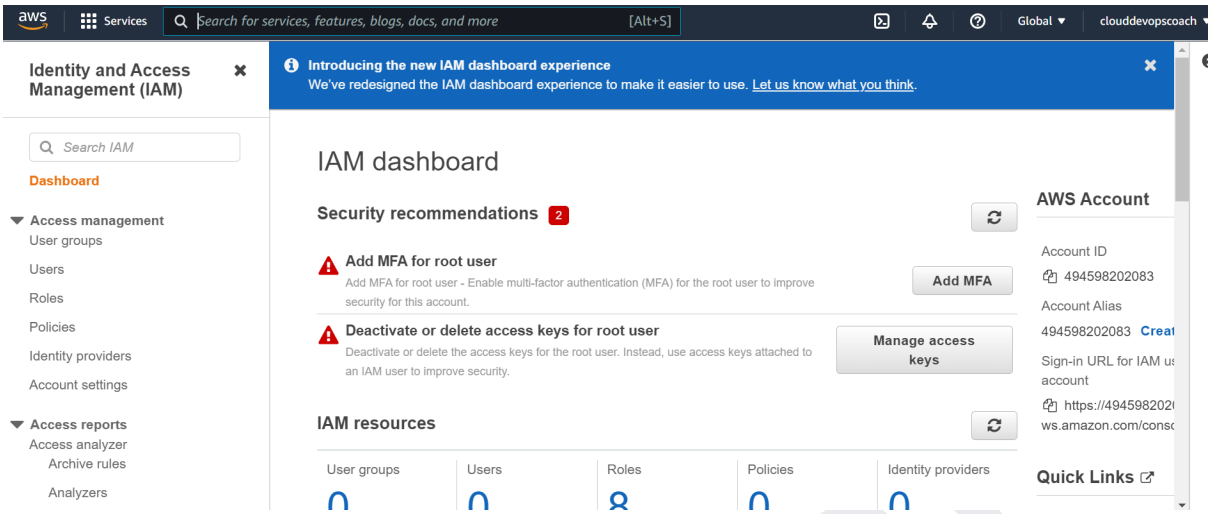
Click on Create Cluster



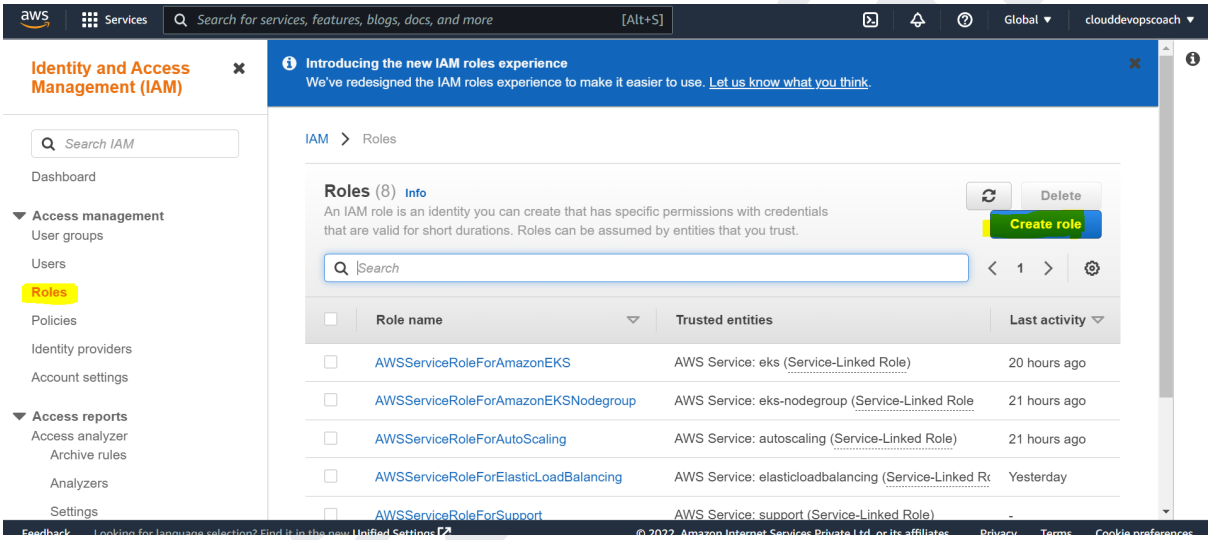
Here we have give name to the cluster and select Kubernetes Version

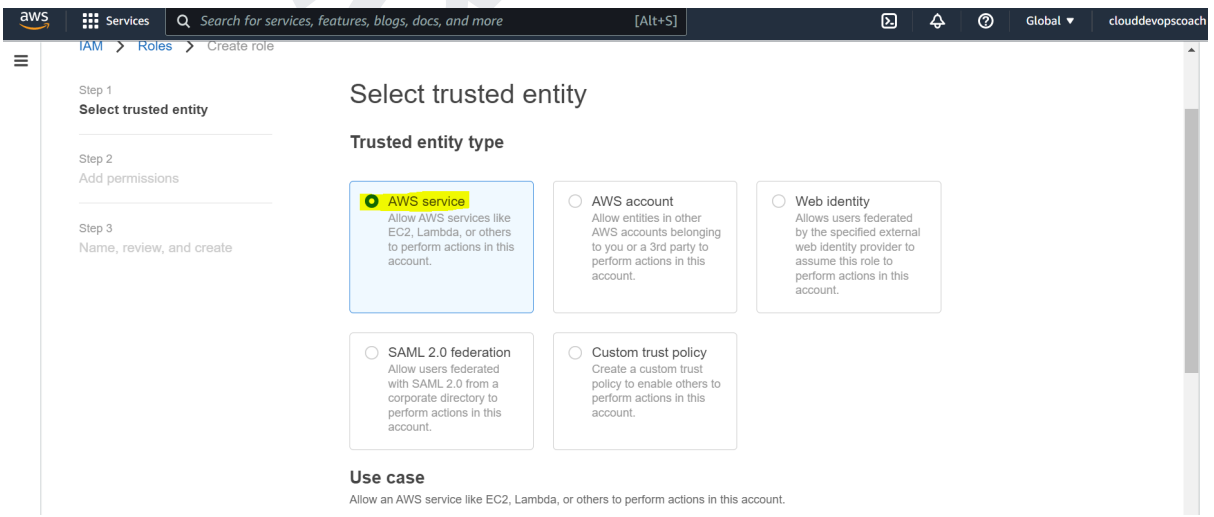
In this it will also ask for Cluster service role
We have to create a role , to give permission to EKS cluster to manage AWS resources.
So, create a role

Open a new tab and
Go to IAM Console

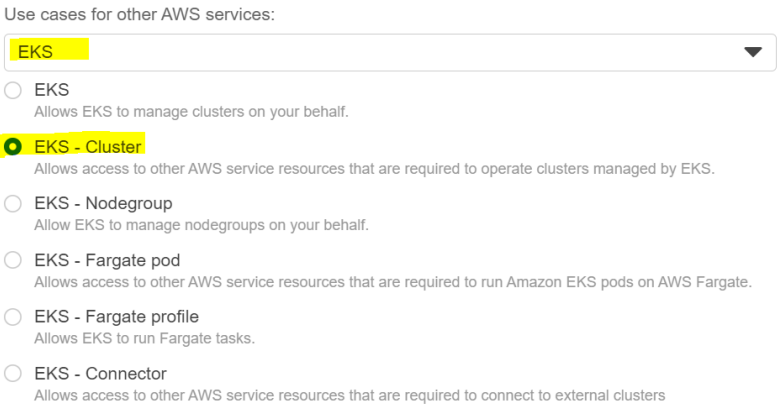


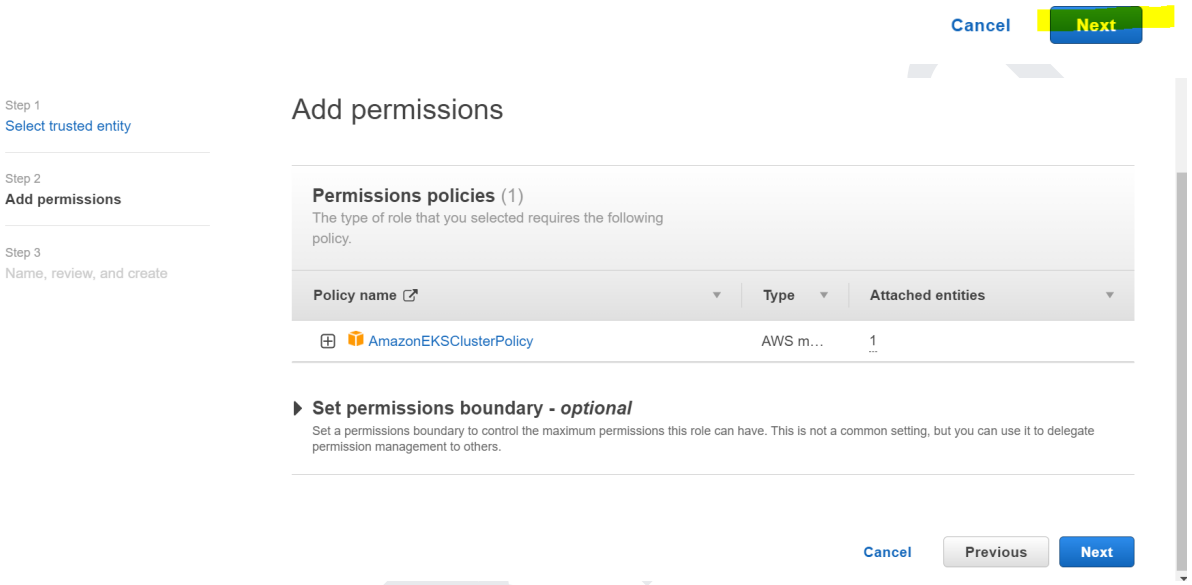
Click on Roles and Create Role



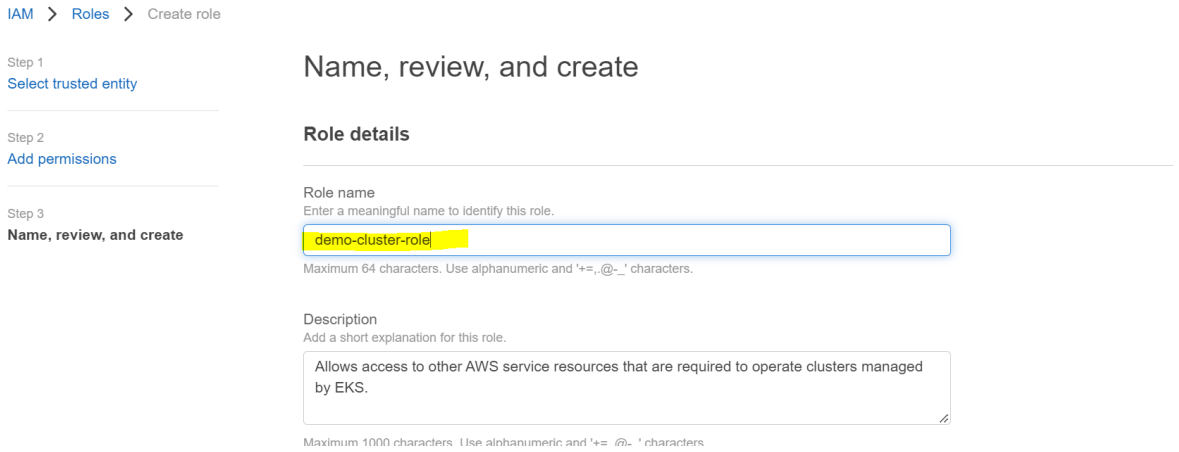


Choose EKS from Dropdown and Choose EKS Cluster and click on Next





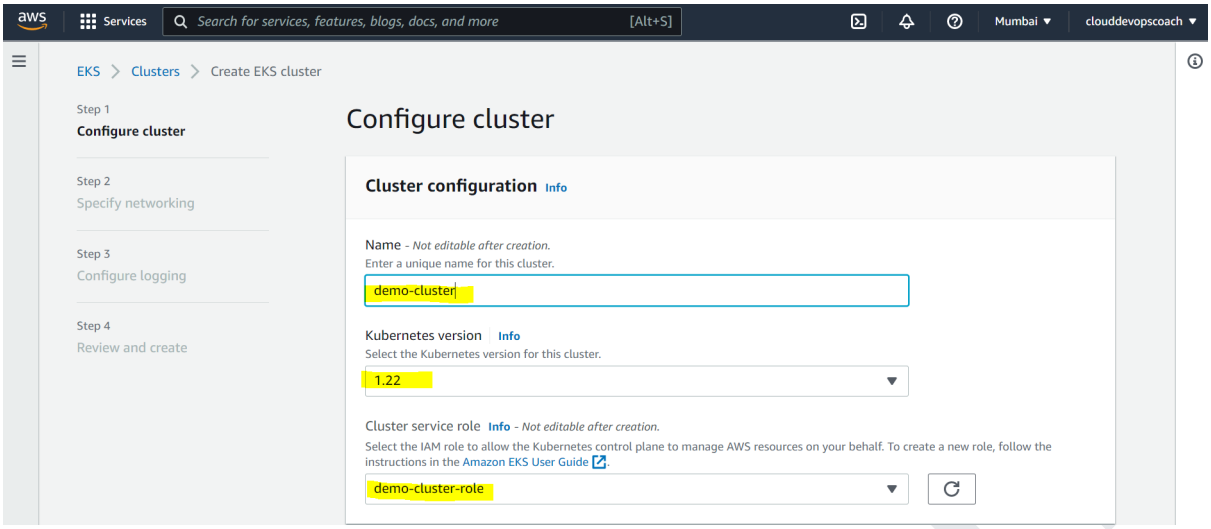
Click on Next
Skip the Tags Section
Give a name to the role in Role Name field. In this case demo-cluster-role



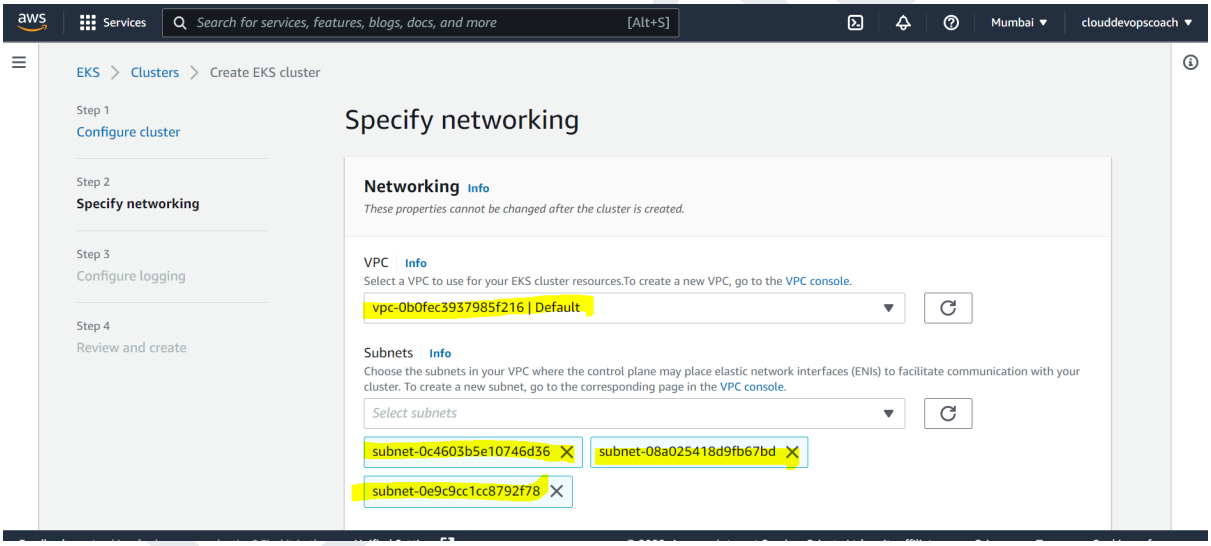
Click on Create role

Now Go back to Elastic Kubernetes Service page
Enter the name for the cluster. In my case it is demo-cluster
Choose a Kubernetes Version.

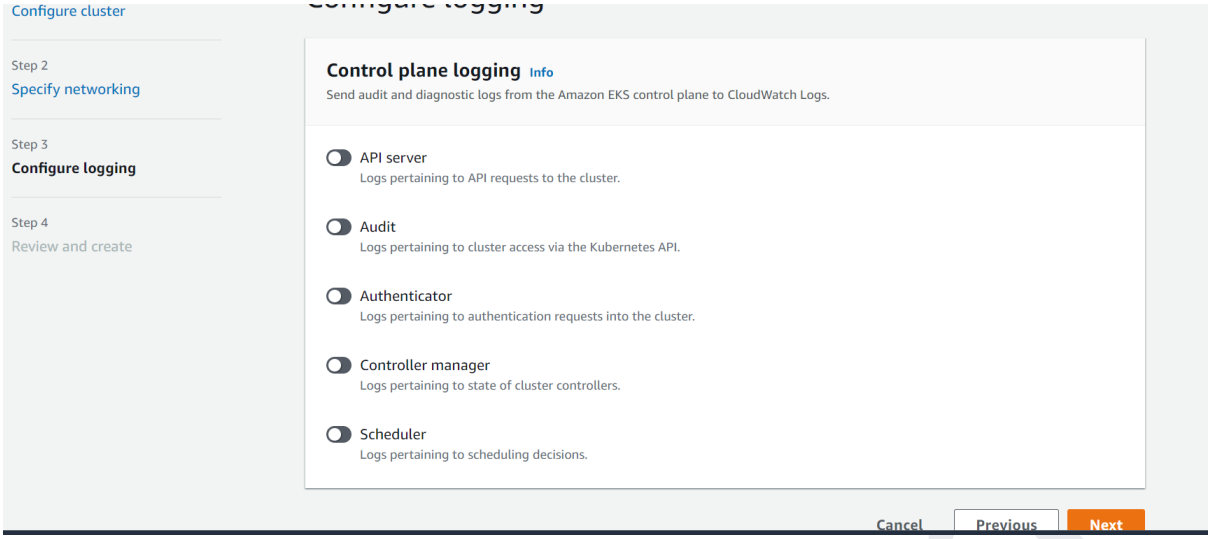
Choose the Cluster service role that you have created earlier i.e. demo-cluster-role



Click on Next
In Networking Section, Choose the default VPC and Subnets. They are auto populated.



Keep everything default and Click on Next

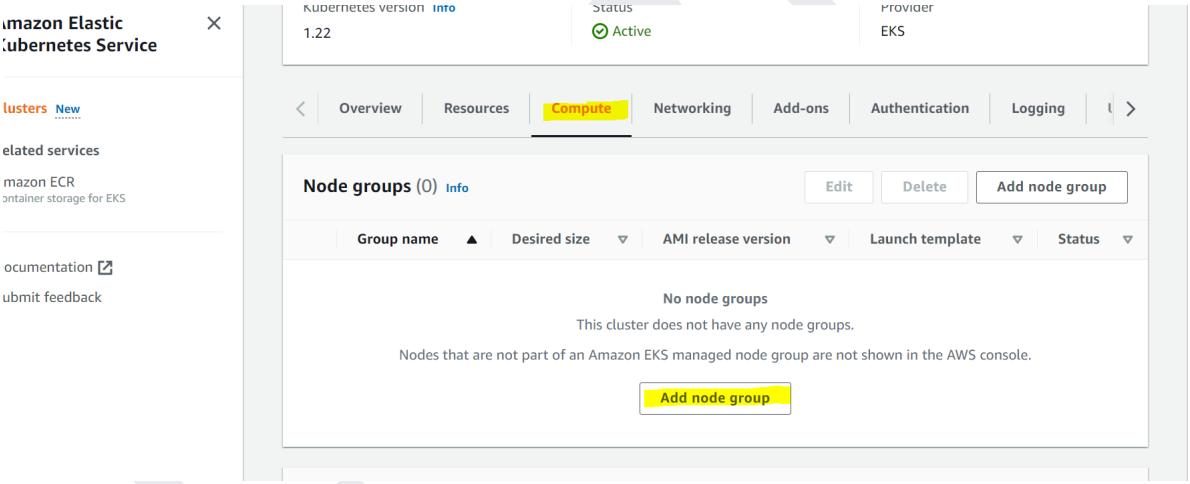


Keep everything default. I want to keep it off. Click on Next

Click on Create.
Wait for the cluster to be created. It usually take 10 to 15 minutes of time.

Once Cluster is created. We have to add a Node Group .

