



---

# KUBERNETES FOR DEVELOPERS

---

Handbook



RAMA MUKKAMALLA

[www.millionvisit.blogspot.com](http://www.millionvisit.blogspot.com)

---

# Table of Contents

1. Kubernetes Architecture and Features .....	1
2. Kubernetes for Local Development .....	3
3. kubectl CLI .....	5
4. Enable kubectl bash autocompletion .....	8
5. Kubernetes Web UI Dashboard .....	10
6. Kubernetes Objects .....	15
7. Imperative vs. Declarative Kubernetes Objects .....	21
8. Kubernetes Object Name, Labels, Selectors and Namespace .....	25
9. Kubernetes Pod Lifecycle .....	30
10. Kubernetes Pod YAML manifest in-detail .....	33
11. Pod Organization using Labels .....	39
12. Effective way of using K8 Liveness Probe .....	42
13. Effective way of using K8 Readiness Probe .....	46
14. Kubernetes Deployment YAML manifest in-detail .....	50
15. Kubernetes Service YAML manifest in-detail .....	54
16. Kubernetes Service Types - ClusterIP, NodePort, LoadBalancer and ExternalName .....	59
17. Expose service using Kubernetes Ingress .....	68
18. Manage app settings using Kubernetes ConfigMap .....	75
19. Manage app credentials using Kubernetes Secrets .....	82

<b>20. Create Automated Tasks using Jobs and CronJobs.....</b>	<b>88</b>
<b>21. Kubernetes Namespace in-detail.....</b>	<b>95</b>
<b>22. Access to Multiple Clusters or Namespaces using kubectl .....</b>	<b>99</b>
<b>23. Kubernetes Volume emptyDir in-detail .....</b>	<b>104</b>
<b>24. Kubernetes Volume hostPath in-detail.....</b>	<b>109</b>
<b>25. PersistentVolume and PersistentVolumeClaim in-detail.....</b>	<b>114</b>
<b>26. Managing Container CPU, Memory Requests and Limits .....</b>	<b>119</b>
<b>27. Configure LimitRange for setting default Memory/CPU for a Pod.....</b>	<b>124</b>

---

# 1. Kubernetes Architecture and Features

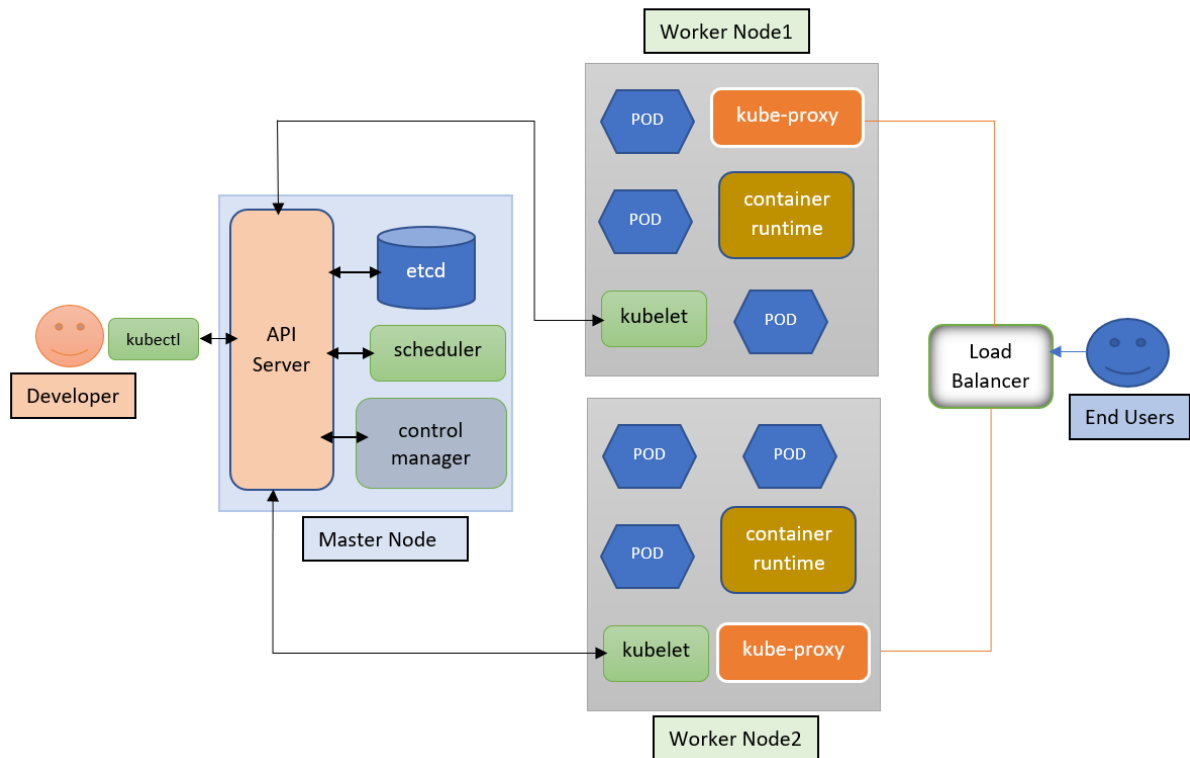
Kubernetes(K8) is an open-source container orchestration tool used for automating deployment, scaling and management of containerized applications.

Features:

- 1. Self-Healing:** It restarts container that failed or kills the container that do not respond to container health check endpoint. It always maintains user defined number of replicas
- 2. Secret and configuration management:** It maintains an application-level configuration and secrets in a separate location. So, it can be modified without re-building and deploying container
- 3. Horizontal scaling:** It is easy to scale-up/down containers with a simple command or automatically based on CPU usage.
- 4. Automatic bin packing:** It automatically places containers into the required Worker nodes based on specified CPU/Memory.
- 5. Storage orchestration:** It allows to mount a local storage or cloud providers
- 6. Automated rollouts and rollbacks:** It allows to rollout new application changes by spinning up new container without killing existing container until proper health check verified. It will rollback automatically if new container does not respond to user defined health check.
- 7. Service discovery and Load balancing:** It does by using Labels and Selectors associated with PODs and Services, and can load-balance across them

## Architecture & Components:

Kubernetes(K8) architecture designed as a cluster. It consists of one master node and one worker node at least. It supports multiple master nodes and worker nodes.



## 1. Master Node:

It is responsible for maintaining entire Kubernetes cluster and there might be more than one master node in the cluster for providing fault-tolerance and high availability.

It has various components like API Server, Control manager, Scheduler and etcd, commonly known as Control plane.

- a. **API Server:** It is the only entry point for the entire cluster and exposes REST endpoints to communicate with cluster.
- b. **etcd:** It is key-value data store used as maintain cluster data like configurations, network activities and etc
- c. **scheduler:** It is responsible for scheduling new POD based on nodes workload. It maintains resource information about each worker node and distribute workload accordingly.
- d. **control manager:** It has different individual processes to maintain cluster stability
  - ❖ **Node controller:** Responsible for noticing and responding when nodes go down.
  - ❖ **Replication controller:** Responsible for maintaining the correct number of pods as per replica specification
  - ❖ **Endpoints controller:** Populates the endpoint objects by using Labels and selectors of pods and services.
  - ❖ **Service Account & Token controller:** Create default accounts and API access tokens for new namespaces

## 2. Worker node:

It is the place where every pod/container run. It consists of several components

- a. **kubelet:** It runs on each worker node and gets the pod specifications from API server and interact with the container runtime to perform start/stop of the container, mount pod volume and secrets.

It monitors state of the pods by using heartbeat messages and transmit data to master node API server

- b. **kube-proxy:** It is a network proxy that runs on each worker node in the cluster, it maintains network rules to allow communication to the Pods from inside or outside of the cluster

It routes traffic to the appropriate pod based on the associated service name and the port number of an incoming request

- c. **Container runtime:** It is the software needs to be installed in each worker node to run the containers i.e., Docker, containerd etc.
- d. **Pod:** It is the place where single or multiple containers run together.

