Kubernetes

- **Kubernetes** — also known as “k8s” or “kube” — is a container orchestration platform for scheduling and automating the deployment, management, and scaling of containerized applications.

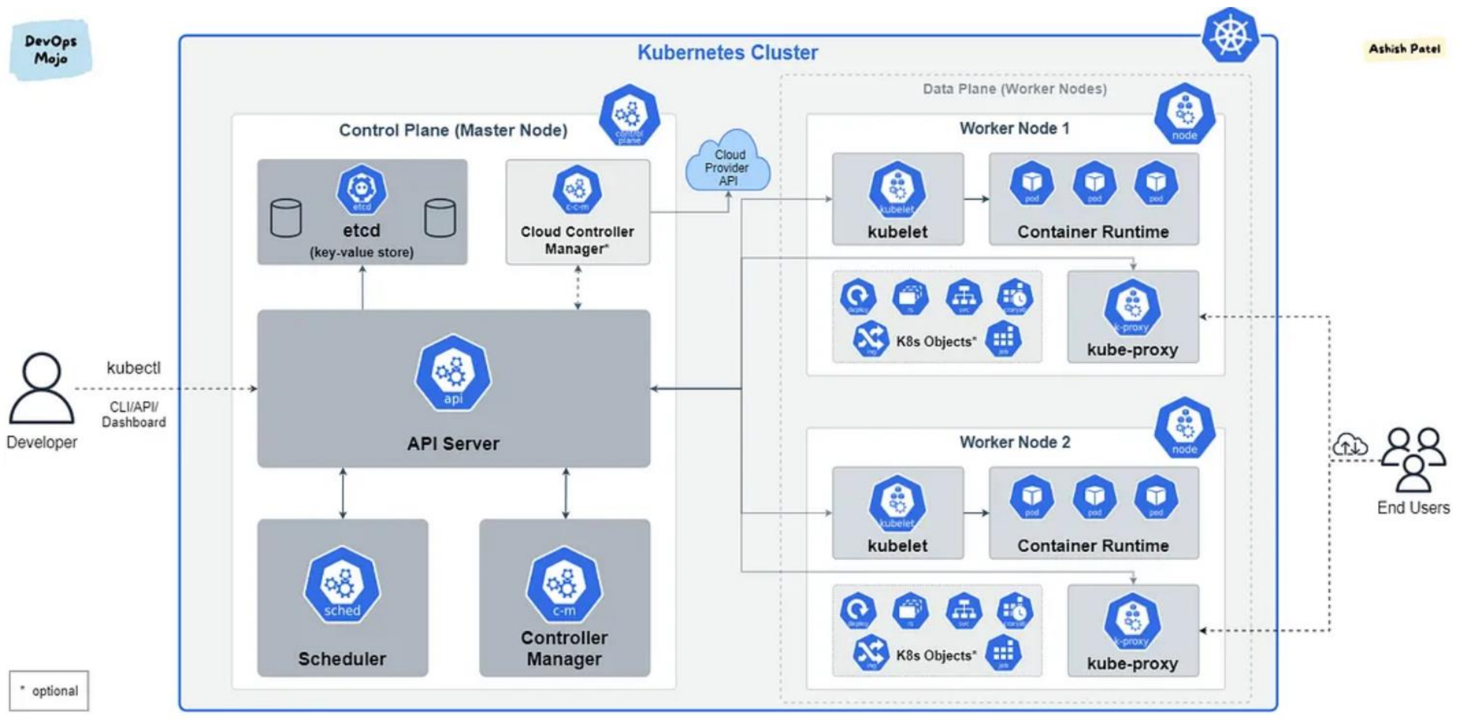
- **Characteristics of Kubernetes:**

  Kubernetes schedules and automates container-related tasks throughout the application lifecycle, including:

  > **Deployment**: Deploy a specified number of containers to a specified host and keep them running in a desired state.
  >
  > **Rollouts**: A rollout is a change to a deployment. Kubernetes lets you initiate, pause, resume, or roll back rollouts.
  >
  > **Service discovery**: Kubernetes can automatically expose a container to the internet or to other containers using a DNS name or IP address.
  >
  > **Storage provisioning**: Set Kubernetes to mount persistent local or cloud storage for your containers as needed.
  >
  > **Load balancing**: Based on CPU utilization or custom metrics, Kubernetes load balancing can distribute the workload across the network to maintain performance and stability.
  >
  > **Autoscaling**: When traffic spikes, Kubernetes autoscaling can spin up new clusters as needed to handle the additional workload.
  >
  > **Self-healing for high availability**: When a container fails, Kubernetes can restart or replace it automatically to prevent downtime. It can also take down containers that don’t meet your health-check requirements.

- **Architecture:**

  - **Clusters** are building blocks of Kubernetes architecture. The Cluster are made up of **nodes**, each of which represents single computer hosts (virtual or physical machine).
  - Kubernetes cluster consists of two types of node:
    1. Master Node
    2. Worker Node



Kubernetes Architecture and Components

Master Node

- Master node serves as the control plane for the cluster, and multiple worker nodes that deploy, run, and manage containerized applications.
- Master node consists of four processes within
  1. **API Server**
  2. **Scheduler**
  3. **Controller Manager**
  4. **Etcd**

1. API Server:

- The API server is the front end for the Kubernetes control plane.
- Whenever user needs to deploy a new application in the kubernetes cluster, user interacts with API server using client such as kubernetes dashboard, kubectl or kubernetes API.
- It acts as a cluster gateway which gets an initial request of any updates or queries from the cluster.
- It also acts as a gatekeeper for authentication such that only authorized request gets through the cluster.
- Any request for scheduling a pod, deploying an application and creating new services or components are directly sent to API server for validation and upon successful validation the requests are then forwarded to other processes.