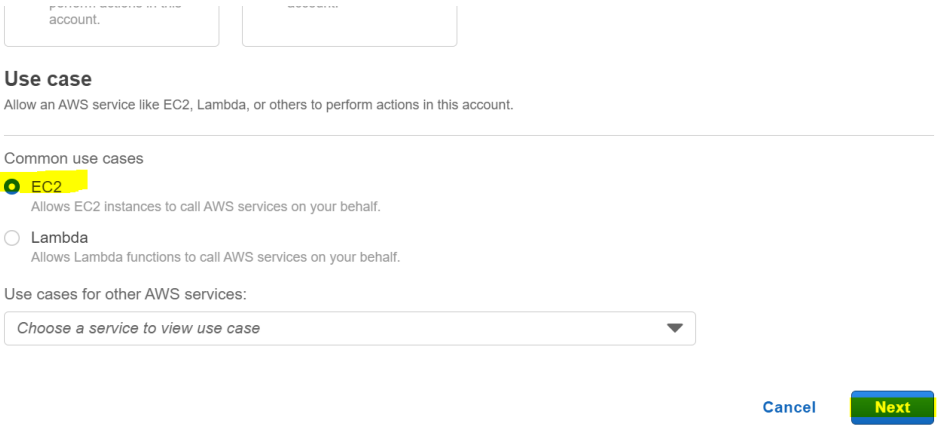
For that go to compute section of the cluster and Click on Add Node Group.



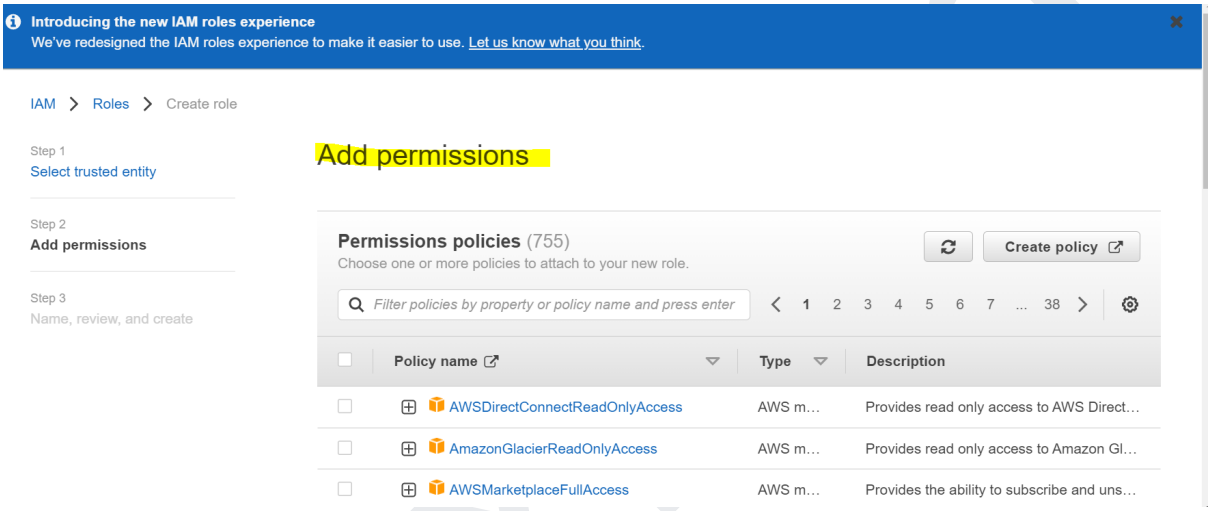
Give a name to the node group. Here it's demo-node-group.

Again we need IAM Role for Node groups as well.
Open new tab and go to IAM console.

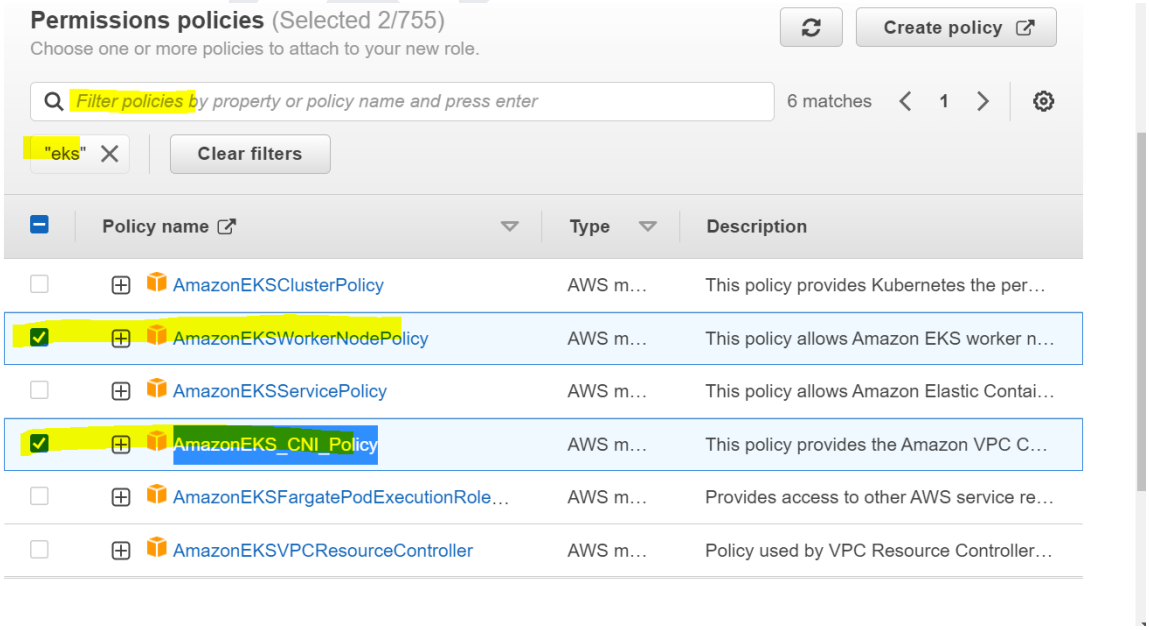
Go to Roles and click on Create role
Choose EC2 as use case and click on Next



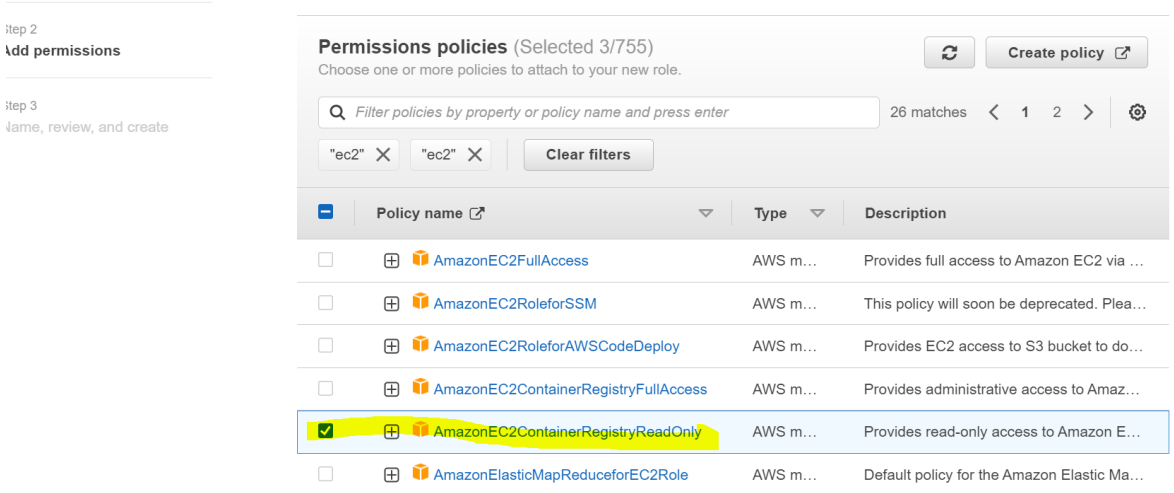
In Permissions policies section, we have to choose the policies.



In Filter/Find field type EKS, it will give some list.
Choose
We have to choose AmazonEKSWorkerNodePolicy and AmazonEKS_CNI_Policy



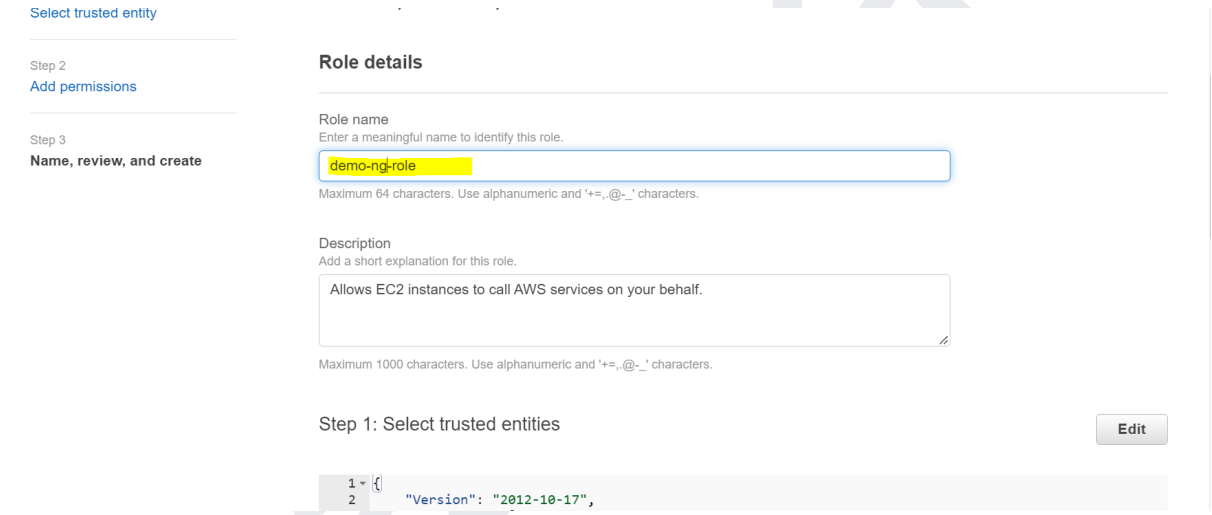
Clear the filter and type EC2. Now choose AmazonEC2ContainerRegistryReadOnly



Click on Next

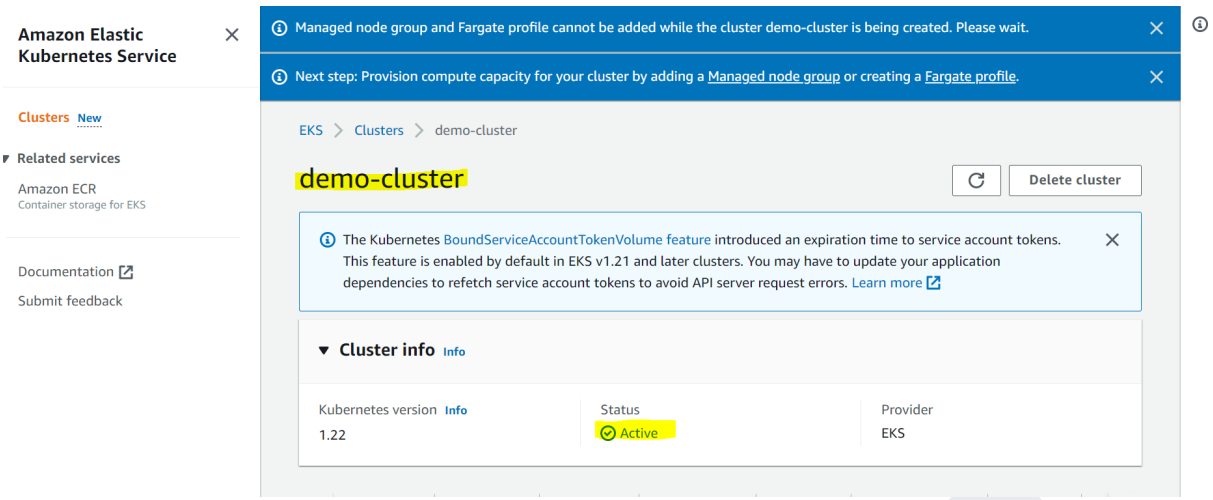
Leave the tags, click on Next

Give a name to the role



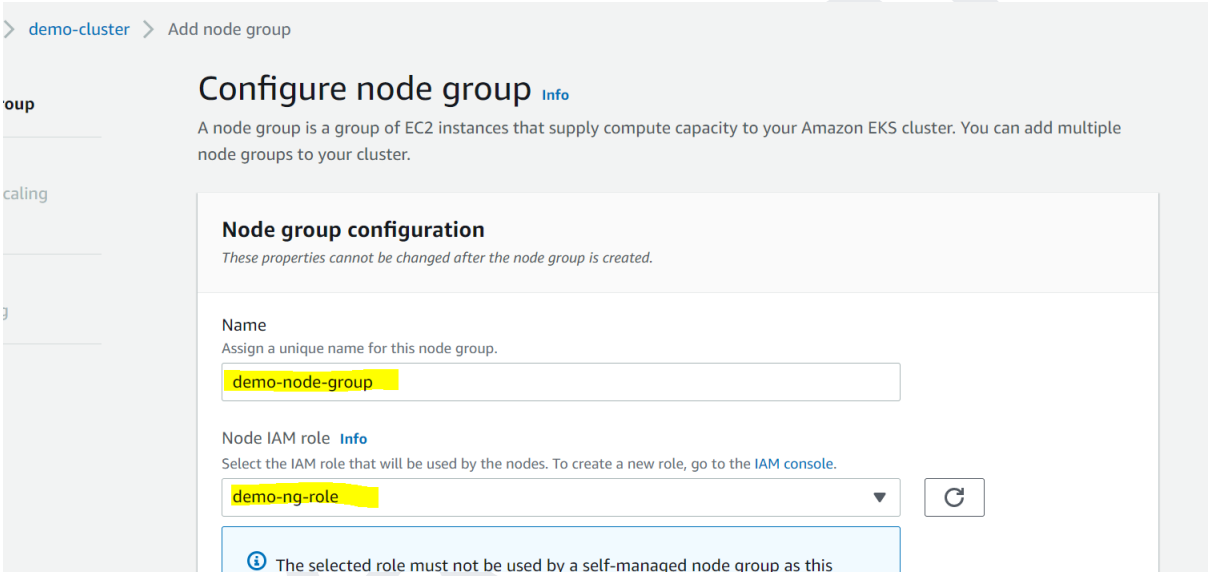
Click on Create role.

Now go to your cluster and check whether it came into Active status.



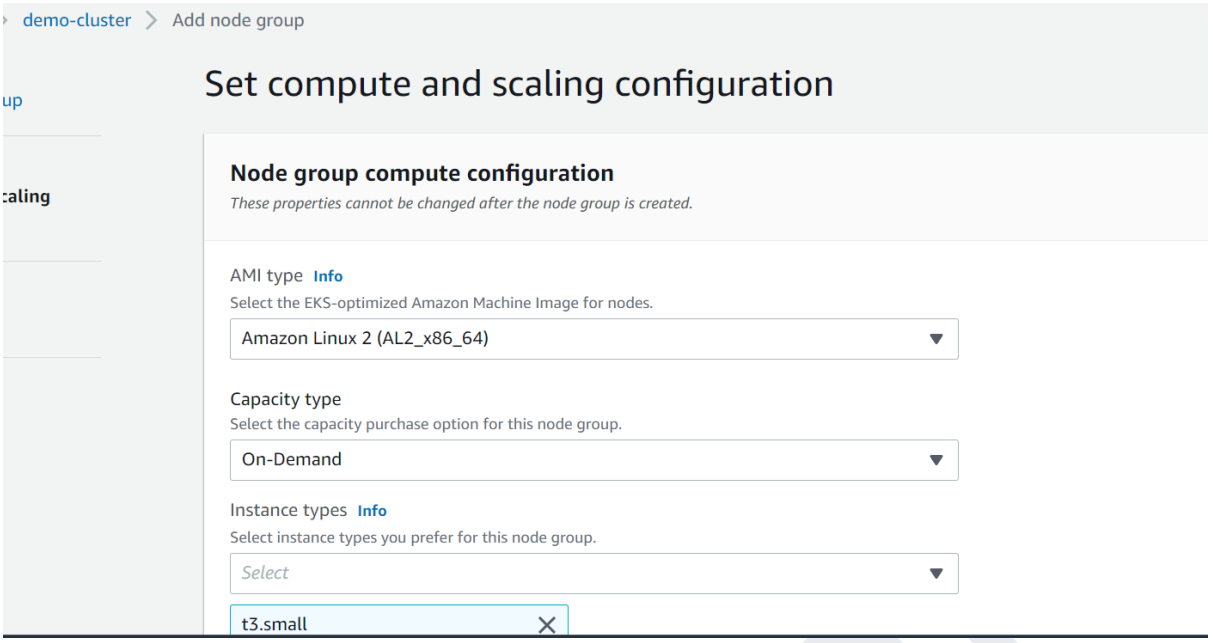
Now go to compute section of the cluster, click on Add Node Group.