---

## 2. Kubernetes for Local Development

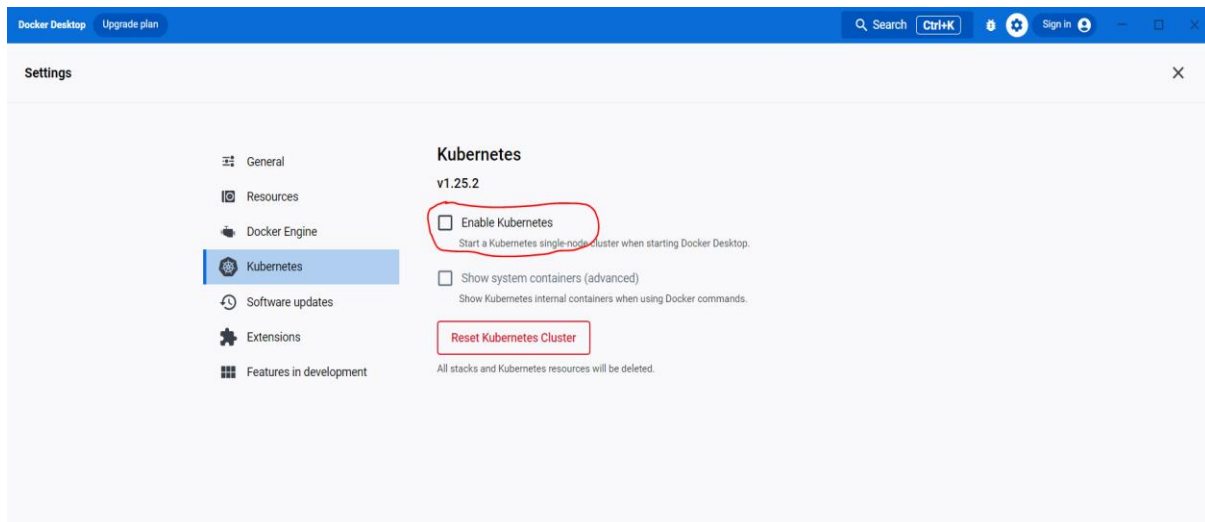
There are multiple ways to run a Kubernetes cluster in the local computer.

### 1. Docker Desktop

It comes with standalone Kubernetes server, client and Docker container runtime.

The Kubernetes server runs locally within your Docker instance as a single node cluster with pre-configured version, kubectl, users and contexts.

To enable Kubernetes in Docker Desktop; Go to Docker Desktop settings, click on Kubernetes menu and check "Enable Kubernetes" checkbox



### Limitations:

- we cannot modify kubernetes version in the Docker Desktop
- There is no out-of-box solution for kubernetes Web UI dashboard
- There is no option to enable add-ons

## 2. Minikube

It is a cross-platform tool works on Linux, mac and windows. It supports different container runtimes like Docker, containerd and CRI-O.

It will start as single node kubernetes cluster with Web UI dashboard and Load Balancer features by issuing **minikube start** command

### Pros:

- It supports multiple clusters
- It supports different Kubernetes versions
- It can be deployed as a VM, a container or on bare metal
- It supports different add-ons like ingress, metric-server and helm

Please visit @ <https://minikube.sigs.k8s.io/> for more details.

## 3) Kind

Kind stands for Kubernetes inside Docker. It was primarily designed for testing kubernetes itself and it supports multi node clusters.

Please visit @ <https://kind.sigs.k8s.io/> for more details.

---

## 3. kubectl CLI

It is a command line interface used for running commands against K8 cluster master node API server. It has a config file called “**kubeconfig**” which maintains cluster information for authentication and connecting to API server.



### Syntax

```
kubectl [command] [TYPE] [NAME] [flags]
```

- a) **Command:** It accepts operation name i.e., create, apply, get, delete, describe, exec, log
- b) **Type:** It accepts resource type in the form of singular, plural or abbreviated. Resource types are case-insensitive

The below three commands give same result

```
> kubectl get pod
> kubectl get pods
> kubectl get po
```

See the following table data with resource type and abbreviated name (i.e., short name)

Name	Short Name	Kind
pods	po	Pod
configmaps	cm	ConfigMap
namespaces	ns	Namespace
nodes	no	Node
replicationcontrollers	rc	ReplicationController
secrets		Secret



services	svc	Service
deployments	deploy	Deployment
replicasets	rs	ReplicaSet
ingresses	ing	Ingress
cronjobs	cj	CronJob

c) **Name:** It is used for specifying name of the resource and it is case-sensitive.

```
> kubectl get pod helloworld
> kubectl get pod helloworld helloworld2
> kubectl get -f ./helloworld.yml
> kubectl get -f ./helloworld.yml -f ./helloworld2.yml
```

### Examples:

```
Display endpoint information about the master and services in the cluster
> kubectl cluster-info

Display both kubectl client version and Kubernetes API server version
> kubectl version

Display cluster configuration settings like contexts, users etc.
> kubectl config view

Display all resources of bindings, short names and KIND
> kubectl api-resources

Display all resources info from default namespace
> kubectl get all

Display all resources info from all namespaces
> kubectl get all --all-namespaces

Create resource from a file
> kubectl create -f ./helloworld.yml

Create resources from multiple files
> kubectl create -f ./helloworld.yml -f ./helloworld2.yml

Create resources from the given directory
> kubectl create -f ./dir

Create/Update resource from a file
> kubectl apply -f ./helloworld.yml

Create/Update resources from multiple files
> kubectl apply -f ./helloworld.yml -f ./helloworld2.yml
```

Create/Update resources from the given directory

```
> kubectl apply -f ./dir
```

Switch to my-namespace

```
> kubectl config set-context --current --namespace=my-namespace
```

List all pods

```
> kubectl get pods
```

List all pods with additional information

```
> kubectl get pods -o wide
```

Get pod yml file

```
> kubectl get pod my-pod -o yml
```

List all services and deployments together

```
> kubectl get services,deployments
```

Display detail state of pod

```
> kubectl describe pods my-pod
```

Delete a pod which specified in the my-pod.yml

```
> kubectl delete -f ./my-pod.yml
```

Delete pods with name pod1 and pod2

```
> kubectl delete pod pod1 pod2
```

Delete pods with label name=mylabel

```
> kubectl delete pods -l name=mylabel
```

Display logs from mypod

```
> kubectl logs mypod
```

Stream logs from mypod

```
> kubectl logs -f mypod
```

Execute command against a container

```
> kubectl exec <pod_name> -c <container_name> -- ls
```

Get interactive shell on a first-container pod

```
> kubectl exec -it <pod_name> -- /bin/sh
```

---

## 4. Enable kubectl bash autocompletion

Kubectl provides autocomplete support for bash which can save us lot of typing.

### Bash on Windows using Git bash

1. Install Git bash from <https://git-scm.com/downloads> if it is not available in your computer
2. Navigate to "C:\Users\<yourname>" from Git bash cli
3. Run following command to get kubectl autocomplete bash commands and save into "kubectl-completion.bash" file

```
> kubectl completion bash > ~/kubectl-completion.bash
```

4. Create or Update .bashrc file source

```
> echo 'source ~/kubectl-completion.bash' >> .bashrc
```

5. Restart Git bash and start typing kubectl get po ( press [TAB] key )



```
/usr/bin/bash --login -i
$ kubectl completion bash > ~/kubectl-completion.bash
$ echo 'source ~/kubectl-completion.bash' >> .bashrc
$ kubectl get pod
poddisruptionbudgets.policy pods          podsecuritypolicies.policy podtemplates
$ kubectl get pod
```

## Bash on Windows using Conemu/Cmdr

It is an another handy multi and split tab windows terminal. It supports multiple command terminals like cmd, PowerShell, bash etc.