$$\begin{tikzcd}
\boxed{\text{Requests}} \arrow[r] & \boxed{\text{API Server}} \arrow[r] & \boxed{\text{Validation}} \arrow[r] & \boxed{\text{Other processes}} \arrow[r] & \boxed{\text{POD}}
\end{tikzcd}$$

2. **Scheduler:**
    - Scheduler is a process that is responsible to decide where to allocate a new pod on a worker node based on resource requirements such as CPU, RAM, and storage.
    - Pods are allocated on a worker node based on available resources among the already running nodes. For example if there are two worker nodes A & B available with 80% and 40% usage of resources respectively, scheduler selects the node B to balance the usage of node.
    - <u>Note</u>: **Scheduler is responsible to decide and select the node for a pod whereas kubelet component inside a node is responsible to run the pod.**

3. **Controller Manager:**
    - Controller Manager is important process that detects the state changes of pods and nodes.
    - It detects if any of the pod or node is crashed and requests the scheduler to replace it with a healthy one.
    - So it is basically responsible for damage control in kubernetes.

4. **etcd:**
    - etcd is an open source distributed key-value store used to hold and manage the critical information that distributed systems need to keep running. Most notably, it manages the configuration data, state data, and metadata for Kubernetes, the popular container orchestration platform.
    - Every changes in the cluster are logged into key-value store of etcd.
    - Other processes in the master node gets the data from etcd to perform operation on worker node and vice-versa.
    - **<u>Note</u>: Application data are not stored in etcd!**

<u>Worker Node</u>

- The worker nodes are the part of the kubernetes cluster which actually executes the containers and application on them.
- They also handle networking to ensure that traffic between applications across the cluster and from outside of the cluster can be properly facilitated.
- Components of a worker node:
    1. **Kubelet**
    2. **Kube-proxy**
    3. **Container runtime**

1. **Kubelet:**

   - An agent that runs on each node in the cluster. It makes sure that containers are running in a Pod.

   - The kubelet takes a set of Pod Specs that are provided through various mechanisms and ensures that the containers described in those Pod Specs are running and healthy. The kubelet doesn't manage containers which were not created by Kubernetes.

2. Kube-Proxy:

- Kube-proxy is a network proxy that runs on each node in your cluster, implementing part of the Kubernetes Service concept.

- Kube-proxy maintains network rules on nodes. These network rules allow network communication to your Pods from network sessions inside or outside of your cluster.

- Kube-proxy uses the operating system packet filtering layer if there is one and it's available. Otherwise, kube-proxy forwards the traffic itself.

3. Container Runtime:

- The container runtime is the software that is responsible for running containers.

- Kubernetes supports container runtimes such as Docker container, CRI-O, and any other implementation of the Kubernetes CRI (Container Runtime Interface).

<u>**Other Components:**</u>

- **Pod:**
  - Pod is a smallest deployable unit in k8s.
  - A pod is a collection of containers and it’s stored inside a node of a Kubernetes cluster. It is possible to create a pod with multiple containers inside it. For example, keeping a database container and app container in the same pod.
  - There are two types of Pods –
    - Single container pod
    - Multi container pod
  - Pods are assigned their own IP address and can communicate with other pod within same node. Pods are allocated a new IP address whenever they are re-created upon crashing.
  -

- **Service:**
  - Service is a component that provides static IP address and DNS name that can be attached to each pod.

  - Service also acts as load balancers. Whenever there is access request, service forwards the request to least busy pod and reduces latency.

- **Ingress:**
  - Ingress is component in k8s which is used when user wants to access the application frontend externally from a browser.

  - As a Service component in a node facilitates the communication internally between the pods, Ingress is used to establish a connection from an external source.

- **ConfigMap:**
  - A ConfigMap is an API object used to store non-confidential data in key-value pairs. Pods can consume ConfigMaps as environment variables, command-line arguments, or as configuration files in a volume.

  - For example, imagine that you are developing an application that you can run on your own computer (for development) and in the cloud (to handle real traffic). You write the code to look in an environment variable named DATABASE_HOST. Locally, you set that variable to localhost. In the cloud, you set it to refer to a Kubernetes Service that exposes the database component to your cluster. This lets you fetch a container image running in the cloud and debug the exact same code locally if needed.

- **Secret:**
  - A Secret is an object that contains a small amount of sensitive data such as a password, a token, or a key. Such information might otherwise be put in a Pod specification or in a container image. Using a Secret means that you don't need to include confidential data in your application code.

  - Secrets are similar to ConfigMaps but are specifically intended to hold confidential data.

- **Volumes:**
  - In Kubernetes, a volume can be thought of as a directory which is accessible to the containers in a pod. We have different types of volumes in Kubernetes and the type defines how the volume is created and its content.
  - The concept of volume was present with the Docker, however the only issue was that the volume was very much limited to a particular pod. As soon as the life of a pod ended, the volume was also lost.

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

- **Deployment:**
  - Deployments are upgraded and higher version of replication controller. They manage the deployment of replica sets which is also an upgraded version of the replication controller. They have the capability to update the replica set and are also capable of rolling back to the previous version.

  - Common service (static IP address & DNS) are used by replicas of application pods.
  - Deployment is a blue-print for pods.
  - User actually writes a Deployment file with all the metadata and configurations to deploy a pod or container.
  - Database pods cannot be replicated via deployment as it has state i.e., multiple application pods accessing a DB pods can lead to overriding the data and inconsistency.