Name the Node Group and select Node IAM role from the drop down



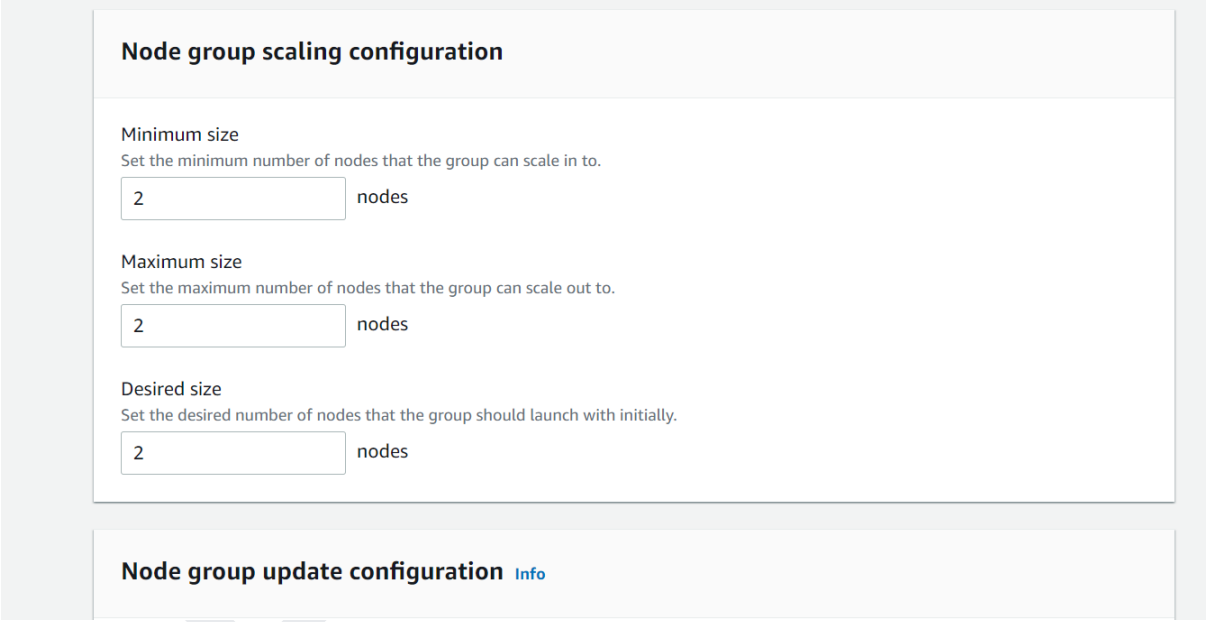
Click on Next

In the Set compute section

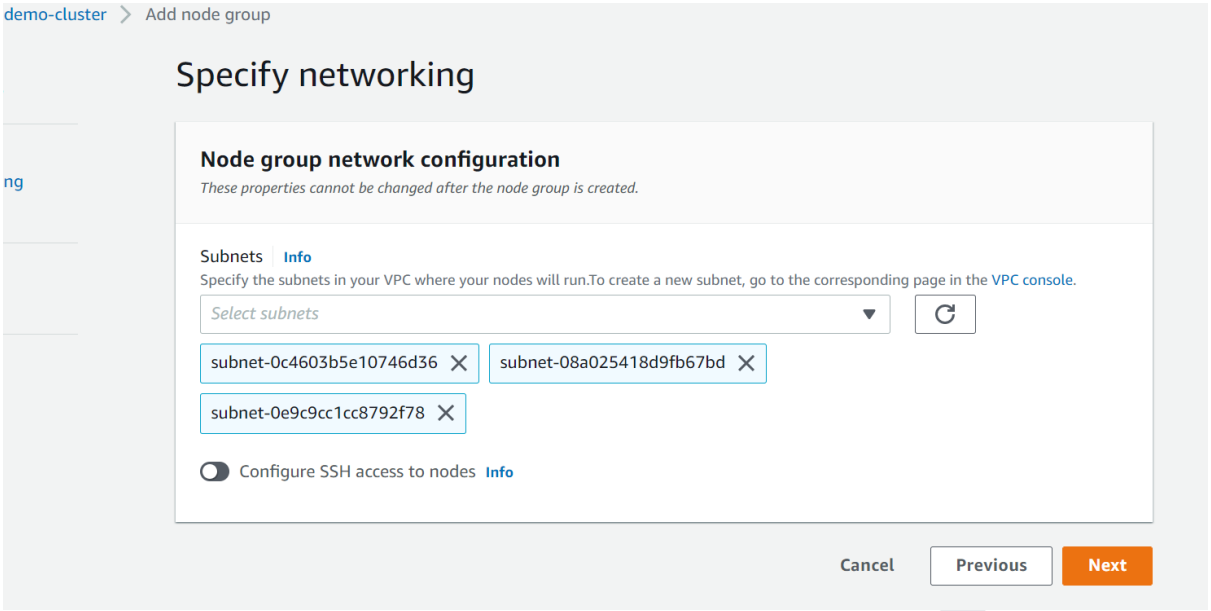


Select AMI type, Instance type. In my case it's t3.small

And select scaling configuration

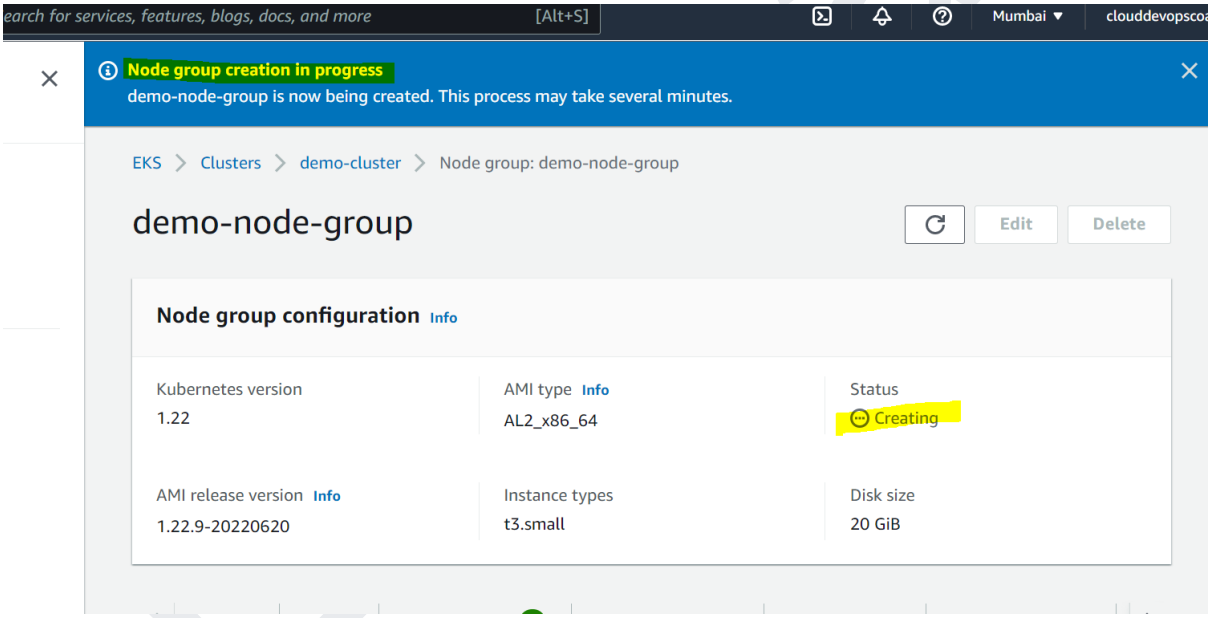


I kept it 2 nodes
Click on Next

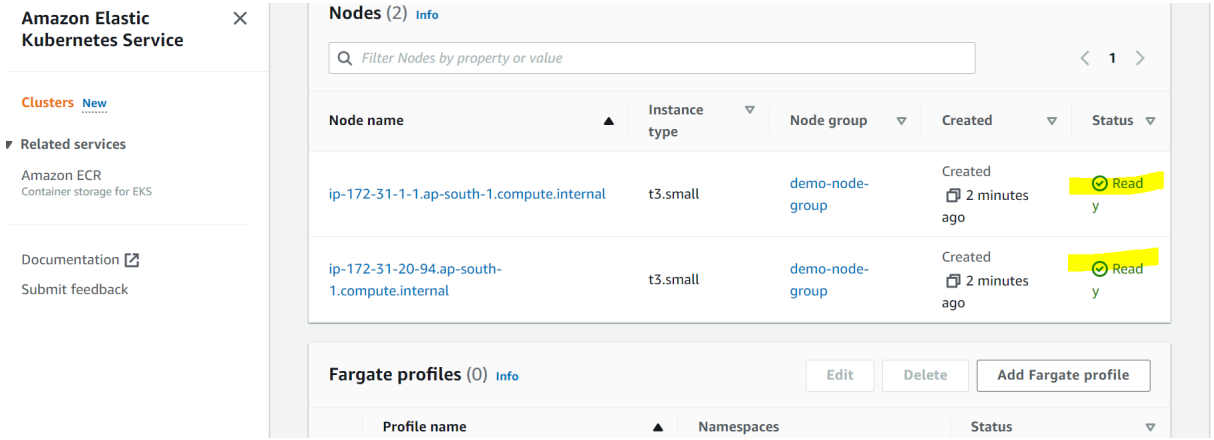


Keep default , or choose if you want to have custom subnets.

Click on Next and then Create



You will see the status as Creating. It takes about 5 to 10 minutes depending on the size of the node group etc. Once it is created, you will get status as Active.
Click on Node Group



How to connect to EKS cluster ?

You should have installed AWS CLI and kubectl

Installing AWS CLI

https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html

Download and run the AWS CLI MSI installer for Windows (64-bit):

https://awscli.amazonaws.com/AWSCLIV2.msi

To confirm the installation, open the Start menu, search for `cmd` to open a command prompt window, and at the command prompt use the

`aws --version` command.

Installing kubectl

https://kubernetes.io/docs/tasks/tools/install-kubectl-windows/#install-kubectl-binary-with-curl-on-windows

https://kubernetes.io/docs/tasks/tools/install-kubectl-linux/

Download the file with the following link

https://dl.k8s.io/release/v1.24.0/bin/windows/amd64/kubectl.exe

And move this to System32 folder which will be located in C:/

Now go to command line and type

```
kubectl version --client
```

Connecting to EKS Cluster

Verify that AWS CLI is installed on your system:
`aws --version`

Configuring the AWS CLI

https://docs.aws.amazon.com/cli/latest/userguide/cli-configure-quickstart.html

In command line type`

aws configure

Create or update the kubeconfig file for your cluster:

```
aws eks --region region update-kubeconfig --name cluster_name

For example

aws eks --region ap-south-1 update-kubeconfig --name demo-cluster
Automatically kubectl will be switched to this context.
If not

kubectl config get-contexts
kubectl config use-context <your-context-name>

kubectl get nodes
```

Imperative Deployment Approach

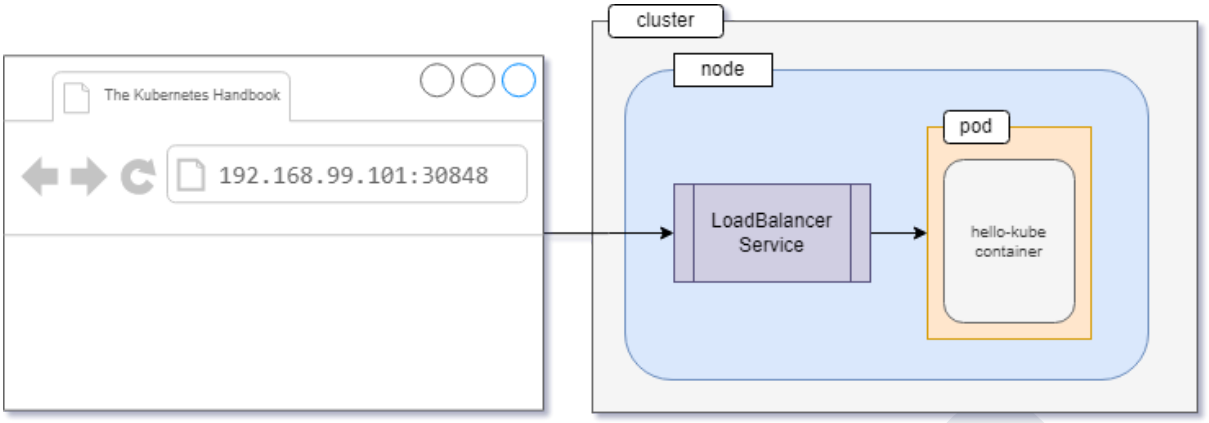
In this approach we have to execute every command one after the other manually. Taking an imperative approach defies the entire point of Kubernetes. But still let's experience this for an understanding

kubectl run hello-nginx --image=nginx
kubectl expose pod hello-nginx --type=LoadBalancer --port=80
kubectl exec -it hello-nginx -- bash

Run service tunnel

minikube service hello-kube

minikube service hello-nginx runs as a process, creating a tunnel to the cluster. The command exposes the service directly to any program running on the host operating system.



kubectl create deployment my-nginx --image=nginx
kubectl expose deployment my-nginx --port 80

Updating objects imperatively

kubectl edit deployment my-nginx
kubectl scale deployment my-nginx --replicas=5
kubectl set image deployment my-nginx nginx=nginx:1.18

kubectl create -f nginx.yml
kubectl replace -f nginx.yml
kubectl delete -f nginx.yml

Declarative Deployment Approach

An ideal approach to deployment with Kubernetes is the declarative approach. In it you, as a developer, let Kubernetes know the state you desire your servers to be in and Kubernetes figures out a way to implement that.

Kubernetes performs container orchestration by using definition files. Definition files are yaml files .
Definition file, will have 4 top level elements

1. apiVersion:
2. kind:
3. metadata:
4. spec:

apiVersion:

Depending on the kubernetes object we want to create, there is a corresponding code library we want to use.

apiVersion refers to code library

Kind	apiVersion
Pod	v1
Service	v1
NameSpace	v1
Secrets	v1
RepliaSet	apps/v1
Deployment	apps/v1

kind:

Refers to the kubernetes object which we want to create.
Ex: Pod, Replicaset, service etc

metadata:

Additional information about the kubernetes object
like name, labels etc

spec:

Contains docker container related information like image name, environment variables, port mapping etc.

pod-definition.yml

```
apiVersion: v1
kind: Pod
metadata:
 name: myapp-pod
 labels:
  app: myapp
  type: front-end
spec:
  containers:
  - name: nginx-container
    image: nginx
```

kubectl create -f pod-definition.yml
kubectl get pods
Kubectl describe pod myapp-pod
kubectl get pods -o wide
kubectl delete pod myapp-pod

kubectl run redis --image=redis --dry-run=client -o yaml > pod.yaml
vi pod.yaml
kubectl apply -f pod.yaml
kubectl get pod redis -o yaml
kubectl edit pod redis

rc-definition.yaml

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: myapp-rc
  labels:
    app: myapp
    type: front-end
spec:
  template:
    metadata:
     name: myapp-pod
     labels:
      app: myapp
      type: front-end
    spec:
      containers:
      - name: nginx-container
        image: nginx
  replicas: 3
```

kubectl create -f rc-definition.yml
kubectl get replicationcontroller
kubectl get rc
kubectl describe rc myapp-rc
kubectl delete rc myapp-rc

rs-definition.yml

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

kubectl apply -f rs-definition.yaml
kubectl get rs
Edit the file and set replicas to 5
kubectl replace -f rs-definition.yaml
Or
kubectl apply -f rs-definition.yaml
kubectl scale --replicas=2 replicaset myapp-replicaset
kubectl scale --replicas=6 replicaset myapp-replicaset
kubectl scale --replicas=3 -f apply rs-definition.yaml

kubectl scale --replicas=3 -f rs-definition.yaml
kubectl describe replicaset myapp-replicaset
Kubectl scale replicaset --replicas=5 myapp-replicaset

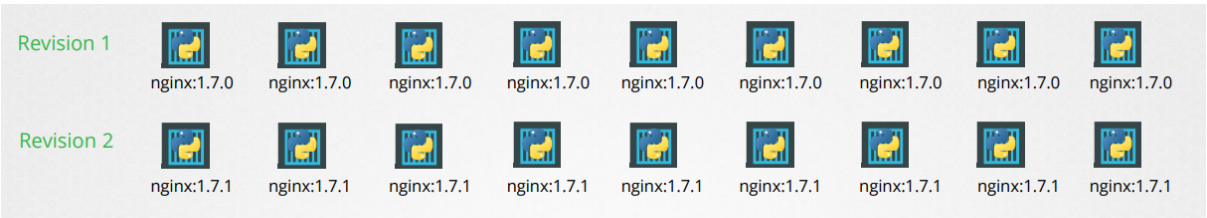
Deployments

Allows rolling updates
Exactly same as replicaset definition except kind

deployment-definition.yaml

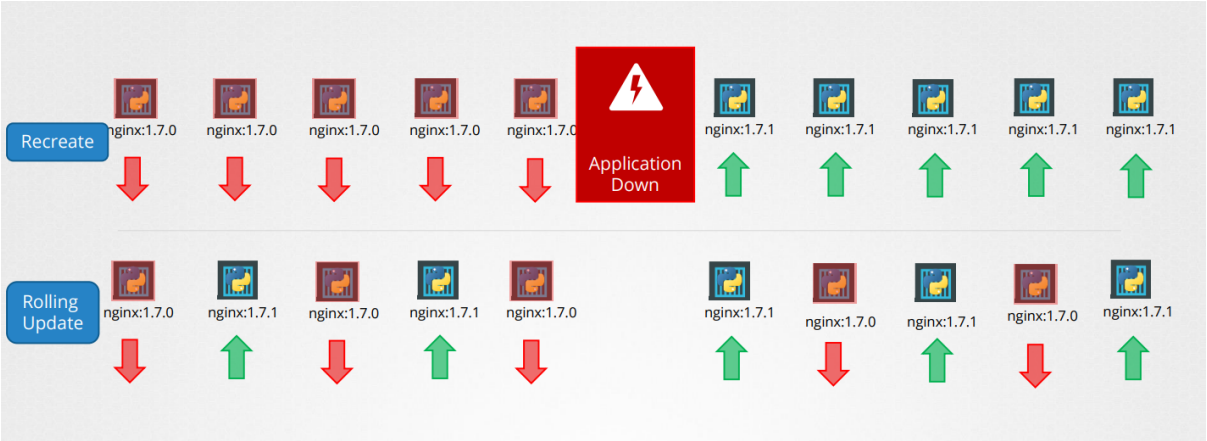
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
  labels:
    app: myapp
    type: front-end
spec:
  template:
    metadata:
     name: myapp-deployment
     labels:
      app: myapp
      type: front-end
    spec:
      containers:
      - name: nginx-container
        image: nginx
  replicas: 3
  selector:
    matchLabels:
      type: front-end
```

maxSurge specifies the maximum number (or percentage) of pods above the specified number of replicas.
`maxUnavailable` declares the maximum number (or percentage) of unavailable pods during the update.
kubectl rollout history deployment myapp-deployment
kubectl rollout status deployment myapp-deployment
kubectl rollout undo deployment myapp-deployment
kubectl rollout undo deploy myapp-deployment --to-revision=3
Rollout and Versioning



kubectl rollout status deployment myapp-deployment
kubectl rollout history deployment myapp-deployment

Deployment Strategy : Recreate and Rolling update



In recreate approach, pods will be down for sometime.

deploy-def.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
  labels:
    app: myapp
    type: front-end
spec:
  template:
    metadata:
     name: myapp-deployment
     labels:
      app: myapp
      type: front-end
    spec:
      containers:
      - name: nginx-container
        image: nginx:1.7.0
  replicas: 3
  selector:
    matchLabels:
      type: front-end
```

kubectl create -f deploy-def.yaml

In this approach, kubernetes adopts rolling update strategy by default. No downtime.
kubectl describe deployment myapp-deployment

kubectl set image deployment/myapp-deployment image=nginx:1.9.1
kubectl describe deployment myapp-deployment

Observe the difference

Now,
kubectl get replicasets

To rollback the upgrade process
kubectl rollout undo deployment/myapp-deployment
kubectl get replicasets

Summary Commands

kubectl create -f deploy-def.yaml
kubectl get deployments
kubectl apply -f deploy-def.yaml
kubectl set image deployment/myapp-deployment nginx=nginx:1.9.1

kubectl rollout status deployment myapp-deployment
kubectl rollout history deployment myapp-deployment
kubectl rollout undo deployment myapp-deployment
kubectl rollout restart deployment myapp-deployment

kubectl describe deployment myapp-deployment
Edit the deployment using kubectl edit deployment myapp-deployment and change the image version for updating.
kubectl get pods
Now
Edit the deployment and change the deployment strategy t o Recreate

kubectl create -f deployment-definition.yaml
kubectl get deployments
kubectl get replicaset #same pods we get
kubectl describe deployment myapp-deployment
kubectl delete deployment myapp-deployment

kubectl create deployment httpd-frontend --image=httpd:2.4-alpine
kubectl scale deployment --replicas=3 httpd-frontend
kubectl get deployments
Interacting with Pod
kubectl exec -it <pod-name> -- bash

Daemon Sets

A *DaemonSet* ensures that all Nodes run a copy of a Pod. As nodes are added to the cluster, Pods are added to them. As nodes are removed from the cluster, those Pods are garbage collected. Deleting a DaemonSet will clean up the Pods it created.

Some typical uses of a DaemonSet are:

- running a logs collection daemon on every node
- running a node monitoring daemon on every node

Daemon Sets use case - kube-proxy

daemon-set-definition.yml

Definition is exactly the same as ReplicaSet except kind. Kind is DaemonSet

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: monitoring-daemon
spec:
  template:
    metadata:
     name: monitoring-agent
     labels:
      app: monitoring-agent
    spec:
      containers:
      - name: monitoring-agent
        image: monitoring-agent
  selector:
    matchLabels:
      type: monitoring-agent
```

kubectl create -f daemon-set-definition.yaml

kubectl get daemonsets
kubectl describe daemonsets monitoring-daemon
kubectl get ds --all-namespaces
kubectl -n kube-system get pods -o wide

kubectl -n kube-system describe ds weave-net

Q. Deploy a DaemonSet for FluentD Logging.
Name: elasticsearch, namespace: kube-system, image: k8s.gcr.io/fluentd-elasticsearch:1.20

kubectl create deployment elasticsearch --image=k8s.gcr.io/fluentd-elasticsearch:1.20 --dry-run=client -o yaml > elastic.yml
vi elastic.yml
Replace Deployment with DaemonSet in kind.
Add namespace: kube-system in metadata

kubectl apply -f elastic.yml
kubectl -n kube-system get ds elasticsearch

Namespaces

In Kubernetes, namespaces provide a mechanism for isolating groups of resources within a single cluster. Names of resources need to be unique within a namespace, but not across namespaces. Namespace-based scoping is applicable only for namespaced objects (e.g. Deployments, Services, etc) and not for cluster-wide objects (e.g. StorageClass, Nodes, PersistentVolumes, etc).
we get the following namespaces automatically.

default
kube-system
Kube-public
Kubernetes starts with four initial namespaces:

default The default namespace for objects with no other namespace

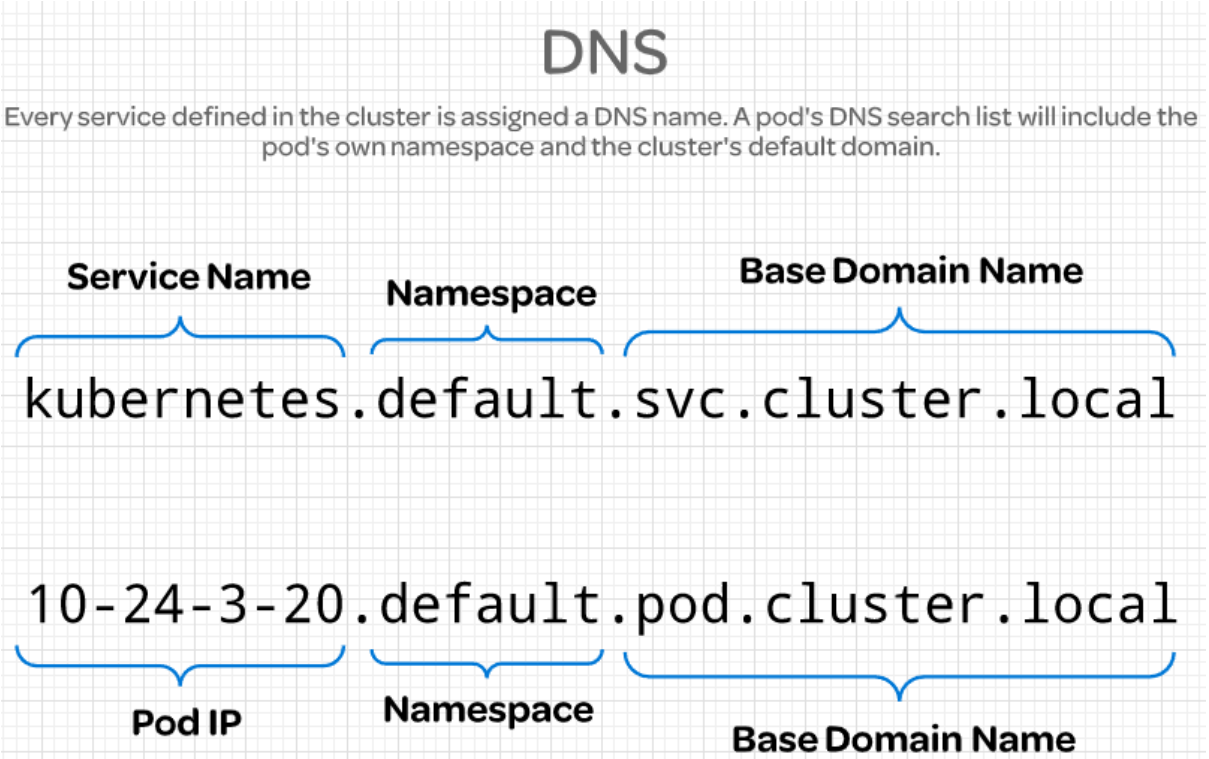
kube-system The namespace for objects created by the Kubernetes system

kube-public This namespace is created automatically and is readable by all users (including those not authenticated). This namespace is mostly reserved for cluster usage, in case that some resources should be visible and readable publicly throughout the whole cluster. The public aspect of this namespace is only a convention, not a requirement.