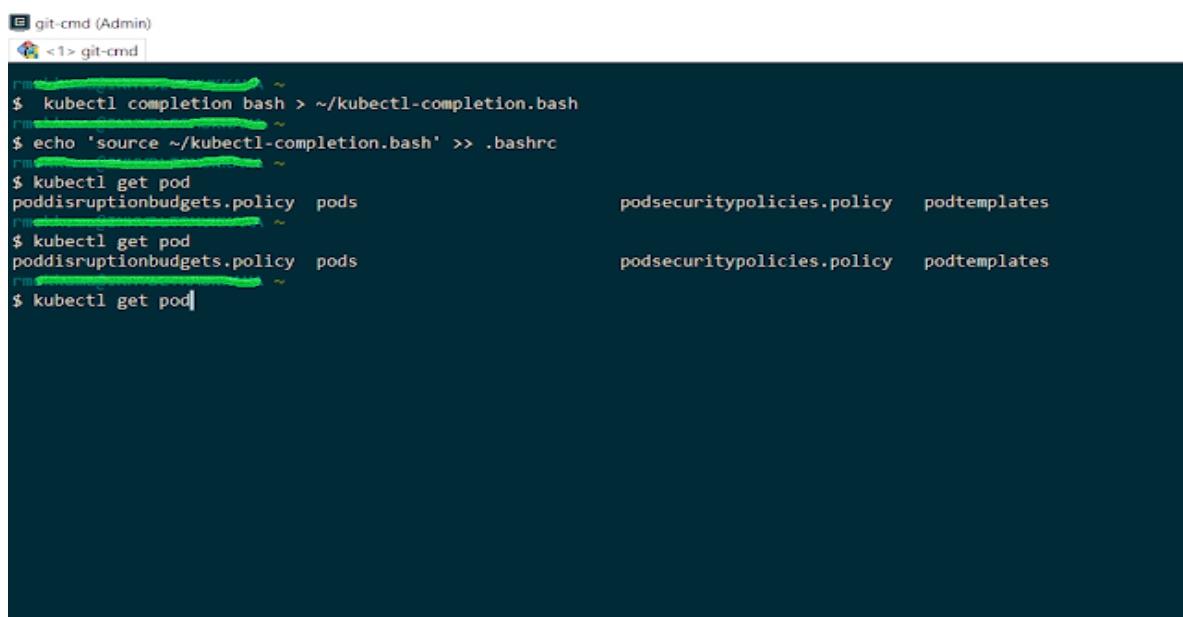
1. Install Git bash from <https://git-scm.com/downloads> if it is not available in your computer
2. Install Conemu from <https://conemu.github.io/> if it is not available in your computer.
3. Open new Git Bash console from Conemu terminal
4. Navigate to "C:\Users\<yourname>"
5. Run following command to get kubectl autocompletion bash commands and save into "kubectl-completion.bash" file

```
> kubectl completion bash > ~/kubectl-completion.bash
```

6. Create or Update .bashrc file source

```
> echo 'source ~/kubectl-completion.bash' >> .bashrc
```

7. Restart Conemu terminal and start typing kubectl get po ( press [TAB] key )



```
git-cmd (Admin)
<1> git-cmd
$ kubectl completion bash > ~/.kubectl-completion.bash
$ echo 'source ~/.kubectl-completion.bash' >> .bashrc
$ kubectl get pod
poddisruptionbudgets.policy pods podsecuritypolicies.policy podtemplates
$ kubectl get pod
poddisruptionbudgets.policy pods podsecuritypolicies.policy podtemplates
$ kubectl get pod
poddisruptionbudgets.policy pods podsecuritypolicies.policy podtemplates
```

## Bash on Linux/macOS

Please go through below link to enable kubectl autocompletion on Linux/macOS

<https://kubernetes.io/docs/tasks/tools/install-kubectl/#enabling-shell-autocompletion>

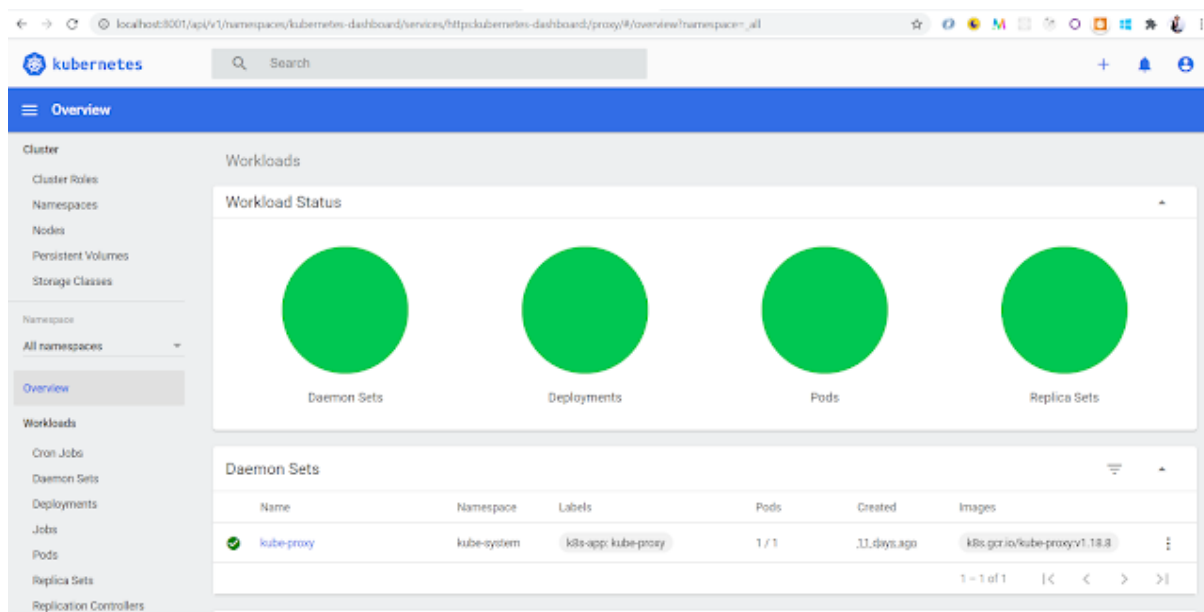
---

## 5. Kubernetes Web UI Dashboard

It is a web-based UI for viewing entire Kubernetes cluster information like Nodes, Deployments, Pods, Services, Jobs, Secrets etc.

It is used to deploy containers into the Kubernetes cluster and troubleshoot the containers in the K8 cluster.

It is used to create or modify individual Kubernetes resources like Deployments, Pods, Services, Jobs etc.



## Enable Web UI Dashboard using Kubectl:

Make sure Kubernetes cluster up and running before executing following kubectl commands

a) run the following command to enable Dashboard

```
> kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.0.0/aio/deploy/recommended.yaml
```

Alternately, you can download same .yaml file into local computer, save as **kubernetes-dashboard.yaml** and run the following command

```
> kubectl apply -f ./kubernetes-dashboard.yaml
```

The above command creates dashboard deployment, service, service account, roles, role binding and secret.

```
$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.0.0/aio/deploy/recommended.yaml
namespace/kubernetes-dashboard created
serviceaccount/kubernetes-dashboard created
service/kubernetes-dashboard created
secret/kubernetes-dashboard-certs created
secret/kubernetes-dashboard-csrf created
secret/kubernetes-dashboard-key-holder created
configmap/kubernetes-dashboard-settings created
role.rbac.authorization.k8s.io/kubernetes-dashboard created
clusterrole.rbac.authorization.k8s.io/kubernetes-dashboard created
rolebinding.rbac.authorization.k8s.io/kubernetes-dashboard created
clusterrolebinding.rbac.authorization.k8s.io/kubernetes-dashboard created
deployment.apps/kubernetes-dashboard created
service/dashboard-metrics-scraper created
deployment.apps/dashboard-metrics-scraper created
```

b) run the following command for enabling proxy between local computer and Kubernetes Apiserver.

```
> Kubectl proxy
```

Go to the browser and hit the following url to view Web UI Dashboard login page

<http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/>

### Kubernetes Dashboard

☒ Token

Every Service Account has a Secret with valid Bearer Token that can be used to log in to Dashboard. To find out more about how to configure and use Bearer Tokens, please refer to the [Authentication](#) section.