Changing the Deployment

- **Updating** − the user can update the ongoing deployment before it is completed. In this, the existing deployment will be settled and new deployment will be created.
- **Deleting** − the user can pause/cancel the deployment by deleting it before it is completed. Recreating the same deployment will resume it.
- **Rollback** − we can roll back the deployment or the deployment in progress. The user can create or update the deployment.

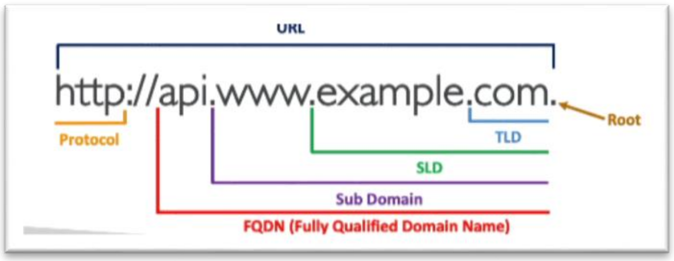
Deployment Strategies

Deployment strategies help in defining how the new RC should replace the existing RC.

- **Recreate** − this feature will kill all the existing RC and then bring up the new ones. This results in quick deployment however it will result in downtime when the old pods are down and the new pods have not come up.
- **Rolling Update** − this feature gradually brings down the old RC and brings up the new one. This results in slow deployment, however there is no deployment. At all times, few old pods and few new pods are available in this process.

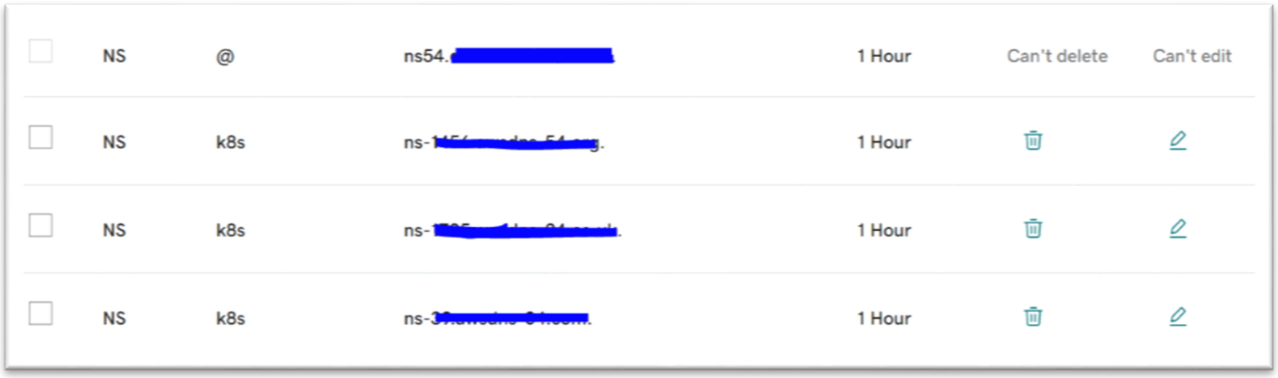
Setup Kubernetes cluster with Kops

- **Prerequisites:**
    - Domain for Kubernetes DNS record(GoDaddy)
    - Create a Linux VM and setup
        - Kops, kubectl, ssh keys, awscli
    - Login to AWS account and Setup
        - S3 Bucket, IAM user for AWS cli, Route53 hosted Zone


URL
http://api.www.example.com.
Root
Protocol
TLD
SLD
Sub Domain
FQDN (Fully Qualified Domain Name)

- **Steps:**

1. Launch an EC2 instance (with port 22 ssh sg)
2. Create an S3 bucket to store the **state of Kops** so that kops command can be run from anywhere as we redirect to that state bucket.
3. Create an IAM user with AWS CLI access.
4. Create a Route 53 Hosted Zone and add sub-domains to NS Records in Go daddy.
    - Create a Route53 hosted zone with Domain name being <hostaname>.<registeredSLD>(ex: k8s.nrvs.xyz)



☐	NS	@	ns54.[redacted]	1 Hour	Can't delete	Can't edit
☐	NS	k8s	ns-[redacted].	1 Hour	🗑	✎
☐	NS	k8s	ns-[redacted].	1 Hour	🗑	✎
☐	NS	k8s	ns-[redacted].	1 Hour	🗑	✎

    - Add the Name Servers (NS) in Go daddy records under registered Domain as shown in above image.

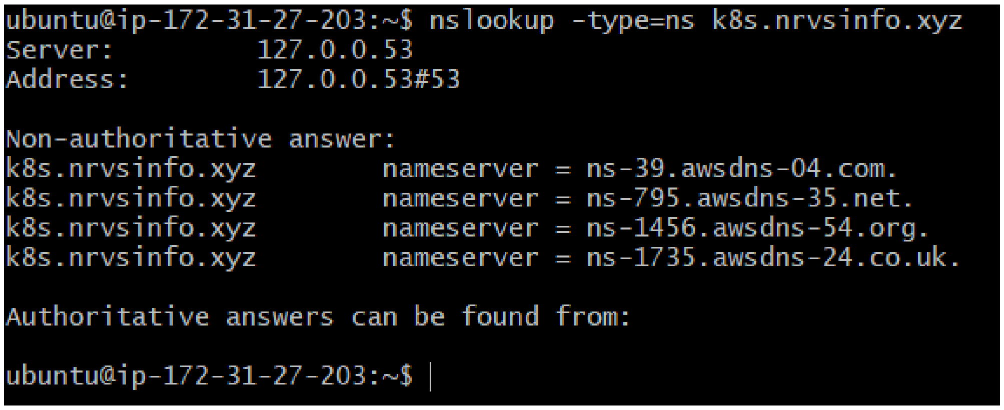
5. Login to EC2 instance via ssh
6. ssh-keygen -> Create a ssh key
7. sudo apt update && sudo apt install awscli –y -> Install AWS CLI