kube-node-lease This namespace holds Lease objects associated with each node. Node leases allow the kubelet to send heartbeats so that the control plane can detect node failure

Namespaces and DNS

When you create a Service, it creates a corresponding DNS entry. This entry is of the form **<service-name>.<namespace-name>.svc.cluster.local**, which means that if a container only uses <service-name>, it will resolve to the service which is local to a namespace.
This is useful for using the same configuration across multiple namespaces such as Development, Staging and Production.
If you want to reach across namespaces, you need to use the fully qualified domain name (FQDN).



mysql.connect("db-service")

mysql.connect("db-service.dev.svc.cluster.local")

kubectl get pods --namespace=kube-system

kubectl create -f pod-definition.yml --namespace=dev

By using yaml file ---> use namespace option ins metadata

ns-dev.yml

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: dev
```

```
```

kubectl create -f ns-dev.yml
kubectl get ns
kubectl delete namespace dev

kubectl create namespace dev

How to switch from default namespace to dev namespace.

kubectl config set-context $(kubectl config current-context) --namespace=dev
kubectl config set-context --current --namespace=dev

kubectl get pods

kubectl get pods --all-namespaces
or
kubectl get pods -A

Resource Quota

compute-quota.yml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-quota-dev
  namespace: dev
spec:
  hard:
    pods: "5"
    requests.cpu: "1"
    limits.cpu: "2"
    requests.memory: 1Gi
    limits.memory: 2Gi
```

kubectl -n dev get pod myapp-pod-dev -o yaml to get the pod configuration in yaml format.
kubectl create -f compute-quota.yml
kubectl get ns --no-headers | wc -l

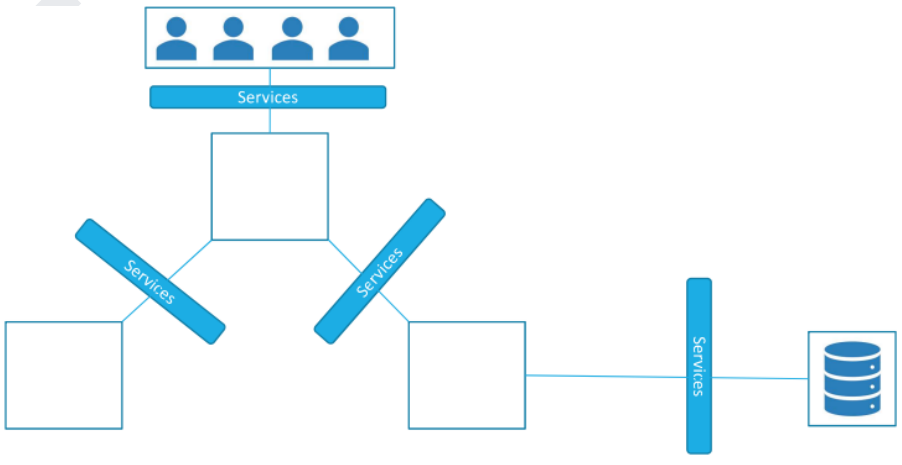
kubectl get pods -n research --no-headers

kubectl run redis --image=redis --dry-run=client -o yaml > pod.yml
vi pod.yml
add namespace finance in metadata
kubectl apply -f pod.yml

Resource for Container

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
spec:
  containers :
  - name: data-processor
    image: nginx
    resources:
      requests:
        memory: "256Mi"
        cpu: 5m
      limits:
        memory: "512Gi"
        cpu: 10m
```

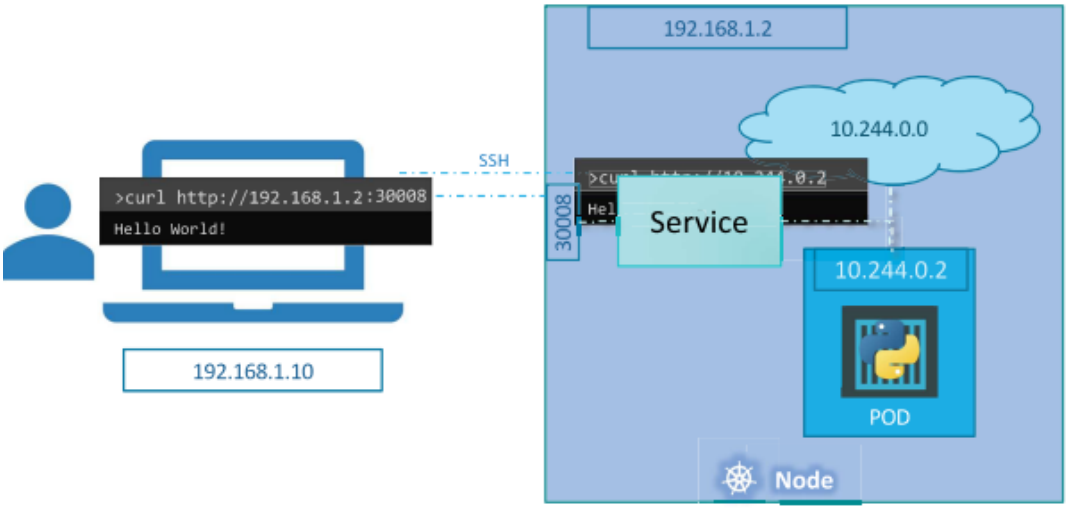
Services



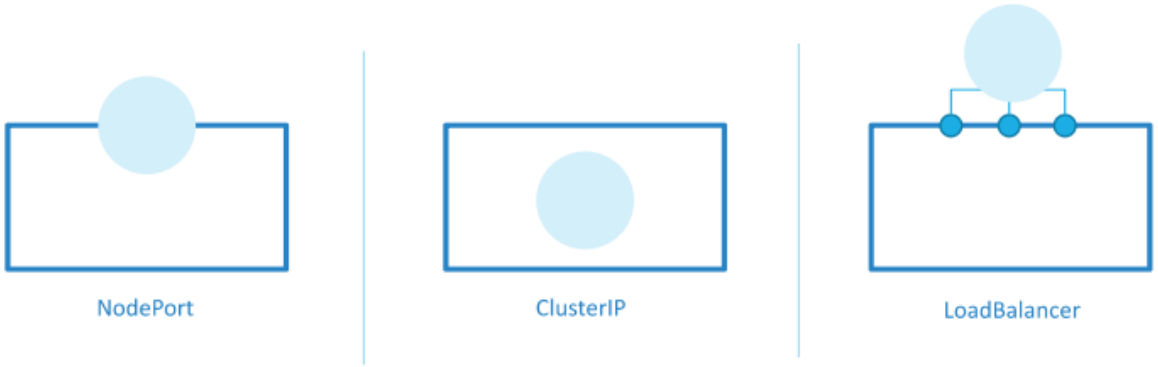
Kubernetes Services enable communication between various components within and outside of the application. Kubernetes services helps us connect applications together with other applications or users.

For example our application has groups of pods running various sections such as a group for serving front end load to users and other group for running back end processes and a third group connecting to an external data source.

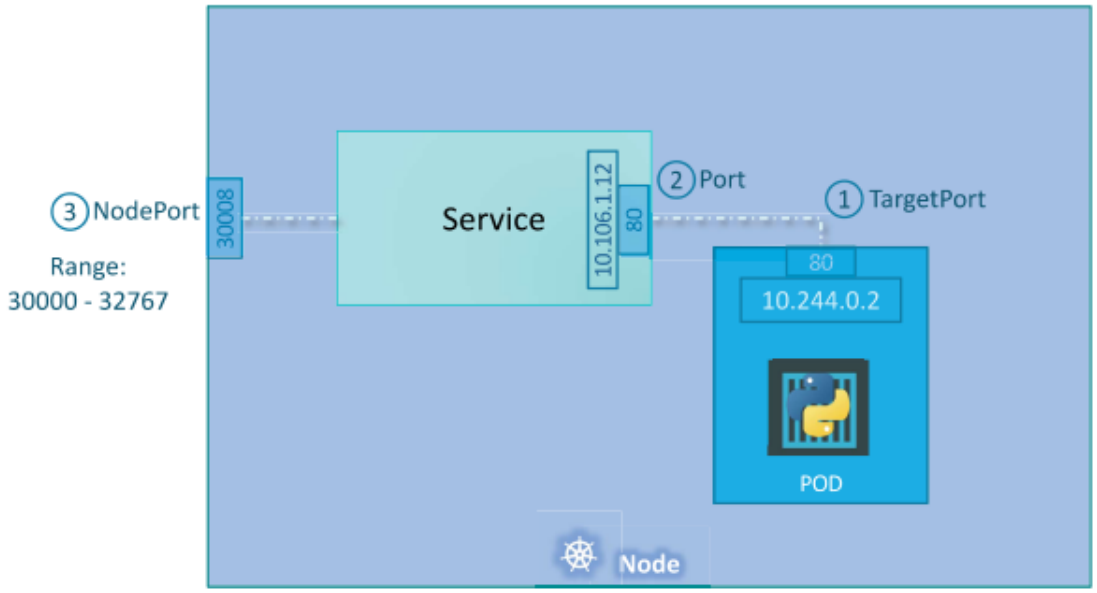
It is services that enable connectivity between these groups of pods. Services enable the front end application to be made available to end users, it helps communication between back end and front end pods and helps in establishing connectivity to an external data source.



Service types : NodePort, ClusterIP, LoadBalancer



NodePort:



Service Definition File:

deployment-definition.yaml

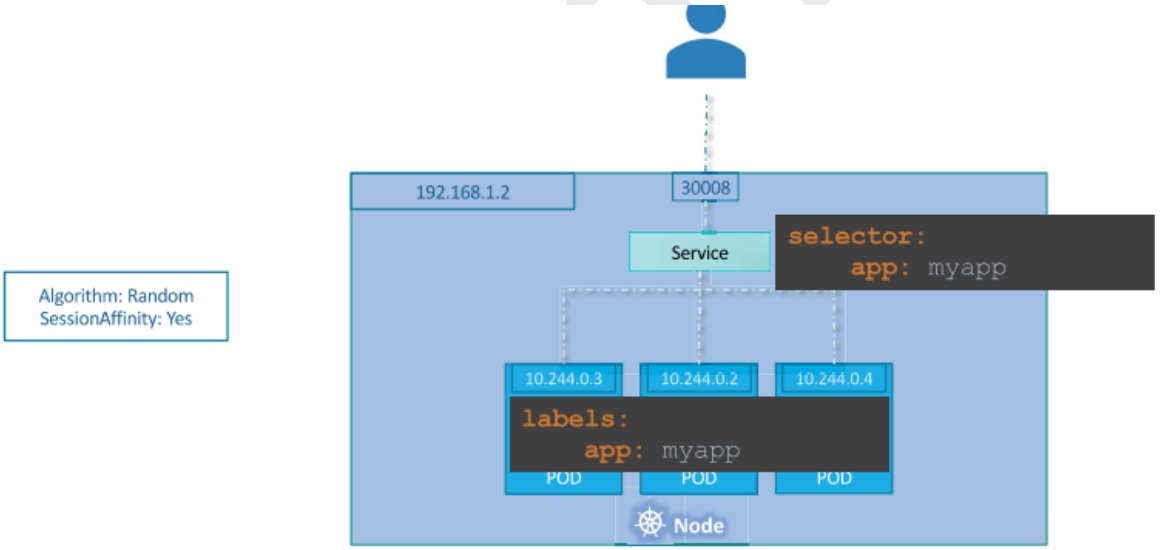
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
  labels:
    app: myapp
    type: front-end
spec:
  template:
    metadata:
     name: myapp-deployment
     labels:
      app: myapp
      type: front-end
    spec:
      containers:
      - name: nginx-container
        image: nginx
  replicas: 3
  selector:
    matchLabels:
      type: front-end
```

service-definition.yml

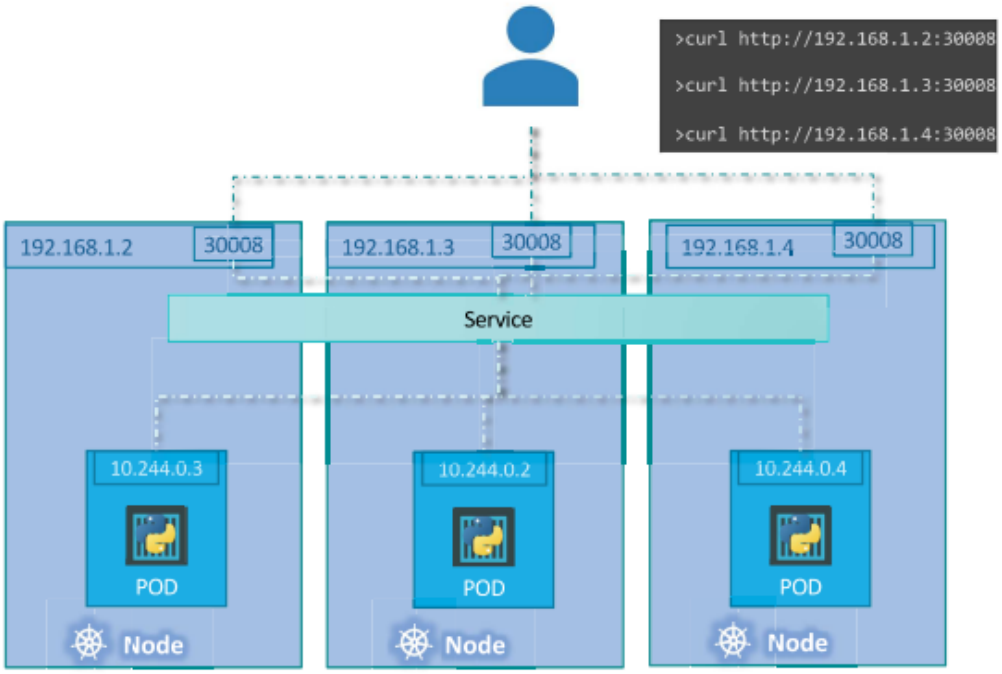
```yaml
apiVersion: v1
kind: Service
metadata:
  name: myapp-service
spec:
  type: NodePort
  ports: # an array
    - targetPort: 80
      port : 80 # port on service object
      nodePort: 30008
  selector:
    app: myapp
    type: front-end
```

kubectl create -f service-definition.yml
kubectl get service
kubectl describe service myapp-service
Curl http://<node-ip-address>:30008

What if we have multiple pods for same service ?

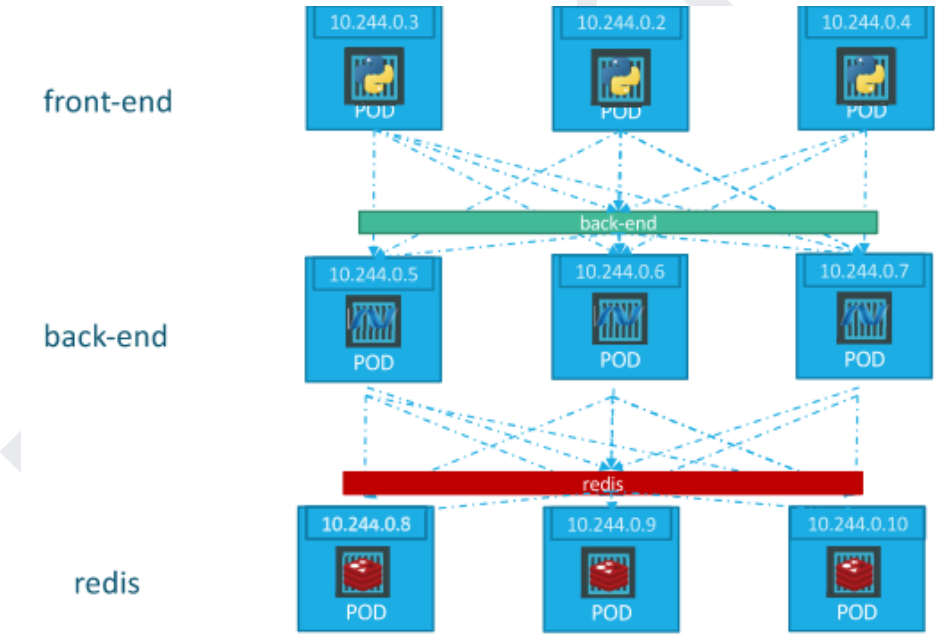


What if the pods are distributed across multiple nodes ?



kubectl delete svc myapp-service

ClusterIP:



clusterip-definition.yml

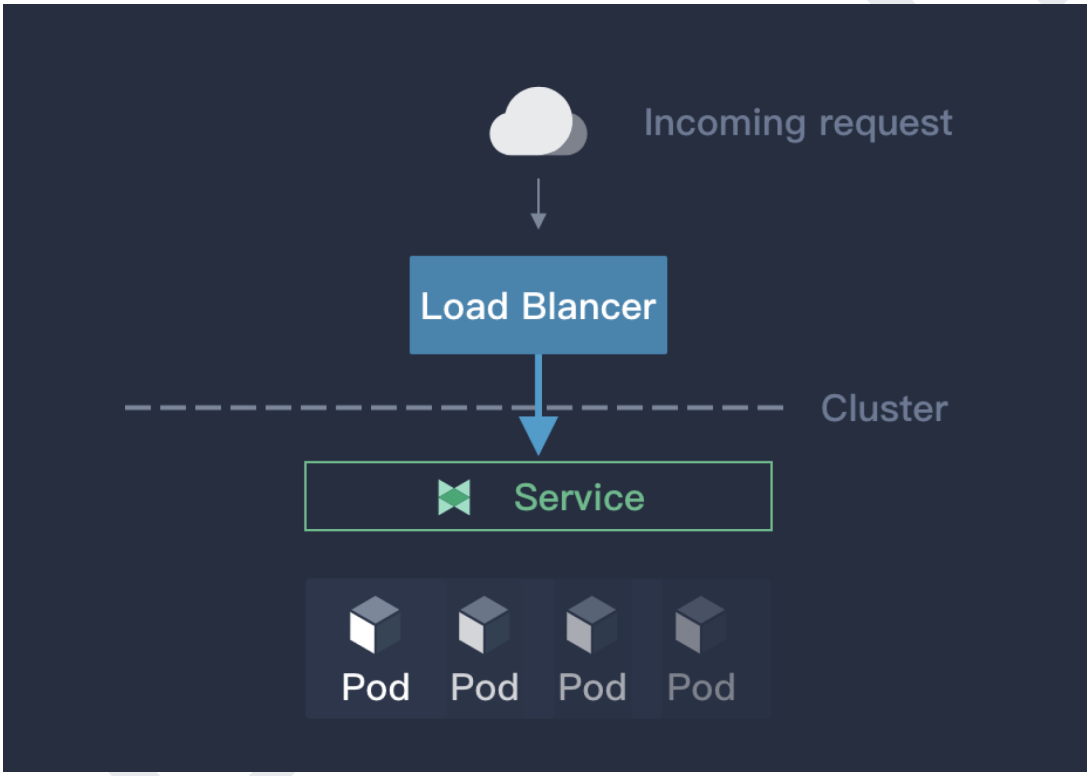
```
apiVersion: v1
kind: Service
metadata:
  name: myapp-service
```

```
spec:
  type: ClusterIP
  ports: # an array
    - targetPort: 80
      port : 80 # port on service object

  selector:
    app: myapp
    type: back-end
```

kubectl create -f clusterip-definition.yml
kubectl get svc

LoadBalancer:



lbservice-def.yml

```
apiVersion: v1
kind: Service
metadata:
  name: myapp-service
spec:
  type: LoadBalancer
  ports: # an array
    - targetPort: 80
      port : 80 # port on service object
```

```
selector:
  app: myapp
  type: front-end
```

kubectl create -f lbservice-def.yml
kubectl get svc

kubectl expose deployment myapp-deployment --name=webapp-service --target-port=8080 --type=NodePort --port=8080 --dry-run=client -o yaml > svc.yml

vi svc.yml
Add nodePort : 30008

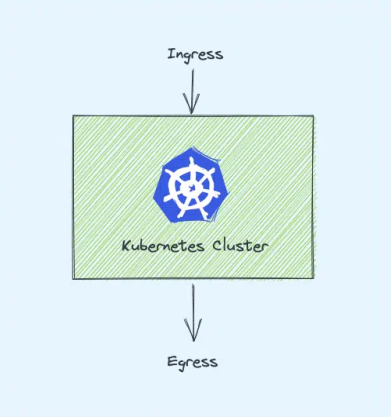
kubectl apply -f svc.yml

Ingress

Kubernetes Ingress is a resource to add rules to route traffic from external sources to the applications running in the kubernetes cluster.