☐ Kubeconfig

Please select the kubeconfig file that you have created to configure access to the cluster. To find out more about how to configure and use kubeconfig file, please refer to the [Configure Access to Multiple Clusters](#) section.

Enter token \*

Sign in

**c) run the following command to get valid bearer token for login**

```
> kubectl -n kubernetes-dashboard describe secret $(kubectl -n kubernetes-  
dashboard get secret | grep admin-user | awk '{print $1}')
```

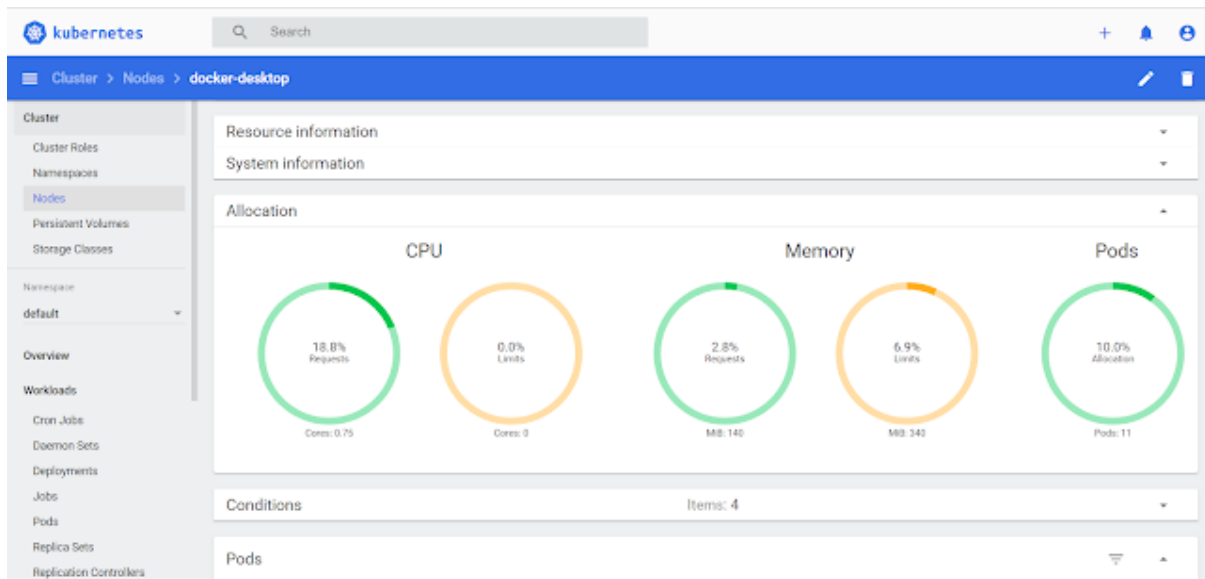
[illegible]

copy the bearer token and enter it into the Web UI Dashboard login page

**Kubernetes dashboard has four main sections i.e.**

### 1. Cluster:

It shows information about Nodes, Namespaces, Persistent Volumes, Roles and Storage Classes. Node list view contains CPU and memory usage metrics aggregated across all Nodes. On click of each node, it shows node status, allocated resources, events and pods running on the node.



## 2. Workloads:

It shows all applications running in the selected namespace including Deployments, Replica sets, Pods, Jobs, Daemon Sets, etc.

The screenshot shows the Kubernetes dashboard for the 'kube-system' namespace. The left sidebar contains navigation links for Nodes, Persistent Volumes, Storage Classes, Namespace (kube-system), Overview, Workloads, Cron Jobs, Daemon Sets, Deployments, Jobs, Pods, Replica Sets, Replication Controllers, Stateful Sets, Discovery and Load Balancing, and Ingresses. The main content area shows a list of pods. The table has columns: Name, Labels, Node, Status, Restarts, CPU Usage (cores), Memory Usage (bytes), and Created. The pods listed are:

Name	Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Usage (bytes)	Created
coredns-66bff467f8-7mcrw	k8s-app: kube-dns pod-template-hash: 66bff467f8	docker-desktop	Running	3	-	-	12 days ago
coredns-66bff467f8-99nfb	k8s-app: kube-dns pod-template-hash: 66bff467f8	docker-desktop	Running	2	-	-	12 days ago
etcd-docker-desktop	component: etcd tier: control-plane	docker-desktop	Running	2	-	-	12 days ago
kube-apiserver-docker-desktop	component: kube-apiserver tier: control-plane	docker-desktop	Running	2	-	-	12 days ago
kube-controller-manager-docker-desktop	component: kube-controller-manager tier: control-plane	docker-desktop	Running	2	-	-	12 days ago
kube-proxy-mstx	controller-revision-hash: 5cf956ffcf k8s-app: kube-proxy	docker-desktop	Running	2	-	-	12 days ago

## 3. Discovery and Load Balancing:

It shows information about services which exposed to external world and internal endpoints within a cluster.



<div> <div>kubernetes</div> <div>Search</div> <div>+</div> <div></div> <div></div> </div>						
Discovery and Load Balancing > Services						
<div> <div>Overview</div> <div>Workloads</div> <div>Cron Jobs</div> <div>Daemon Sets</div> <div>Deployments</div> <div>Jobs</div> <div>Pods</div> <div>Replica Sets</div> <div>Replication Controllers</div> <div>Stateful Sets</div> <div>Discovery and Load Balancing</div> <div>Ingresses</div> <div>Services</div> <div>Config and Storage</div> <div>Config Maps</div> <div>Persistent Volume Claims</div> <div>Secrets</div> </div>						
Services						
Name	Namespace	Labels	Cluster IP	Internal Endpoints	External Endpoints	Created
✓ <a href="#">kubernetes</a>	default	component: apiserver provider: kubernetes	10.96.0.1	kubernetes:443 TCP kubernetes:0 TCP	-	12 days ago
✓ <a href="#">kube-dns</a>	kube-system	k8s-app: kube-dns kubernetes.io/cluster-service: true <a href="#">Show all</a>	10.96.0.10	kube-dns.kube-system:53 UDP kube-dns.kube-system:53 UDP kube-dns.kube-system:53 TCP kube-dns.kube-system:9153 TCP kube-dns.kube-system:0 TCP	-	12 days ago
✓ <a href="#">dashboard-metrics-scraper</a>	kubernetes-dashboard	k8s-app: dashboard-metrics-scraper	10.101.205.100	dashboard-metrics-scraper.kubernetes-dashboard:8000 TCP dashboard-metrics-scraper.kubernetes-dashboard:0 TCP	-	4 hours ago
✓ <a href="#">kubernetes-dashboard</a>	kubernetes-dashboard	k8s-app: kubernetes-dashboard	10.96.82.175	kubernetes-dashboard.kubernetes-dashboard:443 TCP kubernetes-	-	4 hours ago

#### 4. Config and Storage:

It shows information about configurations and secrets which is used for the containers.