8. aws configure -> Add access key and secret key for IAM user

9. **Install and setup kubectl** (https://kubernetes.io/docs/tasks/tools/install-kubectl-linux/)
   - curl -LO https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl
   - ls -> To view kubectl binary file in directory
   - chmod +x kubectl -> to provide executable permission
   - sudo mv kubectl /usr/local/bin/ -> Move the file to bin directory to run the commands from anywhere in server
   - kubectl version -> to confirm the version

10. **Install and setup Kops** (https://kops.sigs.k8s.io/getting_started/install/)
   - curl -Lo kops https://github.com/kubernetes/kops/releases/download/$(curl -s https://api.github.com/repos/kubernetes/kops/releases/latest | grep tag_name | cut -d '"' -f 4)/kops-linux-amd64
   - chmod +x kops
   - sudo mv kops /usr/local/bin/

11. Verify the domain name
   - nslookup –type=ns k8s.nrvsinfo.xyz



```
ubuntu@ip-172-31-27-203:~$ nslookup -type=ns k8s.nrvsinfo.xyz
Server:         127.0.0.53
Address:        127.0.0.53#53

Non-authoritative answer:
k8s.nrvsinfo.xyz        nameserver = ns-39.awsdns-04.com.
k8s.nrvsinfo.xyz        nameserver = ns-795.awsdns-35.net.
k8s.nrvsinfo.xyz        nameserver = ns-1456.awsdns-54.org.
k8s.nrvsinfo.xyz        nameserver = ns-1735.awsdns-24.co.uk.

Authoritative answers can be found from:

ubuntu@ip-172-31-27-203:~$ |
```

12. Create a kubernetes cluster with kops

- kops create cluster --name=k8s.nrvsinfo.xyz --state=s3://nrvs-kops-state --zones=us-east-1a,us-east-1b --node-count=2 --node-size=t3.small --master-size=t3.medium --dns-zone=k8s.nrvsinfo.xyz --node-volume-size=8 --master-volume-size=8
  -> Create a cluster configuration
- kops update cluster --name k8s.nrvsinfo.xyz --state=s3://nrvs-kops-state --yes --admin
  -> to run the configuration in AWS and create a k8s cluster
- kops validate cluster --state=s3://nrvs-kops-state -> to view the status of nodes

  - kubectl get nodes -> To get the info of instances initiated by AWS through k8s
  - kops delete cluster --name k8s.nrvsinfo.xyz --state=s3://nrvs-kops-state --yes
    ->To terminate the instances in AWS and delete the k8s cluster

- **KubeConfig File:**

  Kubeconfig is a configuration file used by the Kubernetes command-line tool, kubectl, to interact with Kubernetes clusters. It contains information about the cluster, authentication details, and other settings necessary for establishing a connection to a Kubernetes cluster. Here are some important points to consider about kubeconfig:

  1. **Structure**: Kubeconfig is a YAML file that consists of multiple sections, including clusters, users, and contexts. Each section contains relevant information for establishing connections to the cluster.

  2. **Cluster Configuration**:

  - Clusters section: This section defines the cluster's endpoint URL and any associated certificate authority data.

  - Cluster name: A unique name for the cluster.

  - Cluster server: The URL or IP address of the Kubernetes API server.

  - Cluster certificate authority: The certificate authority data used to verify the authenticity of the server's TLS certificate.

  3. **User Configuration:**

  - Users section: This section defines the user credentials or authentication mechanism used to access the cluster.

  - User name: The name of the user.

  - User credentials: This can include a client certificate, client key, token, or other authentication details.

  4. **Context Configuration:**

  - Contexts section: This section binds a cluster to a user and specifies the default namespace to use.

  - Context name: A unique name for the context.

  - Context cluster: The cluster to use for this context.

  - Context user: The user to authenticate with for this context.

  - Context namespace: The default namespace to use for this context.

  5. **Switching Contexts:** Kubectl allows switching between different contexts to work with multiple clusters or switch between different user credentials.

6. **Multiple Clusters and Users**: Kubeconfig can store configurations for multiple clusters and users. This is helpful when managing access to different Kubernetes environments.

7. **Location**: By default, kubectl looks for the kubeconfig file in the `~/.kube/config` directory. However, you can specify a different file location using the `--kubeconfig` flag.