The literal meaning: Ingress refers to the act of entering.

It is the same in Kubernetes world as well. Ingress means the traffic that enters the cluster and egress is the traffic that exits the cluster.

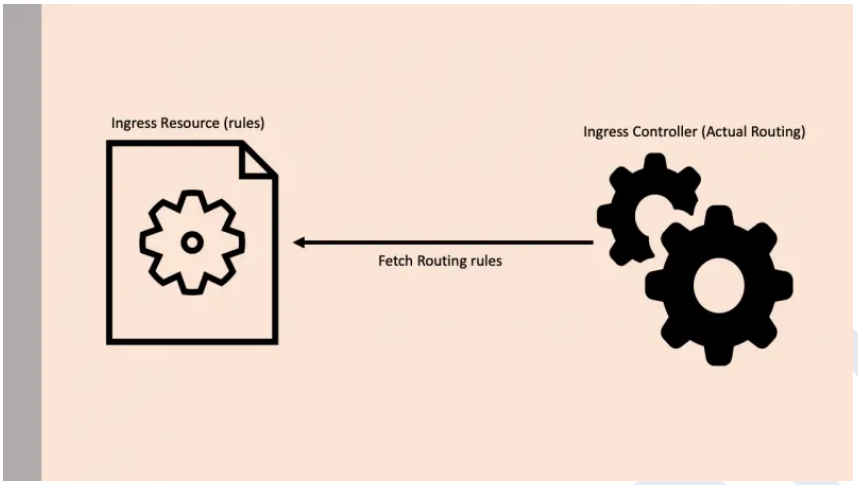


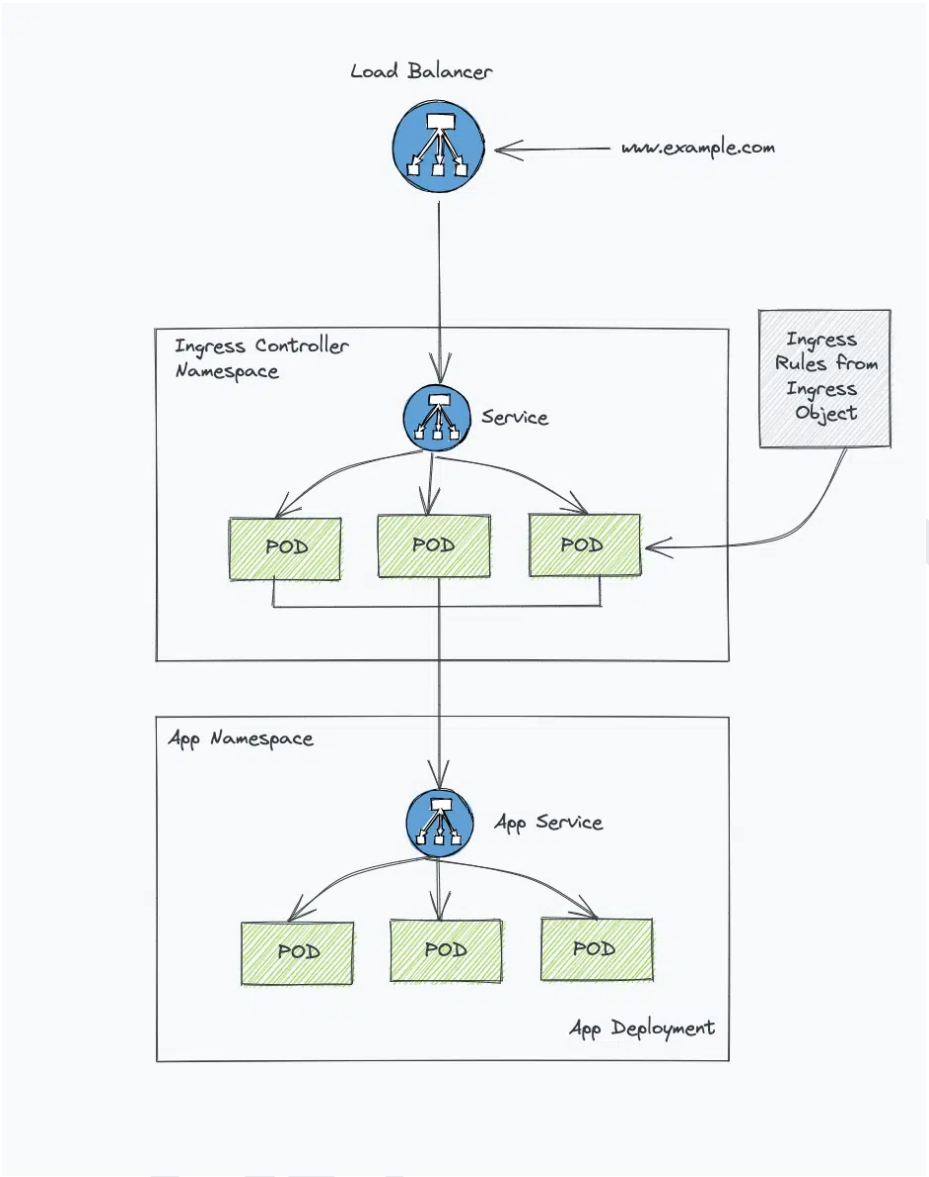
Ingress is a native Kubernetes resource like pods, deployments, etc. Using ingress, you can maintain the DNS routing configurations. The ingress controller does the actual routing by reading the routing rules from ingress objects stored in etcd.

Without Kubernetes ingress, to expose an application to the outside world, you will add a service Type Loadbalancer to the deployments.

You need to be very clear about two key concepts to understand that.

Kubernetes Ingress Resource: Kubernetes ingress resource is responsible for storing DNS routing rules in the cluster.
Kubernetes Ingress Controller: Kubernetes ingress controllers (Nginx/HAProxy etc.) are responsible for routing by accessing the DNS rules applied through ingress resource.





The Kubernetes Ingress resource is a native kubernetes resource where you specify the DNS routing rules. Meaning, you map the external DNS traffic to the internal Kubernetes service endpoints.

It requires an ingress controller for routing the rules specified in the ingress object.

Now, let’s implement the Ingress in Kubernetes.

```
helm upgrade --install ingress-nginx ingress-nginx \
  --repo https://kubernetes.github.io/ingress-nginx \
  --namespace ingress-nginx --create-namespace
```

kubectl get pods ingress-nginx-controller --namespace=ingress-nginx
If you don't have Helm or if you prefer to use a YAML manifest, you can run the following command instead:kubectl get service ingress-nginx-controller --namespace=ingress-nginx

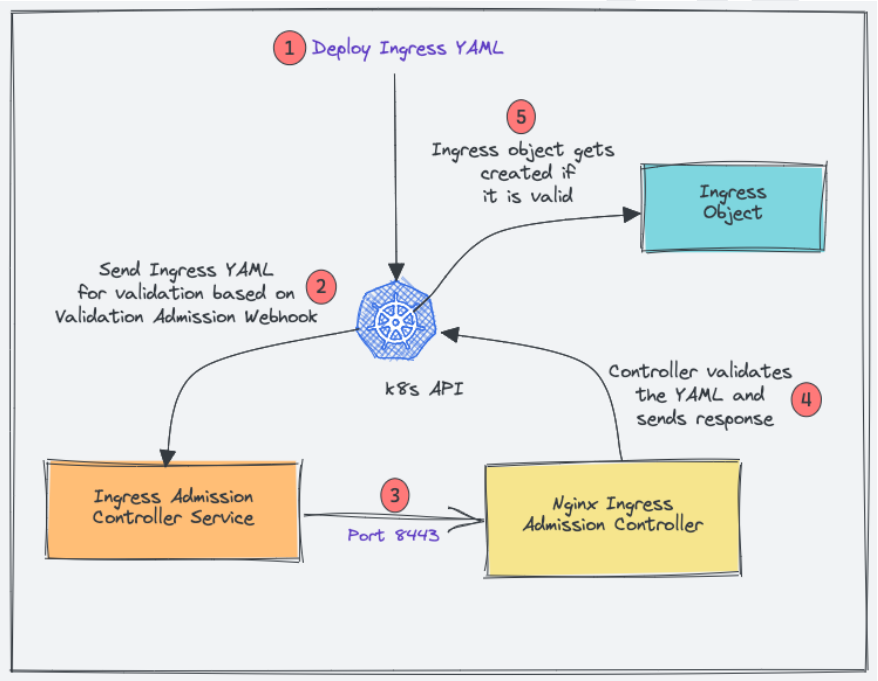
Deploy all ingress controller objects using the following command

kubectl apply -f
https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.1.1/deploy/static/provider/cloud/deploy.yaml

kubectl get pods -n ingress-nginx
kubectl get service ingress-nginx-controller --namespace=ingress-nginx

In this ingress admission controller will also be deployed.

Kubernetes Admission Controller is a small piece of code to validate or update Kubernetes objects before creating them. In this case, it's an admission controller to validate the ingress objects. In this case, the Admission Controller code is part of the Nginx controller which listens on port 8443



If you observe, all the Nginx controller objects are deployed in the ingress-nginx namespace.

To ensure that deployment is working, check the pod status.

kubectl get pods -n ingress-nginx

kubectl get service ingress-nginx-controller --namespace=ingress-nginx
Map a Domain Name To Ingress Loadbalancer IP:
The primary goal of Ingress is to receive external traffic to services running on Kubernetes. Ideally in projects, a DNS would be mapped to the ingress controller Loadbalancer IP.

This can be done via the respective DNS provider with the domain name you own.

I used EKS so instead of Loadnbalacer IP, I have a DNS of network load balancer endpoint which will be a CNAME.

Create a record for that in DNS Management.

Deploy a Demo Application
create a namespace named dev

kubectl create namespace dev

Create a file hello-app.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-app
  namespace: dev
spec:
  selector:
    matchLabels:
      app: hello
  replicas: 2
  template:
    metadata:
      labels:
        app: hello
    spec:
      containers:
      - name: hello
        image: "gcr.io/google-samples/hello-app:2.0"
```

Create a file hello-service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: hello-service
  namespace: dev
  labels:
    app: hello
spec:
  type: ClusterIP
  selector:
```

```
    app: hello
  ports:
  - port: 80
    targetPort: 8080
    protocol: TCP
```

Create a file hello-app-2.yaml

```
kind: Pod
apiVersion: v1
metadata:
  namespace: dev
  name: my-app
  labels:
    app: my-app
spec:
  containers:
    - name: my-app
      image: hashicorp/http-echo
      args:
        - "-text=This is my second application"

---

kind: Service
apiVersion: v1
metadata:
  name: my-service
  namespace: dev
spec:
  selector:
    app: my-app
  ports:
    - port: 5678 # Default port for image
```

kubectl apply -f hello-app-2.yaml

Now let’s create an ingress object to access our hello app and my app using a DNS. An ingress object is nothing but a setup of routing rules.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: test-ingress
```

```
  namespace: dev
spec:
  ingressClassName: nginx
  rules:
  - host: "5382548.ap-south-1.elb.amazonaws.com"
    http:
      paths:
        - pathType: Prefix
          path: "/app1"
          backend:
            service:
              name: hello-service
              port:
                number: 80
        - pathType: Prefix
          path: "/app2"
          backend:
            service:
              name: my-service
              port:
                number: 5678
```

kubectl describe ingress -n dev

Imperative Approach

For creating objects imperatively

kubectl run --image=nginx nginx
kubectl create deployment --image=nginx nginx
kubectl expose deployment nginx --port 80

Updating objects imperatively

kubectl edit deployment nginx
kubectl scale deployment nginx --replicas=5
kubectl set image deployment nginx nginx=nginx:1.18

kubectl create -f nginx.yml
kubectl replace -f nginx.yml
kubectl delete -f nginx.yml

Imperative object configuration files are created automatically when we run imperative commands

Declarative :
Create configuration files and use kubectl apply -f nginx.yml
Can be stored in remote repository.

kubectl replace --force -f nginx.yml
It will delete earlier objects completely and creates newly with updated files

If the object already exists , if we use kubectl create -f nginx.yml, it gives error saying pod already exist.

Before updating objects using replace command, the object must exist already, otherwise error comes.

In declarative approach, kubectl apply command will create objects, if objects do not exist already. For updating also we use apply command.

Lab for Imperative Approach
kubectl run nginx-pod --image=nginx:alpine

kubectl run redis --image=redis:alpine --labels=tier=db

Q. create a service redis-service to expose the redis application within the cluster on port 6379, selector= tier=db

kubectl expose pod redis --name redis-service --port 6379 --target-port 6379

kubectl describe svc redis-service

Q. create a pod called custom-nginx using the nginx image and expose it on container port 8080

kubectl run custom-nginx --image=nginx --port 8080
kubectl describe pod custom-nginx

kubectl create ns dev-ns
kubectl create deployment redis-deploy --image=redis --namespace=dev-ns --dry-run=client -o yaml > redis.yml

vi redis.yml
Edit replicas to 2
kubectl apply -f redis.yml
kubectl get deployments -n dev-ns

Q. create a pod called httpd using the image httpd:alpine in the default namespace. Next create a service of type ClusterIP by the same name (httpd). The target port for the service should be 80.

kubectl run httpd --image=httpd:alpine --port 80 --expose

How kubectl apply command works ?

We have three files for an object.
Local File(nginx.yml), Last Applied Configuration(json), Live Object Configuraton(on kubernetes)

kubectl apply command will compare the live object configuration and takes decision to update the objects or create if they are not existing.

Live object configuration is stored on kubernetes memory.
Last applied configuration is stored as annotations in live object configuration.

Labels and Selectors

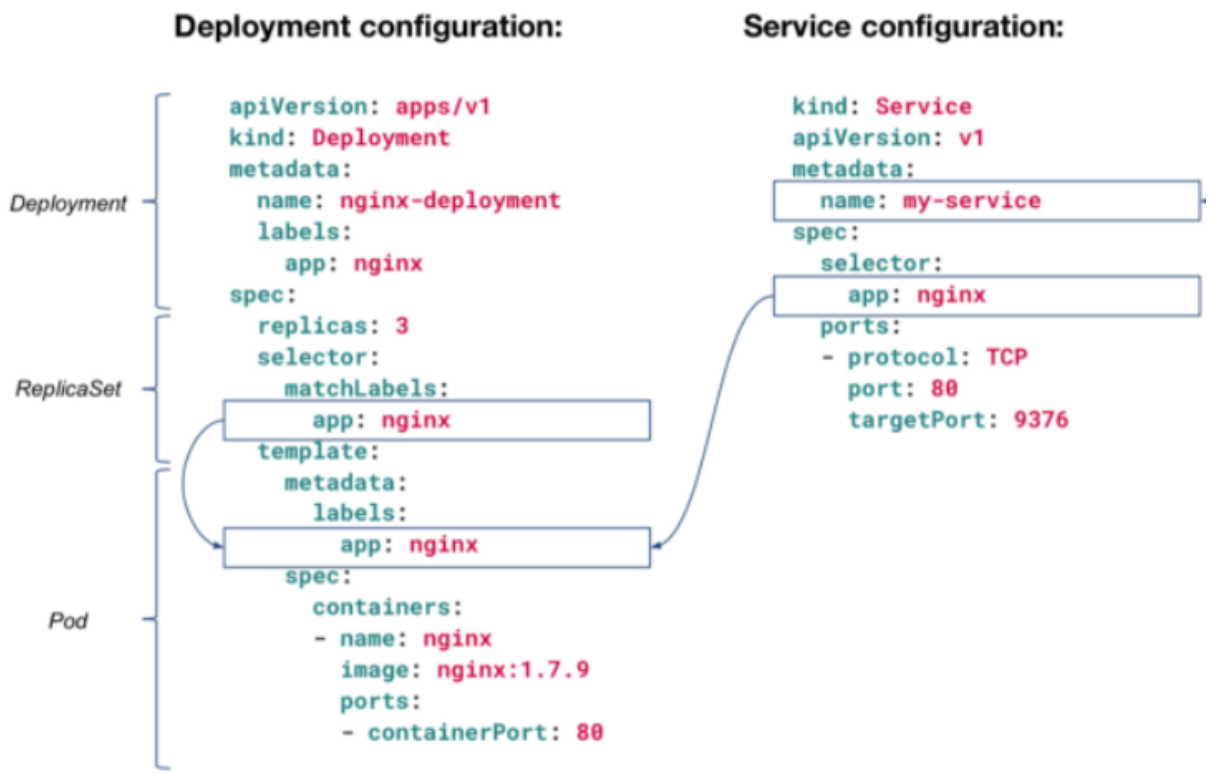
Labels are key/value pairs that are attached to objects, such as pods. Labels are intended to be used to specify identifying attributes of objects that are meaningful and relevant to users, but do not directly imply semantics to the core system. Labels can be used to organise and to select subsets of objects. Labels can be attached to objects at creation time and subsequently added and modified at any time. Each object can have a set of key/value labels defined.

kubectl get pods --selector app=App1

Annotations

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myapp-replicaset
  labels:
    app: myapp
    type: front-end
  annotations:
    buildversion: 1.34
spec:
  template:
    metadata:
     name: myapp-replicaset
     labels:
      app: myapp
```

```
      type: front-end
  spec:
    containers:
    - name: nginx-container
      image: nginx
replicas: 3
selector:
  matchLabels:
    type: front-end
```



kubectl get pods --show-labels

kubectl get pods -l env=dev
kubectl get pods -l env=dev
kubectl get pods -l bu=finance
kubectl get all -l env=prod
kubectl get pods -l env=prod,bu=finance,tier=frontend

Scheduling

Manual scheduling gives the ability to create a pod on a specified node.