<div> <div>kubernetes</div> <div>Search</div> <div>+</div> <div></div> <div></div> </div>						
Config and Storage > Secrets						
<div> <div>Overview</div> <div>Workloads</div> <div>Cron Jobs</div> <div>Daemon Sets</div> <div>Deployments</div> <div>Jobs</div> <div>Pods</div> <div>Replica Sets</div> <div>Replication Controllers</div> <div>Stateful Sets</div> <div>Discovery and Load Balancing</div> <div>Ingresses</div> <div>Services</div> <div>Config and Storage</div> <div>Config Maps</div> <div>Persistent Volume Claims</div> <div>Secrets</div> <div>Custom Resource Definitions</div> <div>Settings</div> </div>						
Secrets						
Name	Namespace	Labels	Type	Created		
<a href="#">default-token-8tgcx</a>	default	-	kubernetes.io/service-account-token	12 days ago		
<a href="#">default-token-ptq66</a>	kube-node-lease	-	kubernetes.io/service-account-token	12 days ago		
<a href="#">default-token-f62vh</a>	kube-public	-	kubernetes.io/service-account-token	12 days ago		
<a href="#">attachdetach-controller-token-k9hlp</a>	kube-system	-	kubernetes.io/service-account-token	12 days ago		
<a href="#">bootstrap-signer-token-cp889</a>	kube-system	-	kubernetes.io/service-account-token	12 days ago		
<a href="#">certificate-controller-token-z9wvf</a>	kube-system	-	kubernetes.io/service-account-token	12 days ago		
<a href="#">clusterrole-aggregation-controller-token-q7dv8</a>	kube-system	-	kubernetes.io/service-account-token	12 days ago		
<a href="#">coredns-token-4cz2g</a>	kube-system	-	kubernetes.io/service-account-token	12 days ago		
<a href="#">cronjob-controller-token-7y5kh</a>	kube-system	-	kubernetes.io/service-account-token	12 days ago		
<a href="#">daemon-set-controller-token-723wg</a>	kube-system	-	kubernetes.io/service-account-token	12 days ago		

---

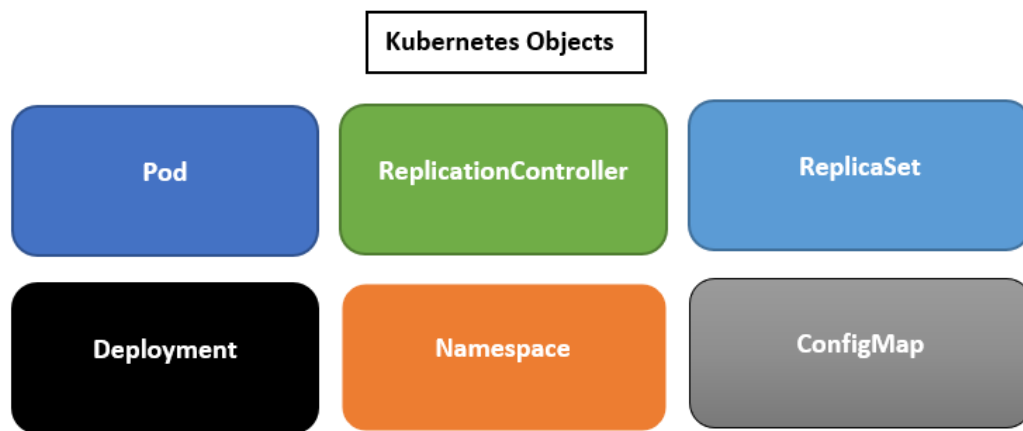
## 6. Kubernetes Objects

Kubernetes objects are persistent entities and used to represent the state of the cluster.

It will be created by using imperative command on live objects or declaratively from a file.

**kubectl** command line interface used to send Kubernetes objects to API server and make necessary actions in the cluster.

In general, we specify Kubernetes objects in the .yml file and send to kubectl CLI. Kubectl converts the information to JSON while interacting with API server



**Syntax:**

```
apiVersion: v1    # v1, apps/v1, and extensions/v1beta1
kind: Pod         # Pod, Deployment, Service etc.
metadata:
  name: nginx-pod
spec:
```

- **apiVersion** - version of the Kubernetes API you are using to create an object
- **kind** - kind of object you want to create
- **metadata** - It helps to uniquely identify the object, including a name, labels and optional namespace
- **spec** – It is used to define desired state for the object

### 1. Pod

A pod is the most basic unit of the Kubernetes cluster. It usually contains one or more running containers. Pods are designed to be ephemeral in nature which means that they can be destroyed at any time. Containers in a pod share the same network and storage

```

apiVersion: v1
kind: Pod
metadata:
  name: first-pod
  labels:
    name: first-pod
spec:
  containers:
    - name: first-pod
      image: hello-world
      ports:
        - containerPort: 8080

```

**containers:** It contains,

- **name:** The name of the container that you'll run in your pod.
- **image:** The image of the application you want to run in your pods.
- **containerPort:** It is the port of your application container is listening to.

## 2. ReplicationController

It is used to create multiple instances of same pod in the cluster node. It ensures that at any given time, the desired number of pods specified are in the running state. If a pod stops or dies, the ReplicationController creates another one to replace it.

```

apiVersion: v1
kind: ReplicationController
metadata:
  name: myapp
spec:
  replicas: 2
  selector:
    app: myapp
  template:
    metadata:
      name: first-pod
      labels:
        name: first-pod
    spec:
      containers:
        - name: first-pod
          image: hello-world
          ports:
            - containerPort: 8080

```

Definition says **replicas:2** it means that any given time two pods must be running in the cluster. If any pod fails, replication controller creates new pod immediately.

The template section provides characteristics of the pod. It is like Pod definition.

### 3. Replicaset

It is the next generation of ReplicationController

Replicaset	ReplicationController
It uses matchLabels specified under selection and works as set-based selector	It uses labels selected under selector section and works as equality-based selector
Ex: env=prod/qa	Ex: env=prod
It selects all the objects where key=env irrespective of the value	It selects all objects where key=env and value=prod
<b>selector</b> attribute is mandatory	<b>selector</b> attribute is not required
It uses rollout technique and will be used internally by Deployment objects	It uses rolling update technique. It means, each pod template changes one at a time until all the pods are updated.
It is not meant to be created on their own, it creates automatically when deployment objects are created	It is used to create on their own
It belongs to apiVersion: apps/v1	It belongs to apiVersion: v1

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
        - name: php-redis
          image: gcr.io/google_samples/gb-frontend:v3
```

## 4. Deployment

It encapsulates both Replicaset and Pod to provide declarative method for defining a resource.

It is used for managing pods and internally uses Replicaset for creating number of pods.

It can be used to scale your application by increasing the number of running pods, or update the running application

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
            - containerPort: 80
```

## 5. Namespace

It is used for grouping Kubernetes objects in the cluster and actions will be performed against the namespace.