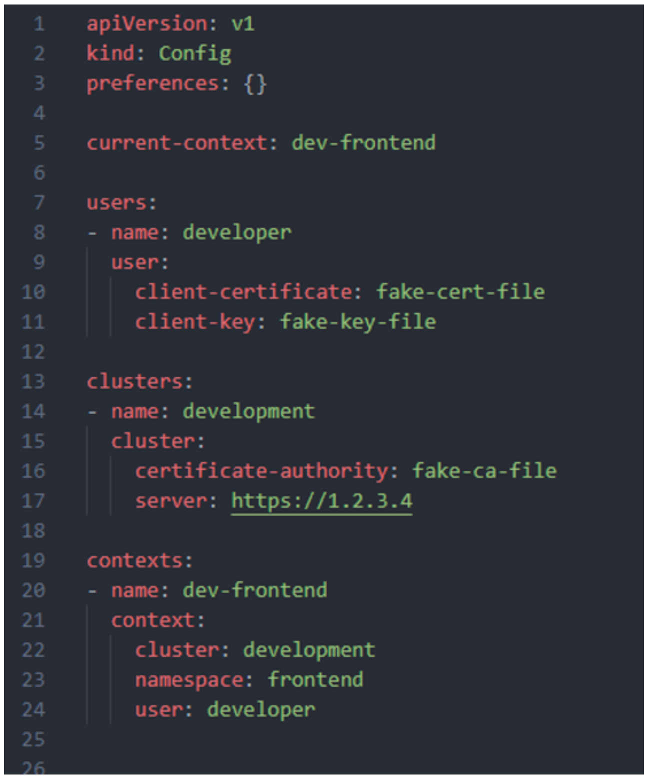
8. **Integration with Tools:** Various tools and libraries in the Kubernetes ecosystem, such as Kubernetes Dashboard and client libraries, utilize the kubeconfig file for authentication and cluster access.

9**. Cluster Access and Permissions:** The kubeconfig file should be protected and accessible only to authorized users, as it contains sensitive information like credentials and access details.

10. **Generating Kubeconfig:** Kubeconfig files can be manually created and edited or generated automatically using tools like `kubeadm` or cloud provider-specific utilities.

Remember that kubeconfig files can vary depending on the cluster's setup and the authentication mechanisms used. Understanding kubeconfig is crucial for managing and interacting with Kubernetes clusters using kubectl or other Kubernetes tools.

Note: Kubeconfig file can be copied from a host server to any other machine with kubectl installed on it and run the commands.



```yaml
apiVersion: v1
kind: Config
preferences: {}

current-context: dev-frontend

users:
- name: developer
  user:
    client-certificate: fake-cert-file
    client-key: fake-key-file

clusters:
- name: development
  cluster:
    certificate-authority: fake-ca-file
    server: https://1.2.3.4

contexts:
- name: dev-frontend
  context:
    cluster: development
    namespace: frontend
    user: developer
```

Namespaces:

Namespaces are a way to organize clusters into virtual sub-clusters — they can be helpful when different teams or projects share a Kubernetes cluster. Any number of namespaces are supported within a cluster, each logically separated from others but with the ability to communicate with each other.

Each Namespaces must have their own configMap and secret files.

Command: kubectl get namespace

There are four initial namespaces:

1. **default**
    - Kubernetes includes this namespace so that you can start using your new cluster without first creating a namespace.
2. **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.
3. **kube-public**
    - This namespace shows publicly accessed data
    - It has a ConfigMaps, which contains the information of cluster**.**
4. **kube-system**
    - This namespace is created for system processes such as Master and kubectl.

Create a namespace

- kubectl create ns <NAMESPACE> -> to create a namespace

  Or
- Create a new YAML file called my-namespace.yaml

```
apiVersion: v1
kind: Namespace
metadata:
  name: <insert-namespace-name-here>
```

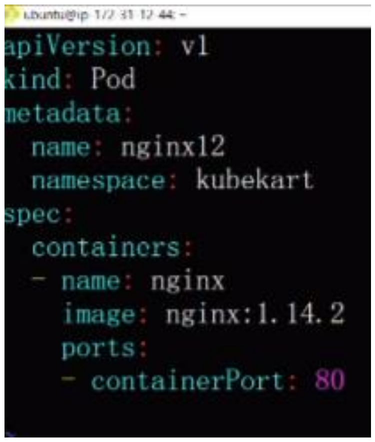
- kubectl create -f ./my-namespace.yaml -> to run the yaml file and create a NS

- kubectl run nginx1 --image=nginx -n <NAMESPACE > -> to run a pod in created namespace
- kubectl delete <NAMESPACE> -> to delete a namespace

To group the pod or any other resource into namespace:

- vim pod1.yaml -> to edit the pod and add the namespace

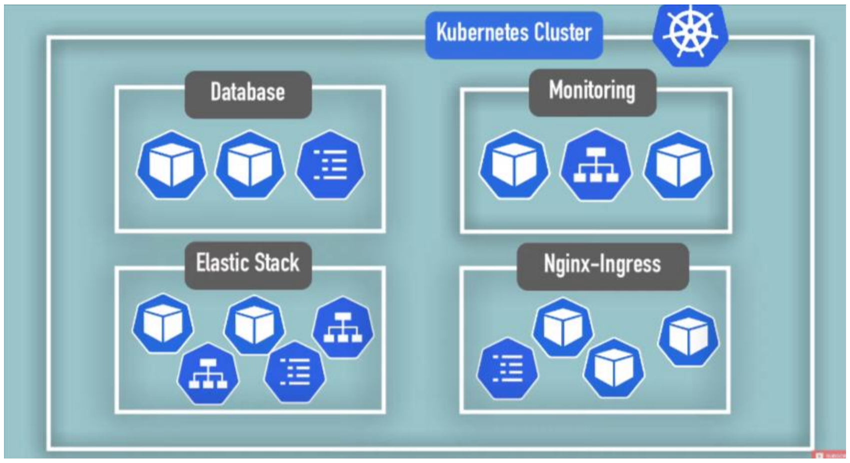
- add the namespace in metadata section
- kubectl apply -f pod1.yaml -> to apply the changes and create a pod



```
apiVersion: v1
kind: Pod
metadata:
  name: nginx12
  namespace: kubekart
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

Use case of Namespace:

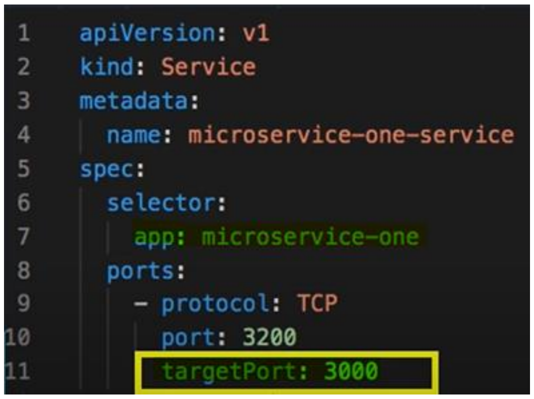
1. Whenever a complex project with more resources are deployed in a k8s cluster, by default all pods are grouped into a default namespace. This grouping of all of the pods in single logical space would be hectic and unorganized. To overcome this problem user can create a separate namespace for each group of resources and avoid unnecessary



2. When multiple teams are using a same k8s cluster, teams might accidentally use a same labeling of resources which then over-rides each other's work. To avoid this problem, teams can create their own logical namespaces and run their pods on it.
3. Two namespaces can share a common resources without having to setup an extra pods/nodes.
4. To limit the access of resources on namespaces such as CPU, RAM, Storage.

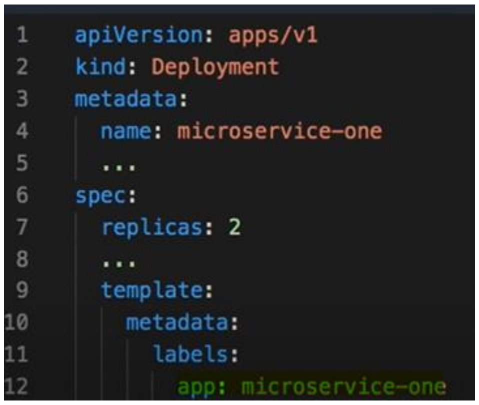
- **Service:**

- Every pod in a node has its own IP address. Upon replacing the pod, its IP address too is replaced with a new one and this frequent changing of IP address each time causes an issue with connecting to the pod network.
- To overcome this problem, K8s has a **service** component which create a **static IP address** and acts a **load balancer**.
- Service establishes a connection within (among different k8s components) and outside (browser) the cluster.
- kubectl get pod -o wide -> To view the IP address of pods.
- Pods are added to a service or pods are identified to be a part of service by mentioning label selectors and port in a yaml file. And the requests are forwarded to only those pods with a label selectors & port defined.



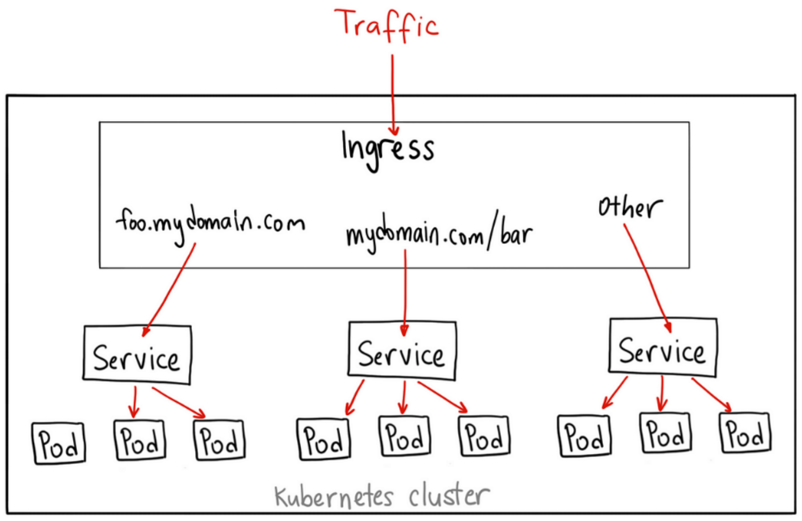
```yaml
apiVersion: v1
kind: Service
metadata:
  name: microservice-one-service
spec:
  selector:
    app: microservice-one
  ports:
    - protocol: TCP
      port: 3200
      targetPort: 3000
```

Service.yaml



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: microservice-one
  ...
spec:
  replicas: 2
  ...
  template:
    metadata:
      labels:
        app: microservice-one
```

deployment.yaml



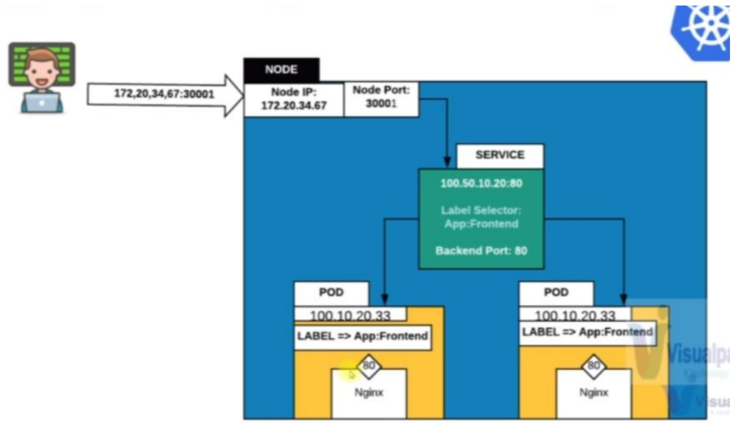
- **Types of Service:**
  1. ClusterIP Services
  2. NodePort Service
  3. LoadBalancer Services