Add nodeName in the spec section.

pod-definition.yml

```
apiVersion: v1
kind: Pod
metadata:
 name: myapp-pod
 labels:
  app: myapp
  type: front-end
spec:
  containers:
  - name: nginx-container
    image: nginx
    ports:
      - containerPort: 8080
  nodeName: node02
```

How to schedule a pod which is running without scheduling.

binding-definition.yml

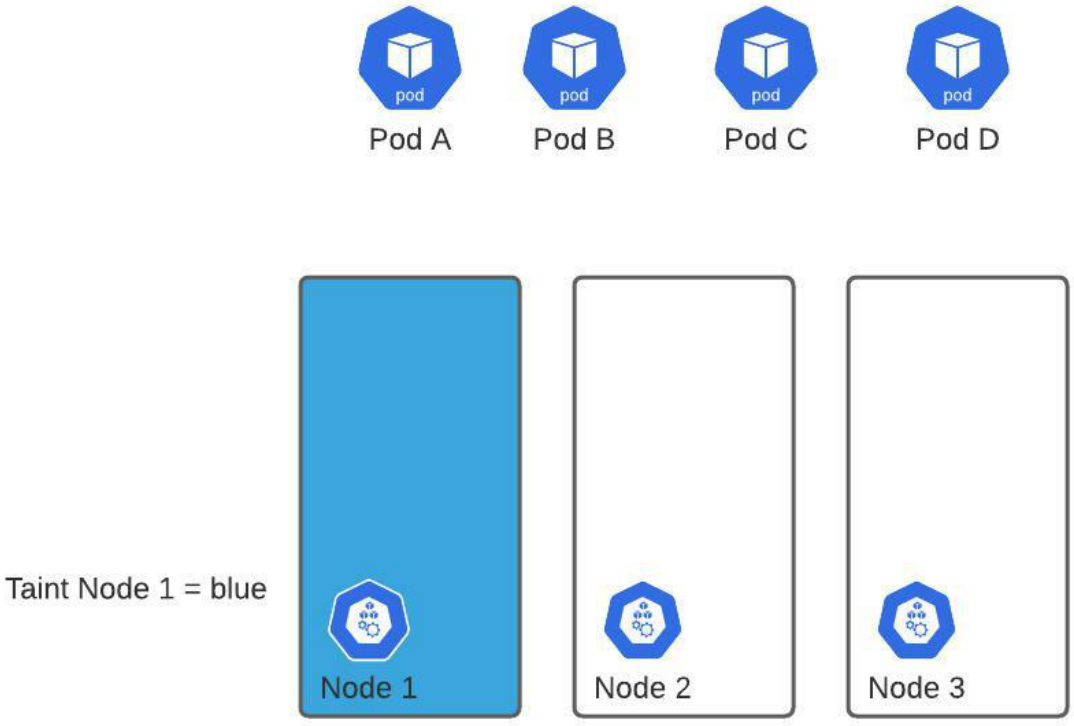
```
apiVersion: v1
kind: Binding
metadata:
  name: nginx
target:
  apiVersion: v1
  kind: Node
  name: node01
```

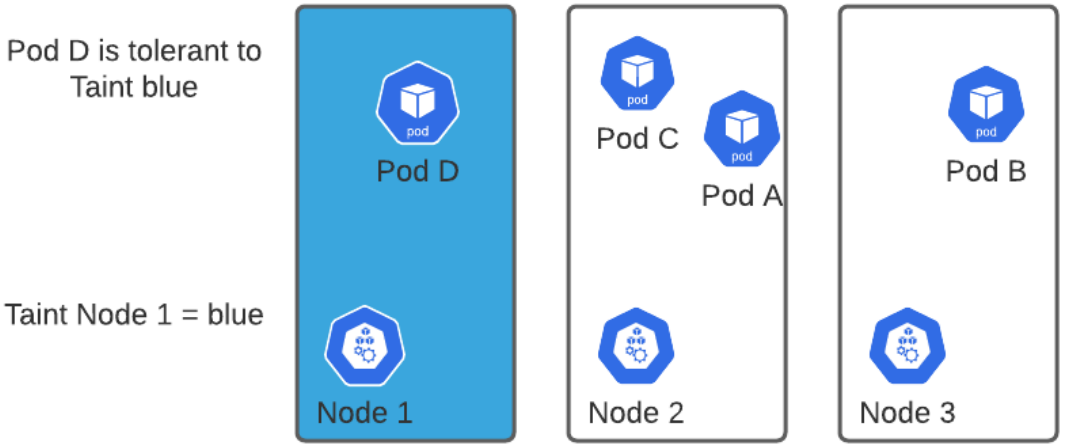
Taints and Tolerations

Node affinity is a property of Pods that attracts them to a set of nodes (either as a preference or a hard requirement). Taints are the opposite -- they allow a node to repel a set of pods.

Tolerations are applied to pods. Tolerations allow the scheduler to schedule pods with matching taints. Tolerations allow scheduling but don't guarantee scheduling: the scheduler also evaluates other parameters as part of its function.

Taints and tolerations work together to ensure that pods are not scheduled onto inappropriate nodes. One or more taints are applied to a node; this marks that the node should not accept any pods that do not tolerate the taints
Taints are a property of nodes that push pods away if they are not tolerate to node taint.





kubectl taint nodes node-name key=value:taint-effect
Taint-effect : NoSchedule, PreferNoSchedule, NoExecute
kubectl taint nodes node1 app=blue:NoSchedule
To get the taints of nodes
kubectl get nodes -o custom-columns=NAME:.metadata.name,TAINTS:.spec.taints
Toleration to Pods

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  labels:
spec:
  containers :
  - name: nginx-container
    image: nginx
  tolerations:
  - key : "app"
    operator: "Equal"
    value: "blue"
    effect: "NoSchedule"
```

kubectl describe node kubemaster | grep Taint

kubectl describe node node01 | grep -i taint
kubectl taint node node01 spray=mortein:NoSchedule

kubectl describe node node01 | grep -i taint
kubectl run mosquito --image=nginx --restart=Never
What is the state of the pod
Pending
Why do you think the pod is in pending state ?
Because pod can not tolerate tain mortein
kubectl describe pod mosquito
kubectl run bee --image-nginx --restart=Never --dry-run -o yaml > bee.yml
Add tolerations in bee.yml
Copy from kubectl explain pod --recursive | less
kubectl explain pod --recursive | less
kubectl explain pod --recursive | grep -A5 tolerations

bee.yml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  labels:
spec:
  containers :
  - name: nginx-container
    image: nginx
  tolerations:
  - effect : NoSchedule
    key : spray
    operator: Equal
    value: mortein
```

kubectl get pods

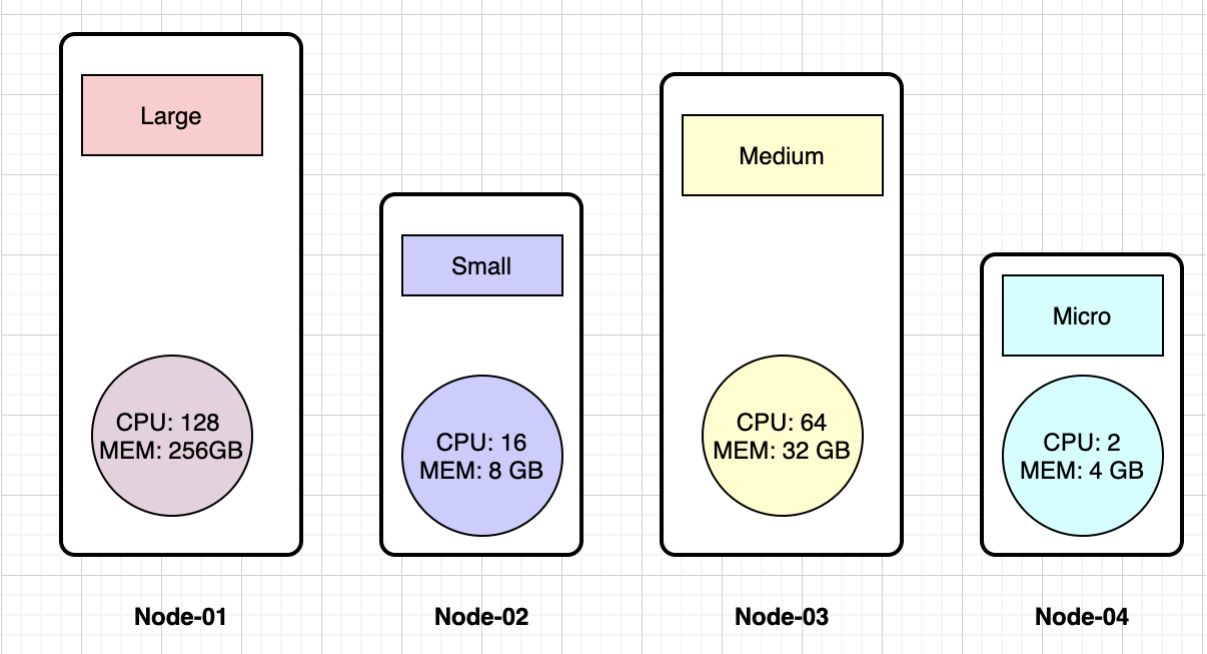
Now remove the taint on the master

kubectl describe node master | grep -i taint
kubectl taint node master node-role.kubernetes.io/master:NoSchedule-

We us minus (-) symbol to remove the taint

What is the state of the pod mosquito now ?

Node Selectors



Label Nodes

kubectl label nodes <node-name> <label-key>=<label-value>
kubectl label nodes Node-1 size=Large

pod-definition.yml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  labels:
spec:
  containers :
  - name: nginx-container
    image: nginx
  nodeSelector:
    size: Large
```

kubectl apply -f pod-definition.yml

Node Affinity

What if our requirement is complex like what if we desire to place a pod on either Large or Medium, Not Small nodes ?
It means to place a pod on any node that is not small. How to achieve that ?
You cannot achieve this using node selectors.
To achieve this node affinity concept is introduced.
The primary purpose of node affinity feature is to ensure that pods are hosted on particular nodes.
pod-definition.yml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
spec:
  containers :
  - name: data-processor
    image: nginx
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: size
            operator: In
            values:
            - Large
```

For Large or Medium

```yaml
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: size
            operator: In
            values:
            - Large
            - Medium
```

For not in Small node

```
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: size
          operator: NotIn
          values:
          - Small
```

Exists operator will simply check if the label **'size'** exists on the nodes, and you don't need the values section for that, because it does not check the values.

```
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: size
          operator: Exists
```

Node Affnity Types:

1. requiredDuringSchedulingIgnoredDuringExecution
2. preferredDuringSchedulingIgnoredDuringExecution

	During Scheduling	During Execution
Type 1	Required	Ignored
Type 2	Preferred	Ignored

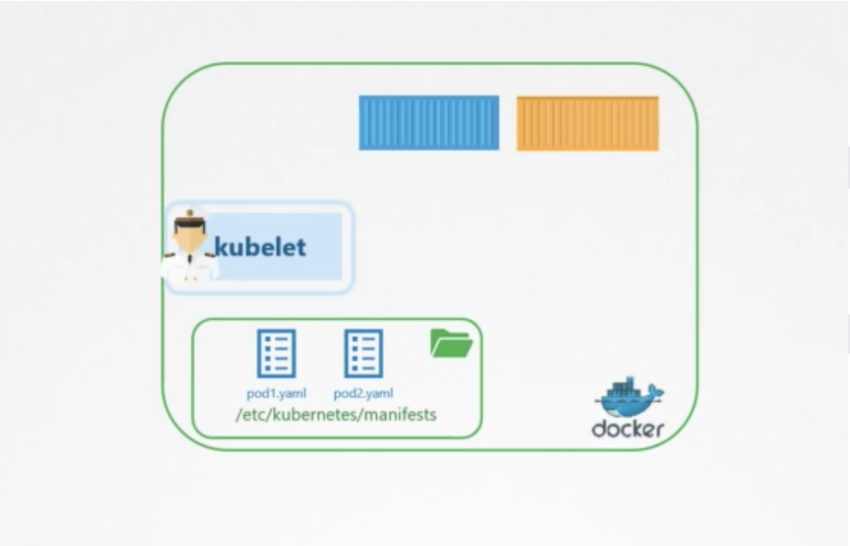
kubectl get nodes node1 --show-labels
kubectl label nodes node1 color=blue
kubectl get nodes node1 --show-labels

Static Pods

Static pods are pods created and managed by kubelet daemon on a specific node without API server observing them. If the static pod crashes, kubelet restarts them. Control plane is not involved in lifecycle of static pod. Kubelet also tries to create a mirror pod on the kubernetes api server for each static pod so that the static pods are visible i.e., when you do kubectl get pod for example, the mirror object of static pod is also listed.

The path of the directory holding the static pod definition file is

/etc/kubernetes/manifests



docker ps command to check pods running in the node

kubectl get pods -n kube-system

Q. How many static pods exists in this cluster in all namespaces
kubectl get pods --all-namespaces
Note that name of the static pod is appended by the node at the end (-master, -node)
kubectl get pods --all-namespaces | grep “\-master”

Q. what is the path of the directory holding the static pod definition file?
ps -ef | grep kubelet
grep -i static /var/lib/kubelet/config.yaml
cd /etc/kubernetes/manifests

cd /etc/kubernetes/manifests
ls

Logging and Monitoring

Monitor Cluster Components
Metrics Server
minikube addons enable metrics-server

For others
git clone https://github.com/*kubernetes-incubator*/*metrics*-server.git
kubectl apply -f deploy/1.8+/
kubectl apply -f
https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml

kubectl top node

kubectl top pod

watch “kubectl top node”

Application Logs

```
kubectl create -f event-simulator.yaml
kubectl logs -f event-simulator-pod event-simulator
kubectl logs <pod-name>

kubectl get pods
kubectl logs webapp-1 -c
c flag to show containers
kubectl logs webapp-1 -c <container-name>
kubectl logs webapp-1 -c db
```

Environment Variables

When you create a Pod, you can set environment variables for the containers that run in the Pod. To set environment variables, include the env or envFrom field in the configuration file.

pod-definition.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp-color
spec:
```

```
  containers:
  - name: simple-webapp-color
    image: simple-webapp-color
    ports:
      - containerPort: 8080
    env:
      - name: APP_COLOR
        value: pink
```

ENV value types:
1. Plain key value

```
env:
  - name: APP_COLOR
    value: pink
```

2. ConfigMap

```
env:
  - name: APP_COLOR
    valueFrom:
      configMapKeyRef:
```

3. Secrets

```
env:
  - name: APP_COLOR
    valueFrom:
      secretKeyRef:
```

ConfigMap
APP_COLOR: blue
APP_MODE: prod

Imperative:
kubectl create configmap app-config --from-literal=APP_COLOR=blue \
--from-literal=APP_MODE=prod

kubectl create configmap <config-name> --from-file=<path-to-file>
kubectl create configmap \
app-config --from-file=app_confi.properties

Declarative:
config-map.yaml

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  APP_COLOR: blue
  APP_MODE: prod
```

kubectl create -f config-map.yaml

kubectl get configmaps
kubectl describe configmaps

ConfigMap in Pods

Pod-definition.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp-color
spec:
  containers:
  - name: simple-webapp-color
    image: simple-webapp-color
    ports:
      - containerPort: 8080
    envFrom:
      - configMapRef:
          name: app-config
```

kubectl create -f pod-definition.yaml
Single View

```yaml
env:
  - name: APP_COLOR
    valueFrom:
      configMapKeyRef:
        name: app-config
```

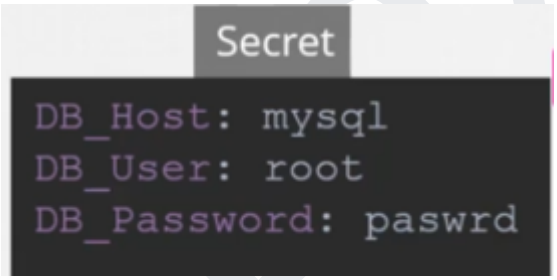
```
        key: APP_COLOR
```

Volumes

```
volumes:
- name : app-config-volume
  configMap:
    name: app-config
```

kubectl get pods
kubectl describe pod webapp-color | grep -i environment
kubectl get cm
Identify the database host from the configmap ‘db-config’
kubectl describe cm db-config
kubectl create cm webapp-config-map --from-literal=APP_COLOR=darkblue
kubectl explain pods --recursive | grep envFrom -A3

Secrets



Imperative:
kubectl create secret generic <secret-name> --from-literal=<key>=<value>
kubectl create secret generic \
App-secret --from-literal=DB_Host=mysql \
--from-literal=DB_User=root \
--from-literal=DB_Password=paswd

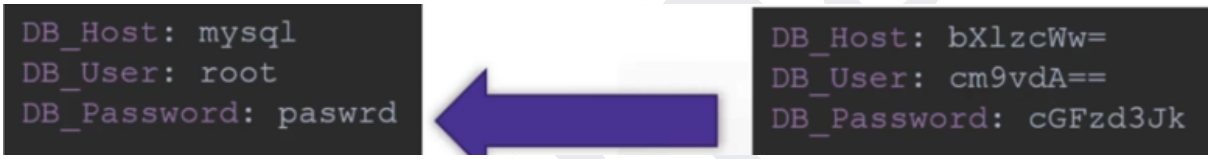
Declarative:
secret-data.yaml

```
apiVersion: v1
```

```yaml
kind: Secret
metadata:
  name: app-secret
data:
  DB_Host: mysql
  DB_User: root
  DB_Password: paswrd
```

kubectl apply -f secret-data.yaml
Encode secrets:

echo -n 'mysql' | base64
kubectl get secrets
kubectl describe secrets
kubectl get secret app-secret -o yaml



echo -n 'bX1zcWw' | base64 --decode

Secrets in Pods

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
  labels:
    app: myapp
    type: front-end
    env: dev
spec:
  template:
    metadata:
     name: myapp-deployment
     labels:
```

```
      app: myapp
      type: front-end
    spec:
      containers:
      - name: nginx-container
        image: nginx
        envFrom:
          - secretRef:
              name: app-secret

  replicas: 3
  selector:
    matchLabels:
      type: front-end
```

kubectl create -f pod-definition.yaml

Env

```
    envFrom:
      - secretRef:
          name: app-secret
```

Single View

```
env:
  - name: DB_Password
    valueFrom:
      secretKeyRef:
        name: app-secret
        key: DB_Password
```

Volumes

```
volumes:
- name : app-secret-volume
  secret:
    secretName: app-secret
```

```
kubectl get secrets
kubectl describe secrets app-secrets
```

```
kubectl get pods, svc
```

Pulling Images from Private Registry

https://kubernetes.io/docs/tasks/configure-pod-container/pull-image-private-registry/

Pull an Image from a Private Registry

kubectl create secret docker-registry my-secret --docker-server=laxmandevops.azurecr.io
--docker-username=laxmandevops
--docker-password=cSGuVaCekbZ5SxsVaxOKH2fcmfFx9+aDibESfCtEiv+ACRAJ4AAZ
--dry-run=client -o yaml > acr-secret.yaml

Kubernetes Multi Container Pods

Why Use Multi Container Pods ?
Well there are many good reasons why to use them rather than not to use them; here are some of them:

- ☐ The primary purpose of a multi-container Pod is to support co-located, co-managed helper processes for a primary application.
- ☐ With the same network namespace, shared volumes, and the same IPC namespace it possible for these containers to efficiently communicate, ensuring data locality.
- ☐ They enable you to manage several tightly coupled application containers as a single unit.
- ☐ Another reason is that all containers have the same lifecycle which should run on the same node.


```yaml
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp-color
spec:
  containers:
  - name: simple-webapp-color
    image: simple-webapp-color
```

```
    ports:
      - containerPort: 8080
  - name: log-agent
    image: log-agent
    Env:
    Ports:
  -
```

kubectl run yellow --image=busybox --restart=Never --dry-run -o yaml > pod.yaml

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

  restartPolicy: Never

  volumes:
  - name: shared-data
    emptyDir: {}

  containers:

  - name: nginx-container
    image: nginx
    volumeMounts:
    - name: shared-data
      mountPath: /usr/share/nginx/html

  - name: ubuntu-container
    image: ubuntu
    volumeMounts:
    - name: shared-data
      mountPath: /pod-data
    command: ["/bin/sh"]
    args: ["-c", "echo Hello, World!!! > /pod-data/index.html"]
```

kubectl apply -f pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: multi-pod
spec:
  volumes:
  - name: shared-data
    emptyDir: {}
  containers:
  - name: nginx-container
    image: nginx
    volumeMounts:
    - name: shared-data
      mountPath: /usr/share/nginx/html
  - name: ubuntu-container
    image: ubuntu
    volumeMounts:
    - name: shared-data
      mountPath: /pod-data
    command: ["/bin/sh"]
    args: ["-c", "echo Hello, World!!! > /pod-data/index.html"]
```

In the above YAML file, you will see that we have deployed a container based on the Nginx image, as our web server. The second container is named ubuntu-container, is deployed based on the Ubuntu image, and writes the text “Hello, World!!!” to the index.html file served up by the first container.

InitContainers

https://kubernetes.io/docs/concepts/workloads/pods/init-containers/

In a multi-container pod, each container is expected to run a process that stays alive as long as the POD's lifecycle. For example in the multi-container pod that we talked about earlier that has a web application and logging agent, both the containers are expected to stay alive at all times. The process running in the log agent container is expected to stay alive as long as the web application is running. If any of them fails, the POD restarts.

But at times you may want to run a process that runs to completion in a container. For example a process that pulls a code or binary from a repository that will be used by the main web application. That is a task that will be run only one time when the pod is first created. Or a process that waits for an external service or database to be up before the actual application starts. That's where **initContainers** comes in.

An **initContainer** is configured in a pod like all other containers, except that it is specified inside a initContainers section, like this:

```
apiVersion: v1
kind: Pod
metadata:
 name: myapp-pod
 labels:
   app: myapp
spec:
 containers:
 - name: myapp-container
   image: busybox:1.28
   command: ['sh', '-c', 'echo The app is running! && sleep 3600']
 initContainers:
 - name: init-myservice
   image: busybox
   command: ['sh', '-c', 'git clone
<some-repository-that-will-be-used-by-application> ; done;']
```

When a POD is first created the initContainer is run, and the process in the initContainer must run to a completion before the real container hosting the application starts.