Example: create namespace for each environment or Team

```
apiVersion: v1
kind: Namespace
metadata:
  name: production
-- --

apiVersion: v1
kind: Namespace
```

```
metadata:
  name: staging
```

namespace will be referred in the metadata section of Kubernetes object.

```
apiVersion: v1
kind: Pod
metadata:
  name: first-pod
  namespace: staging
  labels:
    name: first-pod
    app: hello-world-app
spec:
```

## 6. ConfigMap

It is used for creating configurations for the container and will be changed dynamically while containers are running

There are 3 ways to create configMap

### a) from a directory:

- Create a directory and name it **my\_config**
- Inside the directory, create two files: name1.txt and name2.txt
- In the name1.txt file, add the following lines:

```
name: Rama
gender: male
city: abcd
```

- In the name2.txt file add the following lines:

```
name: Krishna
gender: male
city: xyz
```

execute the following command to create the directory

```
> kubectl create configmap my-config --from-file=my_config/
```

### b) from files:

The approach is the same as the one for creating ConfigMap from a directory. But in this case, we specify the file path instead of a folder

```
> kubectl create configmap my-config --from-file=my_config/name1.txt
```

### c) from literals:

```
> kubectl create configmap my-config --from-literal=name=rama
--from-literal=gender=male
```

We can consume ConfigMaps in the pod by specifying "ConfigMapRef"

```
apiVersion: v1
kind: Pod
metadata:
  name: api-test-pod
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      command: ["/bin/sh", "-c", "env"]
      envFrom:
        - configMapRef:
            name: my-config
```

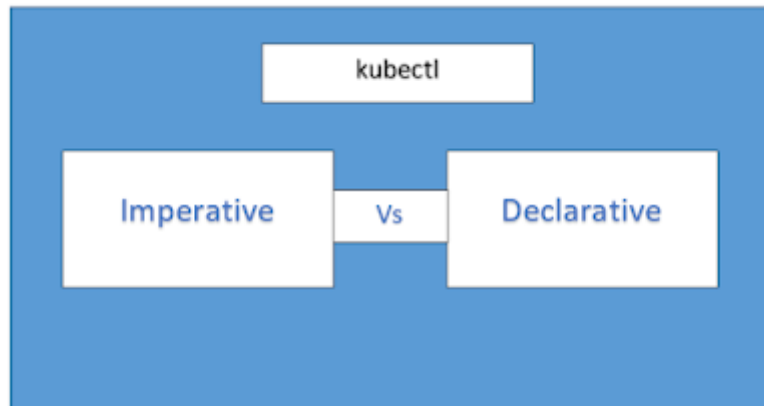
use "configMapKeyRef" for specifying selected key

```
apiVersion: v1
kind: Pod
metadata:
  name: api-test-pod
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      command: ["/bin/sh", "-c", "env"]
      env:
        - name: test
          value: test123
        - name: RAMA_CITY
          valueFrom:
            configMapKeyRef:
              name: my-config
              key: city
```

---

## 7. Imperative vs. Declarative Kubernetes Objects

kubectl CLI supports both imperative and declarative ways to create and manage Kubernetes objects like Pod, Deployment, Service, namespace etc.



### Imperative way:

In this approach, we operate directly on live objects in a cluster using kubectl CLI as arguments or flags.

It follows verb-driven commands to create or manage Kubernetes objects.

### Syntax:

Used for creating new Kubernetes object

`Kubectl create <objecttype> [<subtype>] <name>`

Used for creating new Pod to run a container

`Kubectl run`

Used for creating new service object to load balance traffic across multiple Pods

`Kubectl expose:`

Used for creating new Deployment for auto scaling Pods

`Kubctl autoscale:`

Used for add/remove Pods by updating replica count

`Kubectl scale:`



Used for add/removing a label from an object

Kubectl label:

Used for editing configuration in an editor

Kubectl edit:

Used for deleting an object from a cluster

Kubectl delete:

Used for getting basic information about an object

Kubectl get:

Used for getting detailed information about an object

Kubectl describe:

Used for getting stdout and stderr for a container running in a Pod

Kubectl logs:

#### Example:

```
> kubectl create ns testnamespace
> kubectl create deployment nginx --image nginx
> kubectl run my-helloworld --image helloworld:1.0
> kubectl get deployment my-helloworld
> kubectl describe deployment my-helloworld
> kubectl label deployment my-helloworld foo=bar
> kubectl expose deployment my-helloworld --port 80 --target-port 3000
> kubectl logs deployment/my-helloworld
> kubectl scale deployment my-helloworld --replicas=2
> kubectl delete deployment/nginx
```

for more info visit @ <http://millionvisit.blogspot.com/2020/12/kubernetes-for-developers-3-kubectl-cli.html>

#### Advantages:

- Commands are simple, easy to learn and remember.
- Commands require only a single step to make changes to the cluster.

## Disadvantages

- Commands cannot be committed to source control system and difficult to review.
- Commands do not provide an audit trail associated with changes.
- Commands do not provide a template for creating new objects.

## Declarative way:

In this approach, we create YAML or JSON files for creating, updating, or deleting Kubernetes objects in a cluster using kubectl CLI.

### Syntax:

Used for creating an object from a specified file

```
kubectl create -f <filename|url>
```

Used for updating a live object from a specified file

```
kubectl replace -f <filename|url>
```

Used for deleting an object from a specified file

```
kubectl delete -f <filename|url>
```

Used for viewing info about an object from a specified file

```
kubectl get -f <filename|url> -o yaml
```

Used for creating/Updating an object from a specified file

```
kubectl apply -f <filename|url>
```

Used for creating/Updating all objects specified in the directory

```
kubectl apply -f <directory>
```

### Example:

```
> kubectl create -f nginx.yaml
> kubectl delete -f nginx.yaml -f redis.yaml
> kubectl replace -f nginx.yaml
> kubectl get -f nginx.yaml -o yaml
```

for more info visit @ <http://millionvisit.blogspot.com/2020/12/kubernetes-for-developers-3-kubectl-cli.html>

### Advantages:

- Object configurations can be stored in a source control system for reviewing changes before push.
- Object configuration provides a template for creating new objects.

### Disadvantages

- Object configuration requires basic understanding of the object schema.
- Object configuration requires the additional step of writing a YAML file.

### Convert Imperative to Declarative way:

Option A:

- a. Export the live object to a local object configuration file:

```
kubectl get {<kind>/<name>} --export -o yaml > {<kind>_<name>}.yaml
```

Ex:

```
> kubectl get deployments testnginx --export -o yaml > testnginx2.yaml
```

Option B:

- a. Export the live object to a local object configuration file:

```
kubectl get {<kind>/<name>} -o yaml > {<kind>_<name>}.yaml
```

```
> kubectl get deployments testnginx -o yaml > testnginx2.yaml
```

- b. Remove the status field from the configuration file
- c. use replace command to execute the file

```
kubectl replace -f <kind>_<name>.yaml
```

```
> kubectl replace -f testnginx2.yaml
```

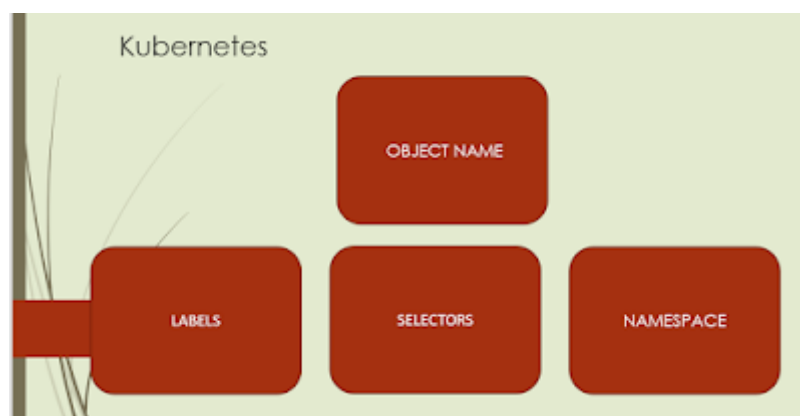
---

## 8. Kubernetes Object Name, Labels, Selectors and Namespace

### 1. Object Name

Every Kubernetes object has a **Name** that is unique for that type of resource within the same namespace. It means, we can only have one Pod name called “**mynginx**” within the same namespace. However, we can use the same name for another Kubernetes object like Deployment, Service, etc.

So, we can set the same name i.e., “**mynginx**” to Pod, Deployment, Service, etc.



The following characteristics need to be followed while setting name to Kubernetes object.

- Name should not exceed 253 characters
- It contains only lowercase alphanumeric characters (a to z), hyphen (-), period(.)
- It should start with an alphanumeric character
- It should end with an alphanumeric character

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

### 2. Labels

Label is a key-value pair which is attached to pods, deployments, etc.