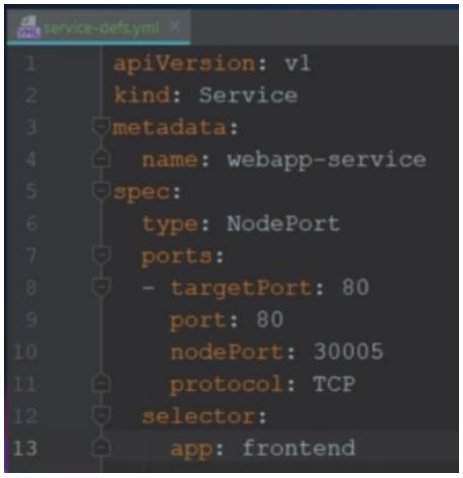
1. **ClusterIP Services**:

- In Kubernetes, the **ClusterIP Service is used for Pod-to-Pod communication within the same cluster**. This means that a client running outside of the cluster, such as a user accessing an application over the internet, cannot directly access a ClusterIP Service.
- When a ClusterIP Service is created, it is assigned a static IP address. This address remains the same for the lifetime of the Service. When a client sends a request to the IP address, the request is automatically routed to one of the Pods behind the Service. If multiple Pods are associated, the ClusterIP Service uses load balancing to distribute traffic equally among them.
- You can expose the Service to the public internet using an Ingress or a Gateway.

2. **NodePort Service:**

- The **NodePort Service is a way to expose your application to external clients**. An external client is anyone who is trying to access your application from outside of the Kubernetes cluster.
- The NodePort Service does this by opening the port you choose (in the range of 30000 to 32767) on all worker nodes in the cluster. This port is what external clients will use to connect to your app. So, if the nodePort is set to 30020, for example, anyone who wants to use your app can just connect to any worker node's IP address, on port :30020.
- Note that a NodePort Service builds on top of the ClusterIP Service type. What this means is that when you create a NodePort Service, Kubernetes automatically creates a ClusterIP Service for it as well. The node receives the request, the NodePort Service picks it up, it sends it to the ClusterIP Service, and this, in turn, sends it to one of the Pods behind it (External Client->Node->**NodePort**->ClusterIP->Pod). And the extra benefits are that internal clients can still access those Pods, and quicker. They can just skip going through the NodePort and reach the ClusterIP directly to connect to one of the Pods.
- One disadvantage of the NodePort Service is that it doesn't do any kind of load balancing across multiple nodes. It simply directs traffic to whichever node the client connected to. This can create a problem: Some nodes can get overwhelmed with requests while others sit idle.





```yaml
apiVersion: v1
kind: Service
metadata:
  name: webapp-service
spec:
  type: NodePort
  ports:
  - targetPort: 80
    port: 80
    nodePort: 30005
    protocol: TCP
  selector:
    app: frontend
```

Service-def.yaml

Steps:

1. **To create a pod:**

- mkdir definition
- cd definition
- mkdir app
- cd app
- vim deploy.yaml -> write a deployment configuration & ensure to use label and port.

```
-   apiVersion: apps/v1
-   kind: Deployment
-   metadata:
-     name: vproapp
-     labels:
-       app: vproapp
-   spec:
-     replicas: 2
-     selector:
-       matchLabels:
-         app: vproapp
-     template:
-       metadata:
-         labels:
-           app: vproapp
-       spec:
-         containers:
-         - name: vproapp
-           image: imranvisualpath/freshtomapp:V7
-           ports:
-           - containerPort: 8080
```

- kubectl create –f deploy.yaml -> to create a deployment

Troubleshoot (While creating a pod):

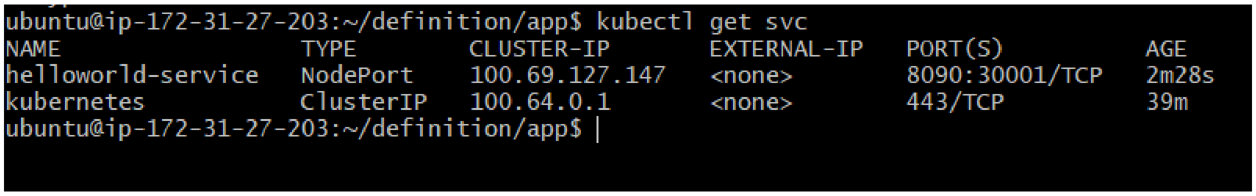
Error from server (BadRequest): error when creating "vproapppod.yaml": Pod in version "v1" cannot be handled as a Pod: strict decoding error: unknown field "metadata.lables"

Solution: Indentation error in a pod yaml file

- cd ..
- vim service.yaml

```
-   apiVersion: v1
-   kind: Service
-   metadata:
-     name: vproapp-service
-   spec:
-     selector:
-       app: vproapp
-     type: NodePort
-     ports:
-     - protocol: TCP
-       port: 8090
-       targetPort: 8080
```

- kubectl create –f service.yaml -> to create a service
- kubectl get svc -> to List all Services
- kubectl describe services -> to describe all services
- kubectl get services -o yaml -> to list all services in yaml format