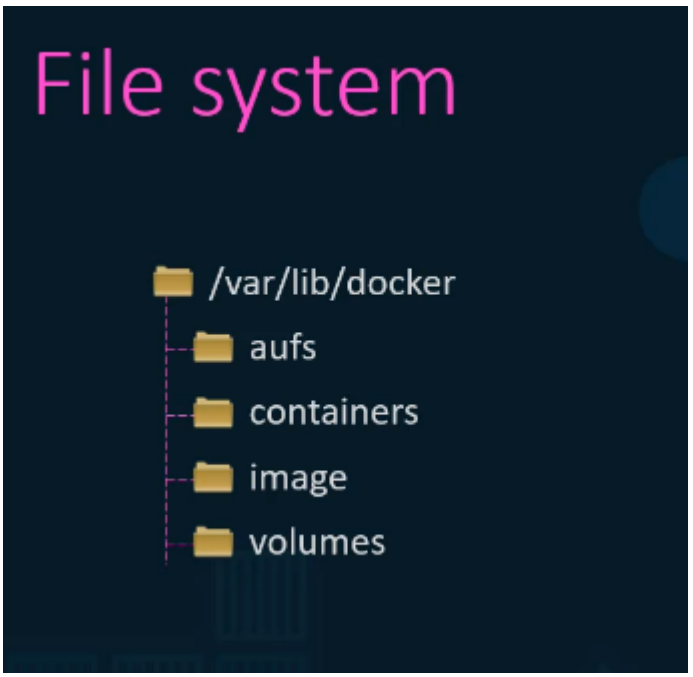
You can configure multiple such initContainers as well, like how we did for multi-pod containers. In that case each init container is run **one at a time in sequential order**.

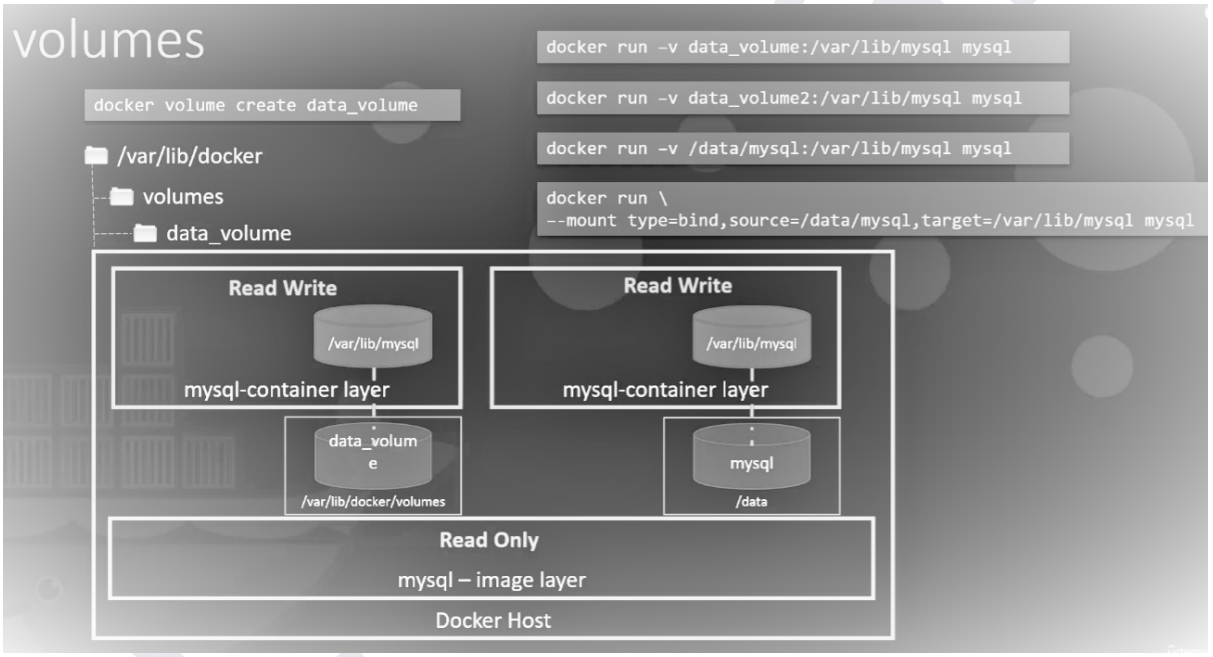
If any of the initContainers fail to complete, Kubernetes restarts the Pod repeatedly until the Init Container succeeds.

```
apiVersion: v1
kind: Pod
metadata:
 name: myapp-pod
 labels:
   app: myapp
spec:
 containers:
 - name: myapp-container
   image: busybox:1.28
   command: ['sh', '-c', 'echo The app is running! && sleep 3600']
 initContainers:
 - name: init-myservice
   image: busybox:1.28
   command: ['sh', '-c', 'until nslookup myservice; do echo waiting for myservice; sleep 2; done;']
 - name: init-mydb
   image: busybox:1.28
   command: ['sh', '-c', 'until nslookup mydb; do echo waiting for mydb; sleep 2; done;']
```

Identify the pod that has initContainer configured

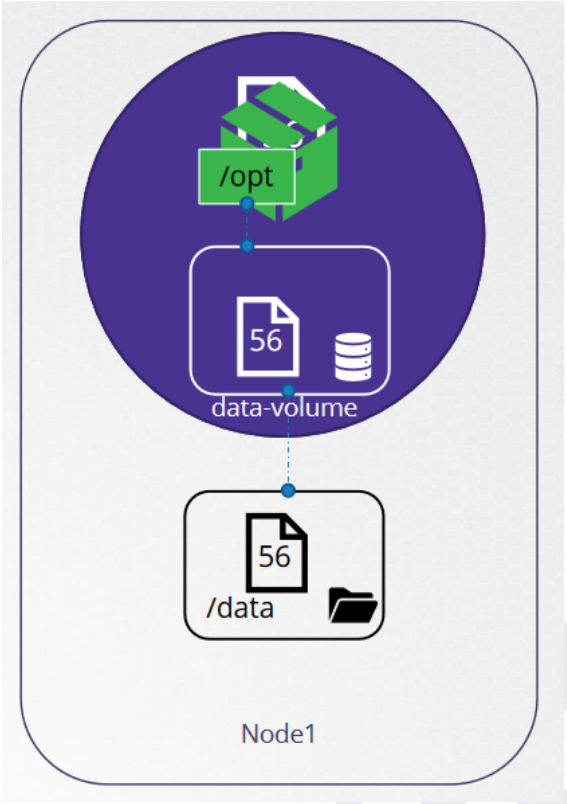
Storage





Storage Drivers:
AUFS
ZFS

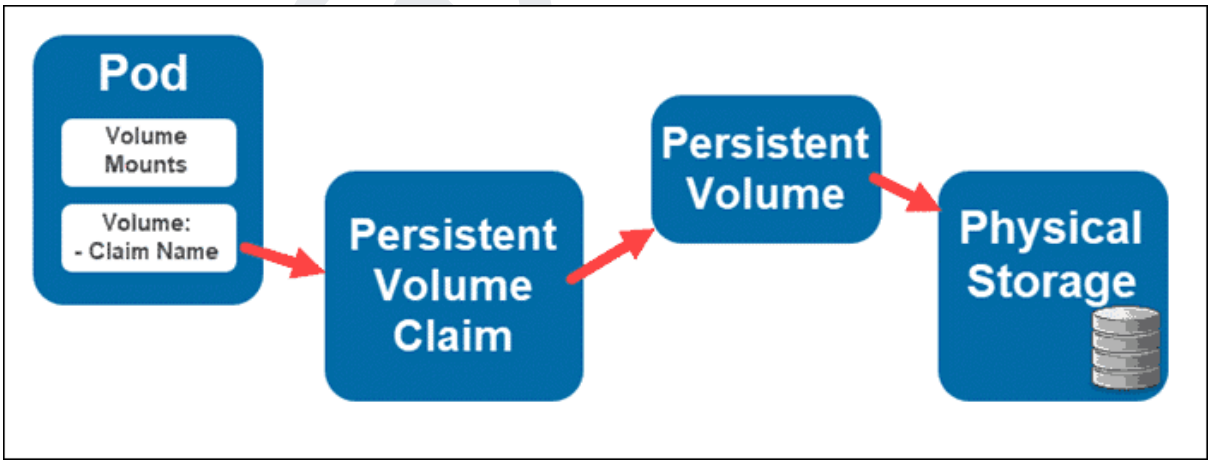
Volumes & Mounts




```yaml
apiVersion: v1
kind: Pod
metadata:
  name: random-number-generator
spec:
  containers:
  - image: alpine
    name: alpine
    command: ["/bin/sh","-c"]
    args: ["shuf -i 0-100 -n 1 >> /opt/number.out;"]
    volumeMounts:
    - mountPath: /opt
      name: data-volume
  volumes:
  - name: data-volume
    hostPath:
      path: /data
      type: Directory
```


```
volumes:
  - name: data-volume
    awsElasticBlockStore:
      volumeID: <volume-id>
      fsType: ext4
```

Persistent Volumes:



pv-definition.yaml

```
apiVersion: v1
kind: PersistentVolume
metadata:
```

```
  name: p-vol1
spec:
  accessModes:
    - ReadWriteOnce
  capacity:
    storage: 5Gi
  awsElasticBlockStore:
    volumeID: <volume-id>
    fsType: ext4
```

pvc-definitino.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myclaim
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 500Mi
```

pod-definition.yaml

```
apiVersion: v1
kind: Pod
metadata:
 name: mypod
spec:
 containers:
  - name: myfrontend
    image: nginx
    volumeMounts:
    - mountPath: "/var/www/html"
```

```
    name: mypd
volumes:
  - name: mypd
    persistentVolumeClaim:
      claimName: myclaim
```

Storage Class:

https://kubernetes.io/docs/concepts/storage/storage-classes/

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
 name: aws-storage
provisioner: kubernetes.io/aws-ebs
```

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myclaim
spec:
  accessModes:
  - ReadWriteOnce
  storageClassName: aws-storage
  resources:
    requests:
      storage: 500Mi
```

Creating Persistent Volume in a Kubernetes cluster using NFS

Installing and Configuring an NFS Server on Ubuntu 18.04

apt update
apt install nfs-kernel-server
mkdir -p /mnt/nfs_share
chown -R nobody:nogroup /mnt/nfs_share/
vim /etc/exports
-insert this content to /etc/exports
/mnt/nfs_share *(rw,sync,no_subtree_check,insecure)
exportfs -a
- to check exports
exportfs -v or showmount -e
systemctl restart nfs-kernel-server

systemctl status nfs-server

Installing nfs-csi drivers in kubernetes
https://github.com/kubernetes-csi/csi-driver-nfs/blob/master/docs/install-csi-driver-v4.1.0.md

curl -skSL
https://raw.githubusercontent.com/kubernetes-csi/csi-driver-nfs/v4.0.0/deploy/install-driver.sh
| bash -s v4.0.0 --
Check pod status

```
kubectl -n kube-system get pod -o wide -l app=csi-nfs-controller
kubectl -n kube-system get pod -o wide -l app=csi-nfs-node
```

pv.yaml

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfs-pv
spec:
  capacity:
    storage: 1Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: nfs
  mountOptions:
    - hard
```

```
    - nfsvers=4.1
  nfs:
    path: /mnt/nfs_share
    server: 13.127.248.225
```

pvc.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nfs-pvc
spec:
  storageClassName: nfs
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
```

pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pv-pod
  labels:
    app: myapp
    type: front-end
spec:
  volumes:
    - name: nginx-pv-storage
      persistentVolumeClaim:
        claimName: nfs-pvc
  containers:
    - name: nginx
      image: nginx
      ports:
        - containerPort: 80
          name: "nginx-server"
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: nginx-pv-storage
```

svc.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: myapp-service
spec:
  type: LoadBalancer
  ports: # an array
    - targetPort: 80
      port : 80 # port on service object
  selector:
    app: myapp
    type: front-end
```

Now create your own index.html in this path /mnt/nfs_share of your nfs server and access the load balancer url .

https://ripon-banik.medium.com/efs-as-storageclass-for-eks-k8s-cluster-604bdcdba8ac

Dynamic Provisioning

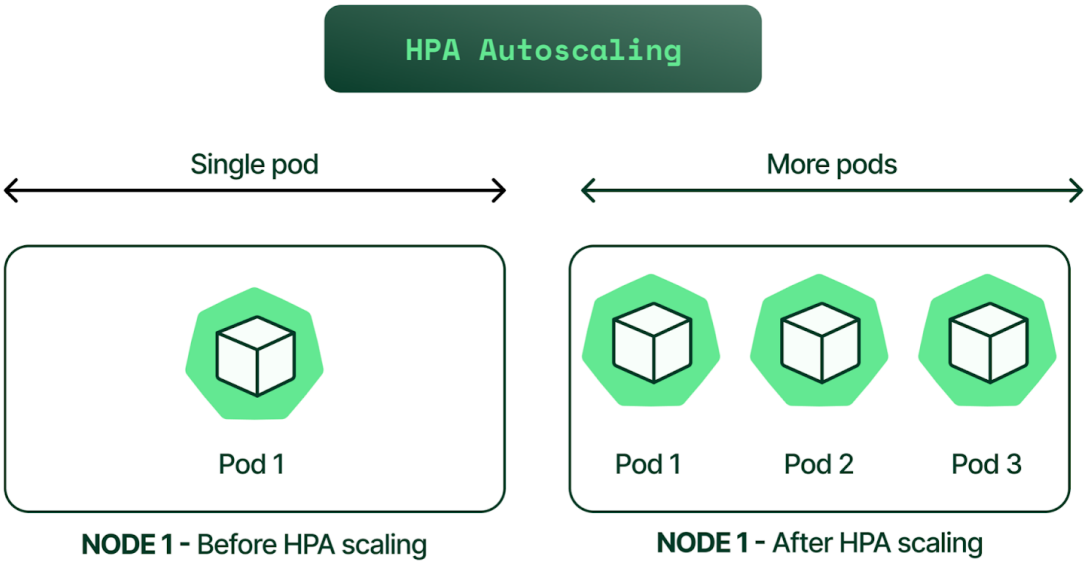
https://aws.amazon.com/premiumsupport/knowledge-center/eks-persistent-storage/
https://repost.aws/knowledge-center/eks-persistent-storage

https://docs.aws.amazon.com/eks/latest/userguide/ebs-csi.html

HorizontalPodAutoscaler

A HorizontalPodAutoscaler (HPA for short) automatically updates a workload resource (such as a Deployment or StatefulSet), with the aim of automatically scaling the workload to match demand.

Horizontal scaling means that the response to increased load is to deploy more Pods. This is different from vertical scaling, which for Kubernetes would mean assigning more resources (for example: memory or CPU) to the Pods that are already running for the workload.



If the load decreases, and the number of Pods is above the configured minimum, the HorizontalPodAutoscaler instructs the workload resource (the Deployment, StatefulSet, or other similar resource) to scale back down.

Install metrics server

kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml

php-apache.yaml

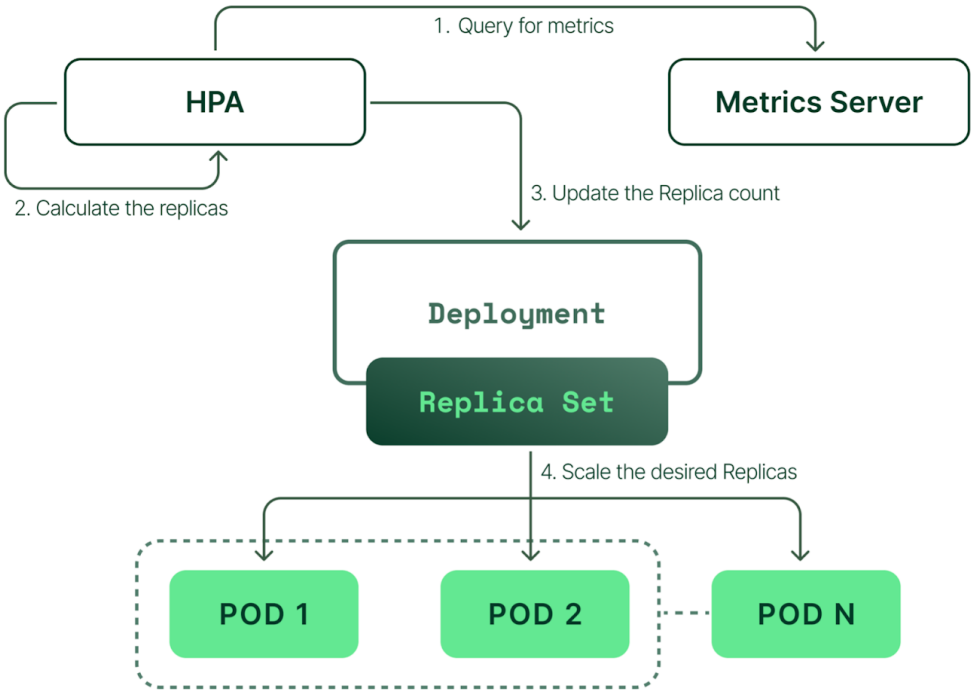
```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: php-apache
spec:
 selector:
   matchLabels:
     run: php-apache
 replicas: 1
 template:
   metadata:
     labels:
       run: php-apache
   spec:
     containers:
     - name: php-apache
       image: k8s.gcr.io/hpa-example
```

```
      ports:
      - containerPort: 80
      resources:
        limits:
          cpu: 500m
        requests:
          cpu: 200m

---
apiVersion: v1
kind: Service
metadata:
 name: php-apache
 labels:
   run: php-apache
spec:
 ports:
 - port: 80
 selector:
   run: php-apache
```

kubectl apply -f application/php-apache.yaml

Roughly speaking, the HPA controller will increase and decrease the number of replicas (by updating the Deployment) to maintain an average CPU utilization across all Pods of 50%.




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

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

```
kubectl get hpa
```

Increase the load

Next, see how the autoscaler reacts to increased load. To do this, you'll start a different Pod to act as a client. The container within the client Pod runs in an infinite loop, sending queries to the php-apache service.

 # Run this in a separate terminal

 # so that the load generation continues and you can carry on with the rest of the steps

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

```
kubectl get hpa php-apache --watch

kubectl get deployment php-apache
```

Stop generating load

To finish the example, stop sending the load.

In the terminal where you created the Pod that runs a `busybox` image, terminate the load generation by typing `<Ctrl> + C`.

Then verify the result state (after a minute or so):

```
# type Ctrl+C to end the watch when you're ready

kubectl get hpa php-apache --watch

kubectl get deployment php-apache
```

Realtime Example

```yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  labels:
  name: example-deployment
  namespace: dev-dub
spec:
  maxReplicas: 3
  minReplicas: 1
  scaleTargetRef:
    apiVersion: apps/v1beta1
    kind: Deployment
    name: example-deployment
```

```
  targetCPUUtilizationPercentage: 90
```

Kubernetes Cluster Using Kubeadm

Following are the prerequisites for Kubeadm Kubernetes cluster setup.

1. Minimum two Ubuntu nodes [One master and one worker node]. You can have more worker nodes as per your requirement.
2. The master node should have a minimum of 2 vCPU and 2GB RAM. (t2.medium)
3. For the worker nodes, a minimum of 1vCPU and 2 GB RAM is recommended.(t2.small)

Following are the high-level steps involved in setting up a Kubernetes cluster using kubeadm.

1. Install container runtime on all nodes- We will be using Docker.
2. Install Kubeadm, Kubelet, and kubectl on all the nodes.
3. Initiate Kubeadm control plane configuration on the master node.
4. Save the node join command with the token.
5. Install the Calico network plugin.
6. Join worker node to the master node (control plane) using the join command.
7. Validate all cluster components and nodes.
8. Install Kubernetes Metrics Server
9. Deploy a sample app and validate the app

Control Plane: Execute the following commands in control plane(master)

For kubeadm to work properly, you need to disable swap on all the nodes using the following command.
sudo swapoff -a
sudo sed -i '/ swap / s/^\(.*\)$/#\1/g' /etc/fstab
curl https://get.docker.com/ | bash
sudo apt-get update && sudo apt-get install -y apt-transport-https curl
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -
cat <<EOF | sudo tee /etc/apt/sources.list.d/kubernetes.list
deb https://apt.kubernetes.io/ kubernetes-xenial main
EOF
sudo apt-get update
sudo apt-get install -y kubelet kubeadm kubectl
sudo apt-mark hold kubelet kubeadm kubectl

---

kubeadm init --pod-network-cidr 192.168.0.0/16

Or use
sudo kubeadm init --pod-network-cidr=192.168.0.0/16 --control-plane-endpoint "PUBLIC_IP:PORT"
PORT=6443
sudo kubeadm init --pod-network-cidr=192.168.0.0/16 --control-plane-endpoint "13.233.74.52:6443"

For making the cluster accessible outside the network.

if you get error
run the following commands
rm /etc/containerd/config.toml
systemctl restart containerd

kubeadm init --pod-network-cidr 192.168.0.0/16

Copy the token somewhere (This is the token which we use to execute in worker nodes)

Execute the following commands in worker nodes:

For kubeadm to work properly, you need to disable swap on all the nodes using the following command.
sudo swapoff -a
sudo sed -i '/ swap / s/^\(.*\)$/#\1/g' /etc/fstab
The fstab entry will make sure the swap if off on system reboots.

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

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

kubeadm join 10.1.1.200:6443 --token ynogss.v28um8uq3pbza9mo \
    --discovery-token-ca-cert-hash
sha256:e5c238a5f964a05cdf2e09adfe8d3458eaabb6925327fa76849ae8463d0ea84f

if you get error if you get error
run the following commands
rm /etc/containerd/config.toml
systemctl restart containerd

kubeadm join 10.1.1.200:6443 --token ynogss.v28um8uq3pbza9mo \
    --discovery-token-ca-cert-hash
sha256:e5c238a5f964a05cdf2e09adfe8d3458eaabb6925327fa76849ae8463d0ea84f

—----------

Execute the following commands in control plane:

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

root@ip-10-1-1-81:~# kubectl get nodes
NAME STATUS ROLES AGE VERSION
ip-10-1-1-81 NotReady master 2m16s v1.18.3

Execute the following command to install the calico network plugin on the cluster.

https://docs.tigera.io/calico/3.25/getting-started/kubernetes/self-managed-onprem/onpremises

curl https://raw.githubusercontent.com/projectcalico/calico/v3.25.0/manifests/calico.yaml -O
kubectl apply -f calico.yaml

root@ip-10-1-1-81:~# kubectl get nodes

NAME STATUS ROLES AGE VERSION
ip-10-1-1-81 Ready master 5m47s v1.18.3

AGAIN ON THE MASTER NODE:
==========

root@ip-10-1-1-81:~# kubectl get nodes
NAME STATUS ROLES AGE VERSION
ip-10-1-1-81 Ready master 7m50s v1.18.3
ip-10-1-1-99 Ready <none> 2m14s v1.18.3
root@ip-10-1-1-81:~# kubectl label node ip-10-1-1-99 node-role.kubernetes.io/worker=worker
node/ip-10-1-1-99 labeled
root@ip-10-1-1-81:~# kubectl get nodes
NAME STATUS ROLES AGE VERSION
ip-10-1-1-81 Ready master 8m19s v1.18.3
ip-10-1-1-99 Ready worker 2m43s v1.18.3

To install the metrics server

kubectl apply -f
https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml

Check the status of metrics server deployment using following command
kubectl -n kube-system get pods

Or

kubectl -n kube-system get deploy

If the pods are not running and reason(which can be found in the description of the pod) is

Readiness probe failed: HTTP probe failed with statuscode: 500
Then Try adding --kubelet-insecure-tls
kubectl edit deploy metrics-server -n kube-system