- It acts as an identifier on K8 object (Ex: Pod or Deployment). So, the other Kubernetes objects (Ex: Service, DaemonSet) can communicate by matching the same label names.

- It can be attached to Kubernetes objects at creation time or directly on live objects by using Imperative way
- Same label key/value can be assigned to multiple Kubernetes objects
- Each object in the label can have set of key/value and each key must be unique for a given object
- Label key should not exceed 63 characters and allowed characters are alphanumeric, dash (-), underscore (\_), dot(.)
- Label value should not exceed 63 characters and allowed characters are alphanumeric, dash (-), underscore (\_), dot(.)

```
metadata:
  name: pod-label-demo
  labels:
    environment: production
    app: nginx
```

#### Example: Creating labels on Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-label-demo
  labels:
    environment: production
    app: nginx
spec:
  containers:
    - name: nginx
      image: nginx:1.14.2
      ports:
        - containerPort: 80
```

There are two labels i.e. **app:nginx**, **environment:production** are assigned to Pod

#### Example: Creating labels on Deployment

```
apiVersion: v1
kind: Deployment
metadata:
  name: deployment-label-demo
  labels:
    environment: production
    app: nginx
spec:
```

There are two labels i.e. **app:nginx**, **environment:production** are assigned to Deployment

Use the following kubectl commands to find each k8 object and respective labels

```
Show all labels on each Pod
> kubectl get pods --show-labels

Show all labels on each Deployment
> kubectl get deployments --show-labels

Show all Pods where label app:nginx
> kubectl get pods -l app=nginx

Add label to running Pod
// syntax
kubectl label Pod <podname> <key>=<value>
// example
> kubectl label Pod pod-label-demo tier=frontend

// delete Pods using label
> kubectl delete pod -l app=nginx
```

### 3. Selector

It is used for grouping Kubernetes objects and perform actions accordingly.

- It is used by Kubernetes Deployment object to talk to all Pods with particular label key/value
- It is used by Kubernetes Service object to expose all Pods with particular label key/value

**Example:** Adding labels on Pod template and Selector tag for identifying all Pod labels in the Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
```

```
containers:
  - name: nginx
    image: nginx:1.7.9
    ports:
      - containerPort: 80
```

Here, we have added **selector** tag with `matchLabels(app:nginx)` used by deployment for identifying all the Pods where label equal to `app:nginx` and apply Deployment rules

**Example:** Create Service and communicate with Pods using labels

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx-service
spec:
  type: LoadBalancer
  ports:
    - port: 80
  selector:
    app: nginx
```

Here, we have added **selector** tag used by Service for identifying all the Pods where label name equal to `app:nginx` and apply Service rules

**Note:** `matchLabels` are not supported by Service. It only supported by Deployment, Replica Set, Daemon Set and Job.

## 4. Namespace

Kubernetes supports multiple virtual clusters by using Namespaces. It helps when multiple teams using same cluster and create separate Roles, Binding and Environments for each team.

- Kubernetes resource name should be unique within a namespace, but not across namespaces
- Namespaces cannot be nested
- Each Kubernetes resource can only be in one namespace
- Don't create namespace with prefix "kube- ". Because it is reserved for K8 system
- By default, each resource created under "default" namespace

```
// create namespace using Imperative way
> kubectl create namespace my-namespace
```

```
# create namespace using Declarative way
apiVersion: v1
kind: Namespace
metadata:
  name: my-namespace
```

```
> kubectl create -f my-namespace.yaml
```

```
# create Pod under my-namespace
apiVersion: v1
kind: Pod
metadata:
  name: my-nginx-pod
  namespace: my-namespace
```

```
// get all namespaces
> kubectl get namespace
```

```
// get all Pods with specific namespace
> kubectl get pods -namespace=my-namespace
```

```
// delete namespace
> kubectl delete namespace my-namespace
```

```
// Setting namespace preference for kubectl subsequent commands in the
current session
> kubectl config set-context --current --namespace=my-name
```



---

## 9. Kubernetes Pod Lifecycle

A Pod is a group of one or more containers (consider docker containers) with shared storage and network resources.

- **Network:** Pods get unique IP address automatically and all the containers in the Pod communicate each other using **localhost** and **Port**
- **Storage:** Pods can be attached to Volumes that can be shared among the containers

A Pod is designed to run a single instance of an application inside node of Kubernetes cluster.

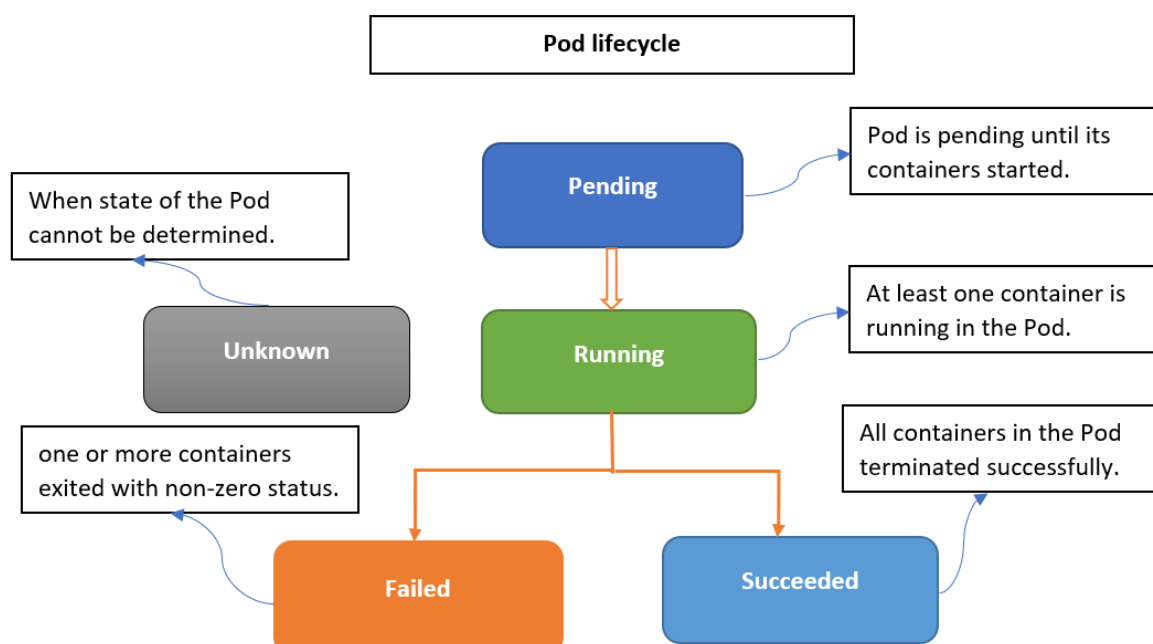
### Types of Pod:

- **Single container Pod:** The “one-container-per-pod” is the most common use case and Kubernetes manage Pod rather than container directly.
- **Multi container Pod:** A Pod can group multiple containers with shared storage volumes and network resources. Generally, we name it as Primary container and Sidecar container.

**Ex:** Primary container used to serve data stored in the filesystem/volume to outer world whereas Sidecar container used to refresh the filesystem/volume.

### Pod lifecycle:

Pods are ephemeral and not designed to run forever, when Pod is terminated it cannot be repair themselves rather it gets deleted and recreated based on Pod policy.



Pod lifecycle starts from “**Pending**” phase, moving through “**Running**” if at least one container starts and then will move to “**Succeeded**” or “**Failed**” phase depending on the container exit status.

Pod phases are continuing to update when,

- Kubelet constantly monitor container states and send info back to KubeAPI Server for updating Pod phase.
- Kubelet stops reporting to the KubeAPI Server.

Pod phases:

<b>Pending</b>	Pod has been created by the cluster, but one or more of its containers are not yet running. This phase includes time spent being scheduled on a node and downloading images
<b>Running</b>	The Pod has been allotted to a node; all the containers have been created. At least one container is still running, or is in the process of starting or restarting
<b>Succeeded</b>	All containers in the Pod have terminated successfully
<b>Failed</b>	One or more containers terminated with non-zero status
<b>Unknown</b>	The state of the Pod cannot be determined. This occurs due to error while communicating with the node

### Container States:

The way Kubernetes maintain Pod phases, it maintains state of each container in the Pod.

Once the scheduler assigns a Pod to a Node, the kubelet starts creating containers for that Pod using a container runtime. There are 3 possible states for the container.

<b>Waiting</b>	When the container still pulling image, applying Secret data etc.
<b>Running</b>	When the container executing without any issues
<b>Terminated</b>	When the container exited with non-zero status