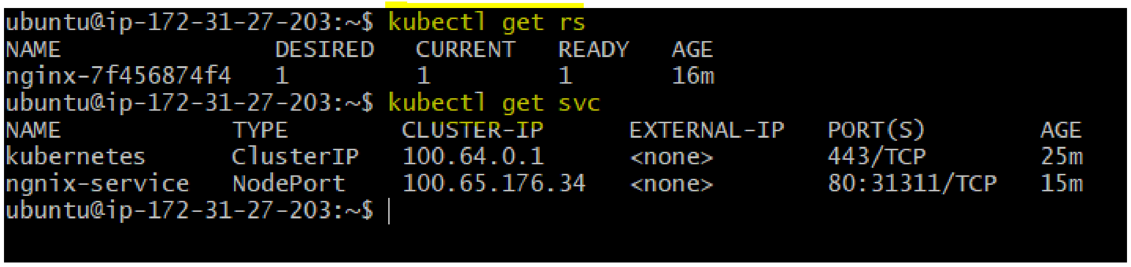


```
ubuntu@ip-172-31-27-203:~/definition/app$ kubectl get svc
NAME                 TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
helloworld-service   NodePort    100.69.127.147   <none>        8090:30001/TCP   2m28s
kubernetes           ClusterIP   100.64.0.1       <none>        443/TCP          39m
ubuntu@ip-172-31-27-203:~/definition/app$ |
```

- **Note**: Endpoint value in service description should match with pod IP address.
- kubectl describe pod | grep IP -> to view the IP address of a Pod.
- Ensure security group for node has http/https inbound rule.
- Open any browser and access the web page at http://<node IP address>:30001 (verify node port which ranges from 30000-32767. In above its 30001)

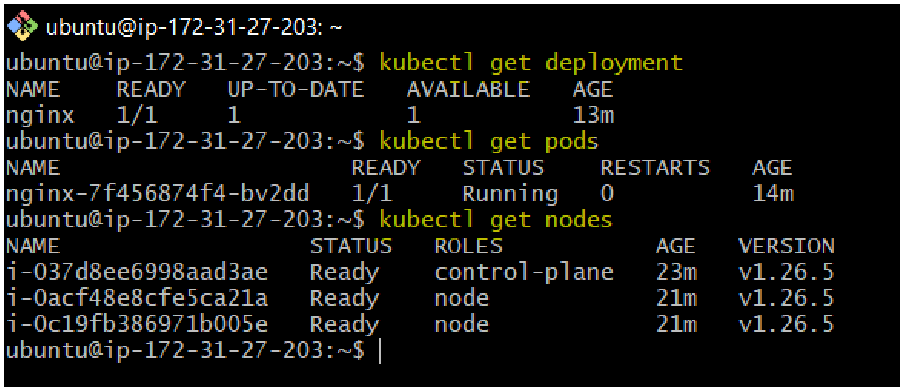
Troubleshooting: Not able to view the Nginx welcome page in website for below deployment description ([Click Here]() for Example deployment)

1. Verify replica set & service running:



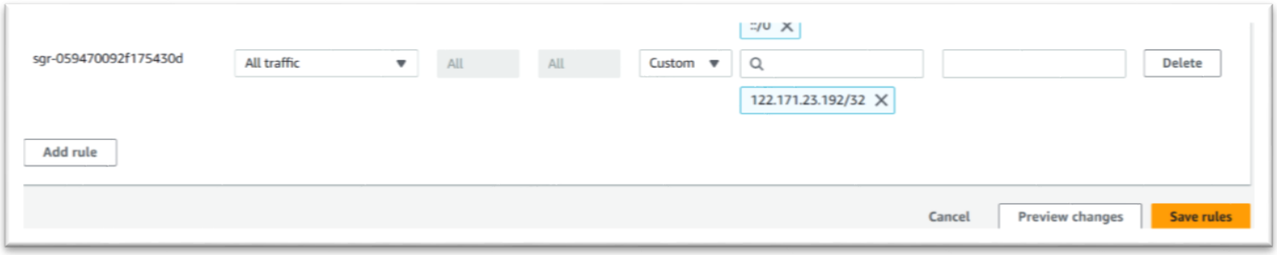
```
ubuntu@ip-172-31-27-203:~$ kubectl get rs
NAME               DESIRED   CURRENT   READY   AGE
nginx-7f456874f4   1         1         1       16m
ubuntu@ip-172-31-27-203:~$ kubectl get svc
NAME            TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
kubernetes      ClusterIP   100.64.0.1      <none>        443/TCP        25m
ngnix-service   NodePort    100.65.176.34   <none>        80:31311/TCP   15m
ubuntu@ip-172-31-27-203:~$ |
```

2. Verify if deployment, pod & Node is running:



```
ubuntu@ip-172-31-27-203: ~
ubuntu@ip-172-31-27-203:~$ kubectl get deployment
NAME    READY   UP-TO-DATE   AVAILABLE   AGE
nginx   1/1     1            1           13m
ubuntu@ip-172-31-27-203:~$ kubectl get pods
NAME                     READY   STATUS    RESTARTS   AGE
nginx-7f456874f4-bv2dd   1/1     Running   0          14m
ubuntu@ip-172-31-27-203:~$ kubectl get nodes
NAME                  STATUS   ROLES           AGE   VERSION
i-037d8ee6998aad3ae   Ready    control-plane   23m   v1.26.5
i-0acf48e8cfe5ca21a   Ready    node            21m   v1.26.5
i-0c19fb386971b005e   Ready    node            21m   v1.26.5
ubuntu@ip-172-31-27-203:~$ |
```

3. Verify if security group has enabled all traffic to MY IP



sgr-059470092f175430d | All traffic | All | All | Custom | ::/0 | 122.171.23.192/32