```
containers:
 - args:
   - --cert-dir=/tmp
   - --secure-port=8448
   - --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname
   - --kubelet-insecure-tls
```

After this restart the deployment

kubectl rollout restart deployment metrics-server -n kube-system

Now check
kubectl -n kube-system get deploy

Run
kubectl top pods

ADDING ADDITIONAL NODE TO CLUSTER:
kubeadm token create --print-join-command

kubeadm join 10.1.1.200:6443 --token azk1md.l4uojch8lssib3a8 --discovery-token-ca-cert-hash sha256:e5c238a5f964a05cdf2e09adfe8d3458eaabb6925327fa76849ae8463d0ea84f

DRAINING and DELETING NODE:
kubectl drain ip-10-1-2-5 --force --ignore-daemonsets

kubectl delete node ip-10-1-2-5

ku drain ip-10-1-2-42 --force --ignore-daemonsets && ku delete node ip-10-1-2-42
ku drain ip-10-1-3-58 --force --ignore-daemonsets && ku delete node ip-10-1-3-58

Role Based Access Control

Role
ClusterRole
RoleBinding
ClusterRoleBinding

Role is limited to namespace
means a role is created for a namespace
Roles can be assigned to users and service accounts.

ClusterROle is a non-namespaced resource

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

—
apiVersion: rbac.authorization.k8s.io/v1
```

```
# This role binding allows "jane" to read pods in the "default"
namespace.

# You need to already have a Role named "pod-reader" in that namespace.

kind: RoleBinding

metadata:

 name: read-pods

 namespace: default

subjects:

# You can specify more than one "subject"

- kind: User

 name: jane # "name" is case sensitive

 apiGroup: rbac.authorization.k8s.io

roleRef:

 # "roleRef" specifies the binding to a Role / ClusterRole

 kind: Role #this must be Role or ClusterRole

 name: pod-reader # this must match the name of the Role or ClusterRole
you wish to bind to

 apiGroup: rbac.authorization.k8s.io
```

Here is an example of a ClusterRole that can be used to grant read access to secrets in any particular namespace, or across all namespaces (depending on how it is bound):

List of apiGroups

https://kubernetes.io/docs/reference/kubectl/#resource-types

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
 # "namespace" omitted since ClusterRoles are not namespaced
 name: secret-reader
rules:
- apiGroups: [""]
 #
 # at the HTTP level, the name of the resource for accessing Secret
 # objects is "secrets"
 resources: ["secrets"]
 verbs: ["get", "watch", "list"]
```

Kubernetes Role for Service Account

Let's consider the following scenario

You have deployments/pods in a namespace called webapps
The deployments/pods need Kubernetes API access to manage resources in a namespace.
The solution to the above scenarios is to have a service account with roles with specific API access.

Create a service account bound to the namespace webapps namespace
Create a role with the list of required API access to Kubernetes resoruces.
Create a Rolebinding to bind the role to the service account.
Use the service account in the pod/deployment

```
kubectl create namespace webapps
```

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: app-service-account
  namespace: webapps
```

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: app-role
  namespace: webapps
rules:
  - apiGroups:
        - ""
        - apps
        - autoscaling
        - batch
        - extensions
        - policy
        - rbac.authorization.k8s.io
    resources:
      - pods
      - componentstatuses
      - configmaps
      - daemonsets
      - deployments
      - events
      - endpoints
      - horizontalpodautoscalers
      - ingress
      - jobs
      - limitranges
      - namespaces
      - nodes
      - pods
      - persistentvolumes
      - persistentvolumeclaims
      - resourcequotas
      - replicasets
      - replicationcontrollers
      - serviceaccounts
      - services
    verbs: ["get", "list", "watch", "create", "update", "patch",
"delete"]
```

kubectl get roles -n webapps

Create a Rolebinding [Attaching Role to ServiceAccount]

With Rolebinding we attach the role to the service account. So the pods which use the service account in webapps namespace will have all the access mentioned in the app-role

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: app-rolebinding
  namespace: webapps
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: app-role
subjects:
- namespace: webapps
  kind: ServiceAccount
  name: app-service-account
```

We will use the bibinwilson/docker-kubectl Docker image with the kubectl utility.
Let's deploy a pod named debug with bibinwilson/docker-kubectl image and our service account app-service-account.

```
apiVersion: v1
kind: Pod
metadata:
  name: debug
  namespace: webapps
spec:
  containers:
  - image: bibinwilson/docker-kubectl:latest
    name: kubectl
  serviceAccountName: app-service-account
```

```
kubectl exec -it debug /bin/bash -n webapps
```

```
To install metrics server
```

kubectl apply -f
https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml
kubectl api-resources
kubectl api-versions

go inside a pod

export KUBECONFIG=config
kubectl get pods

cat config
we are able to access the resources because of the config file where it acts as credential

---
create a service account
create a role
assign role to the service account by using role binding.

---
user account
to give access to the users, we use certificates
kubectl config view

cat /.kube/config
---
to give access to a user ravi,
we should generate
ravi.key and ravi.csr and sign the certificate with kubernetes cluster certificate authority.

once signed we will get ravi.crt
If you install Kubernetes with kubeadm, most certificates are stored in /etc/kubernetes/pki

go to pki(public key infrastructure) folder, go to private folder
and go to ca folder, download key file
now pki/issued/ca and download crt file

mkdir ravi
create a file called CA.key
copy the contents of above download key file and paste into CA.key
now create CA.crt and copy the contents of above crt file.
these CA.key and CA.crt belong to the cluster
using these we have to create certificates for the user.

openssl genrsa -out ravi.key 2048
openssl req -new -key ravi.key -out ravi.csr -subj "/CN=ravi/O=development"
kubectl config view
Copy the ca.crt and ca.kye from etc/kubernetes/pki to ravi folder and execute the following command

openssl x509 -req -in ravi.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out ravi.crt -days 45

To add the user in the Kubeconfig file, we can execute the below command (set-credentials). Please make sure that you provide the correct path to the private key and the certificate of anand.

kubectl config set-credentials ravi --client-certificate ravi.crt --client-key ravi.key

To create kubeconfig for ravi
kubectl --kubeconfig ravi_kubeconfig config set-cluster kubernetes --server https://172.31.14.65:6443 --certificate-authority=ca.crt
kubectl config view

The next step is to add a context in the config file, that will allow this user (ravi) to access the development namespace in the cluster.

kubectl config set-context ravi-context --cluster=kubernetes --namespace=dev-env --user=ravi
kubectl config view
kubectl get pods --context=ravi-context
why
because we just created the user, but user should get the permissions

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: dev-role
  namespace: dev-env
rules:
- apiGroups: ["", "extensions", "apps"] # "" indicates the core API group
  resources: ["pods", "deployments", "replicasets"]
  verbs: ["get", "update", "list", "create", "delete"]
```

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
```

```
metadata:
  name: ravi-RoleBinding
  namespace: dev-env
subjects:
- kind: User
  name: ravi
  apiGroup: ""
roleRef:
  kind: Role
  name: dev-role
  apiGroup: ""
```

```
kubectl get pods --context=ravi-context
```

Go to ~/.kube/config
And copy the ravi content.
Create ravi_config and paste the content.

Put ravi certificates and ravi_config in one folder and give it to ravi.

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: payroll-cluster-wide-role
rules:
- apiGroups: [\"\", "extensions", "apps"] # "" indicates the core API
group
  resources: ["pods", "deployments", "replicasets"]
  verbs: ["get", "update", "list", "create", "delete"]
```

```
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: ClusterRole-Anand
subjects:
- kind: User
  name: ravi
  apiGroup: \"\"
roleRef:
  kind: ClusterRole
  name: payroll-cluster-wide-role
  apiGroup: \"\"
```

kubectl --kubeconfig=ravi_config get pods
kubectl --kubeconfig=ravi_config get run nginx - - image=nginx
kubectl --kubeconfig=ravi_config config get-contexts
kubectl --kubeconfig=ravi_config config use-context <context-name>

Lens IDE for Kubernetes

https://k8slens.dev/

K9s for Kubernetes

https://webinstall.dev/k9s/
https://k9scli.io/topics/install/

Probes in Kubernetes

Pod is a collection of 1 or more docker containers. It is an atomic unit of scaling in Kubernetes. Pod has a life-cycle with multiple phases. For example, When we deploy a pod in the Kubernetes cluster, Kubernetes has to start from scheduling the pod in one of the nodes in the cluster, pulling the docker image, starting a container and ensuring that containers are ready to serve the traffic etc!

As Pod is the collection of docker containers, in order to accept any incoming request, all the containers must be ready to serve the requests! So it will take some time – usually within a minute & but it mostly depends on the application. So, as soon as we send a deployment request, our application is not ready to serve! Also, we know that software will eventually fail! Anything could happen.For example, a memory leak could lead to OOM error in few hours/days. In that case, Kubernetes has to kill the pod when the pods are not working as expected and reschedule another pod to handle the load on the cluster.

The kubelet uses liveness probes to know when to restart a container. For example, liveness probes could catch a deadlock, where an application is running, but unable to make progress. Restarting a container in such a state can help to make the application more available despite bugs.

The kubelet uses readiness probes to know when a container is ready to start accepting traffic. A Pod is considered ready when all of its containers are ready. One use of this signal is to control which Pods are used as backends for Services. When a Pod is not ready, it is removed from Service load balancers.

Liveness Probes: Used to check if the container is available and alive.
Readiness Probes: Used to check if the application is ready to be used and serve the traffic.

Navigate to https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/#before-you-begin

http://www.vinsguru.com/kubernetes-liveness-probe-vs-readiness-probe/

EFK Deployment
https://github.com/scriptcamp/kubernetes-efk

==========

Helm Charts

Helm is widely known as "the package manager for Kubernetes".

Helm uses a packaging format called *charts*. A chart is a collection of files that describe a related set of Kubernetes resources. A single chart might be used to deploy something simple pod, or something complex, like a full web app stack with HTTP servers, databases, caches, and so on.

Charts are created as files laid out in a particular directory tree. They can be packaged into versioned archives to be deployed.

Install Helm Chart Using Script

https://helm.sh/docs/intro/install/

For Linux

```
curl -fsSL -o get_helm.sh
```

```
https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3
```

```
chmod 700 get_helm.sh
```

```
./get_helm.sh
helm version
```

For Windows

https://get.helm.sh/helm-canary-windows-amd64.zip

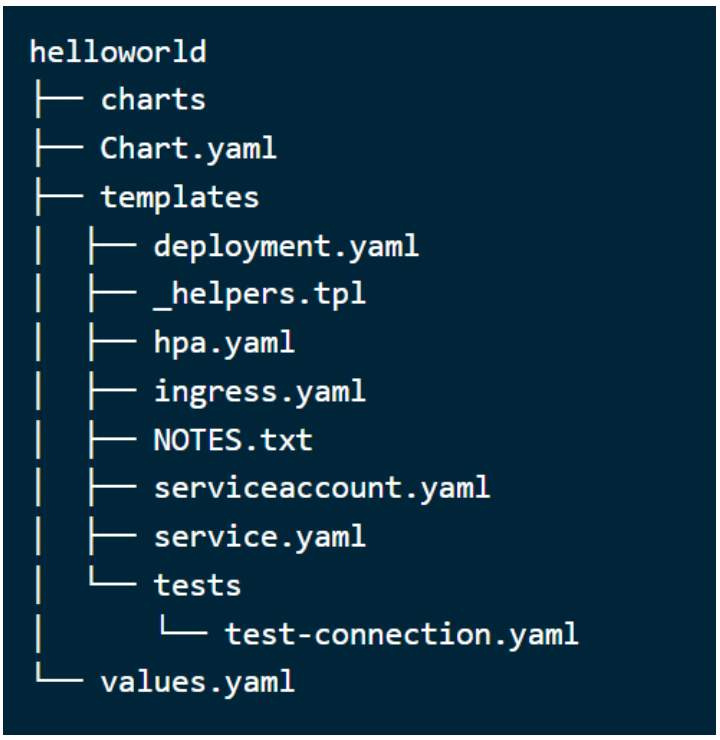
```
helm version
```

We are going to create our first helloworld Helm Chart using the following command

helm create helloworld

It should create a directory helloworld, you can verify it by using the following ls -lart command
To verify the complete directory structure of the HelmChart please do run the command

```
tree helloworld
```



Great now you created your first Helm Chart - helloworld.

In the next steps we are going to run the helloworld Helm Chart.

Update the service.type from ClusterIP to NodePort inside the values.yml

Before you run your helloworld Helm Chart we need to update the service.type from ClusterIP to NodePort.

The reason for this change is - After installing/running the helloworld Helm Chart we should be able to access the service outside of the kubernetes cluster. And if you do not change the service.type then you will only be able to access the service withing kubernetes cluster.

To update the values.yml, first go inside the directory helloworld

```
cd helloworld
```

After that open the `values.yml` in `vi`

```
vi values.yaml
```

Look for the service.type block and update its value to NodePort

```
service:
  type: NodePort
  port: 80
```

Install the Helm Chart using command - helm install

Now after updating the values.yml, you can install the Helm Chart.

Note : The helm install command take two arguments -

First argument - Release name that you pick
Second argument - Chart you want to install
It should look like -

helm install <FIRST_ARGUMENT_RELEASE_NAME> <SECOND_ARGUMENT_CHART_NAME>

```
helm install myhelloworld helloworld
```

```
NAME: myhellworld
LAST DEPLOYED: Sat Nov  7 21:48:08 2020
NAMESPACE: default
STATUS: deployed
REVISION: 1
NOTES:
1. Get the application URL by running these commands:
  export NODE_PORT=$(kubectl get --namespace default -o jsonpath="{.spec.ports[0].nodePort}" ser
  export NODE_IP=$(kubectl get nodes --namespace default -o jsonpath="{.items[0].status.addresse
  echo http://$NODE_IP:$NODE_PORT
```

Verify the helm install command

Now you need to verify your helm release .i.e. myhelloworld and which can be done by running the helm list command.

```
helm list -a
```

```
NAME            NAMESPACE       REVISION        UPDATED                                 STATUS
myhelloworld    default         1               2020-11-07 21:48:08.8550677 +0000 UTC   deployed
```

Get kubernetes Service details and port

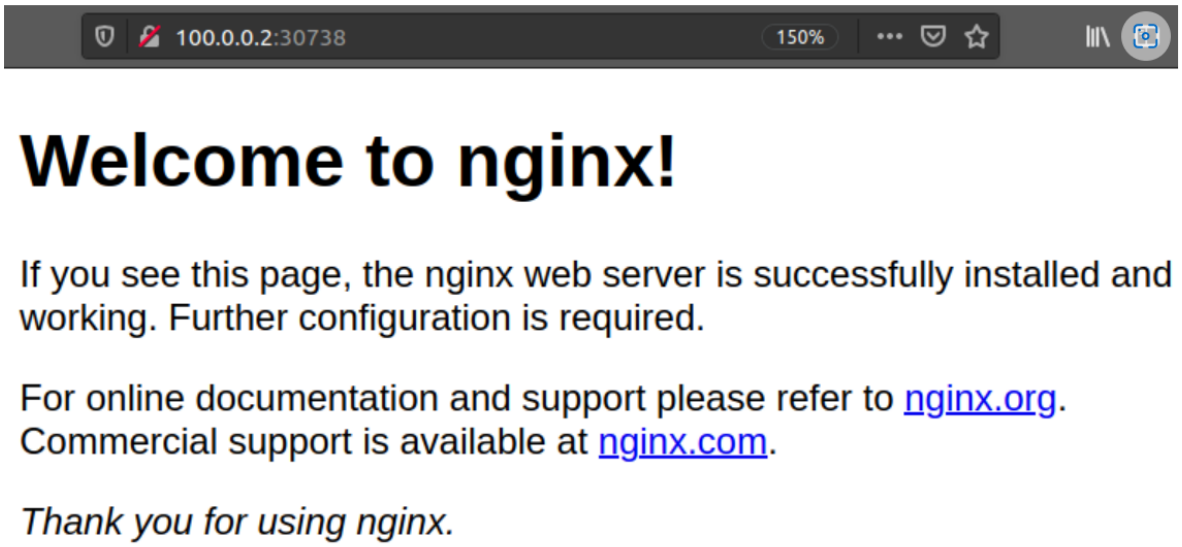
Lets run the kubectl get service command to get the NodePort.

```
kubectl get service
```

```
NAME                    TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
kubernetes              ClusterIP   10.233.0.1      <none>        443/TCP        14d
myhellworld-helloworld  NodePort    10.233.14.134   <none>        80:30738/TCP   7m10s
```

Keep in mind the NodePort number can vary in the range 30000-32767, so you might get different NodePort.

Since my cluster ip is 100.0.0.2 and NodePort is 30738, so I can access my Nginx page of my myhelloworld Helm Chart



Helm: Adding upstream repositories

We have apt,yum,dnf package manager in Linux distros, similarly Helm relies on bitnami chart repositories and Chart Developer can create YAML configuration file and package them into charts and publish it as chart repositories.

For Example - You want to deploy Redis in-memory cache inside your kubernetes cluster from Helm repository, so you can simply run the following command -

```
helm install redis bitnami/redis
```

The above command will search for the redis chart inside bitnami chart repository and then it will install the redis chart inside your kubernetes cluster.

How to ADD upstream Helm chart repository

There are five repo commands provided by Helm which can be used for add,list,remove,update,index the chart repository.

1. add : Add chart repository
2. list : List chart repository
3. update : Update the chart information locally
4. index : For generating the index file
5. remove : Remove chart repository

Deploying Prometheus and Grafana using Helm

helm repo add bitnami https://charts.bitnami.com/bitnami

https://github.com/bitnami/charts/tree/main/bitnami

helm install prometheus bitnami/kube-prometheus

Read more about the installation in the
https://github.com/bitnami/charts/tree/main/bitnami/kube-prometheus/#installing-the-chart

Deploying grafana

helm repo add bitnami https://charts.bitnami.com/bitnami

helm install grafana bitnami/grafana

https://grafana.com/grafana/dashboards/6417-kubernetes-cluster-prometheus/
—----------

https://github.com/kubernetes/examples/tree/master/mysql-wordpress-pd

Statefulsets

https://github.com/microservices-demo/microservices-demo/tree/master/deploy/kubernetes/manifests
https://kubernetes.io/docs/concepts/services-networking/service/#headless-services

Project

https://github.com/microservices-demo/microservices-demo/tree/master/deploy/kubernetes/manifests
https://github.com/eginnovations/webstore/blob/master/K8s-yamls/webstore-hpa-withBTM.yaml