```
// Create Pod using imperative way
> kubectl run pod-nginx --image=nginx
```

```
# Create Pod using Declarative way (.yaml file )
apiVersion: v1
kind: Pod
metadata:
  name: pod-nginx
spec:
  containers:
```

```
- name: nginx
  image: nginx:1.14.2
  ports:
    - containerPort: 80
      protocol: TCP
```

```
// Create Pod using Declarative way
> kubectl apply -f ./pod-nginx.yaml
```

```
// View all running pods and their status
> kubectl get po

// View specific pod running status
// syntax: kubectl get po <pod-name>
> kubectl get po pod-nginx

// Get running Pod definition
// syntax: kubectl get po <pod-name> -o <outout-format>
> kubectl get po pod-nginx -o yaml

// Get Pod phase
> kubectl get po pod-nginx -o yaml | grep phase

// Describe Pod in-detail
// syntax: kubectl describe po <pod-name>
> kubectl describe po pod-nginx

// View logs from Pod running container
// syntax: kubectl logs <pod-name>
> kubectl logs pod-nginx

// View logs from Pod when multiple running containers
// syntax: kubectl logs <pod-name> -c <container-name>
> kubectl logs pod-nginx -c nginx

// Stream logs from Pod
> kubectl logs -f pod-nginx

// Expose Pod for debugging or testing purpose using port-forward proxy
// Syntax: kubectl port-forward <pod-name> <host-port>:<container-port>
// Ex: http://localhost:8444
> kubectl port-forward pod-nginx 8444:80
```

```
// Shell to a running Pod
> kubectl exec -it pod-nginx --sh

// Shell to specific container when multiple containers running in the Pod
// Syntax: kubectl exec -it <pod-name> --container <container-name> -- sh
> kubectl exec -it pod-nginx --container nginx --sh

// View last 100 messages from Pod
> kubectl logs --tail=100 pod-nginx

// Delete a Pod
> kubectl delete po pod-nginx
```

---

## 10. Kubernetes Pod YAML manifest in-detail

In General, K8 resources are usually created using YAML or JSON manifest.

### YAML Structure:

YAML is an Indentation based and superset of JSON, which means any valid JSON file is also a valid YAML file. There are two types of structures we mostly use.

- **Map:** It is key-value pair like Json Object.

**Example:** simple key-value structure where key and value are strings

#yaml

```
apiVersion: v1
kind: Pod
```

#json

```
{
  "apiVersion": "v1",
  "kind": "Pod"
}
```

**Example:** complex key-value structure where value act as another Map

#yaml

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

#json

```
{
  "apiVersion": "v1",
  "kind": "Pod",
  "metadata": {
    "name": "pod-nginx"
  }
}
```

- **List:** It is sequence of values like Json Array represented using dash (-) sign.

**Example:** defining sequence of values

#yaml

```
args:
- "HOSTNAME"
- "PORT_NUMBER"
- "9900"
```

#json

```
args: [
  "HOSTNAME",
  "PORT_NUMBER",
  "9900"
]
```

**Example:** Defining List item as Map like Json array of Objects.

#yaml

```
containers:
- name: nginx-container
  image: nginx
  volumeMounts:
    - name: shared-data
```

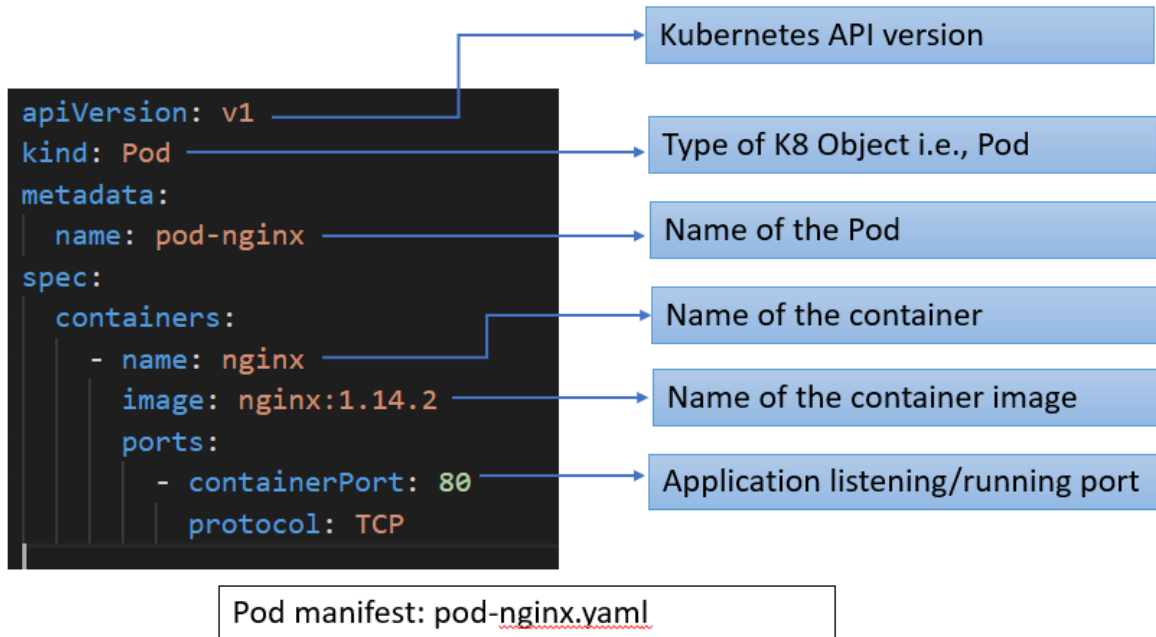
```
    mountPath: /usr/share/nginx/html
- name: debian-container
  image: debian
  volumeMounts:
    - name: shared-data
      mountPath: /pod-data
```

#json

```
{
  "containers": [
    {
      "name": "nginx-container",
      "image": "nginx",
      "volumeMounts": [
        {
          "name": "shared-data",
          "mountPath": "/usr/share/nginx/html"
        }
      ]
    },
    {
      "name": "debian-container",
      "image": "debian",
      "volumeMounts": [
        {
          "name": "shared-data",
          "mountPath": "/pod-data"
        }
      ]
    }
  ]
}
```

### Pod Manifest:

A pod is the most basic unit of the Kubernetes cluster. It usually contains one or more running containers. Pods are designed to be ephemeral in nature which means that they can be destroyed at any time. Containers in a pod share the same network and storage.



### Syntax:

```
apiVersion: v1    # v1, apps/v1, and extensions/v1beta1
kind: Pod        # Pod, Deployment, Service etc.
metadata:
  name: nginx-pod
spec:
```

- **apiVersion** - version of the Kubernetes API you are using to create an object
- **kind** - kind of object you want to create
- **metadata** - It helps to uniquely identify the object, including a name, labels and optional namespace
- **spec** – It is used to define desired state for the object

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

### Example: Creating labels on Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-label-demo
  labels:
    environment: production
    app: nginx
spec:
  containers:
    - name: nginx
      image: nginx:1.14.2
      ports:
        - containerPort: 80
```

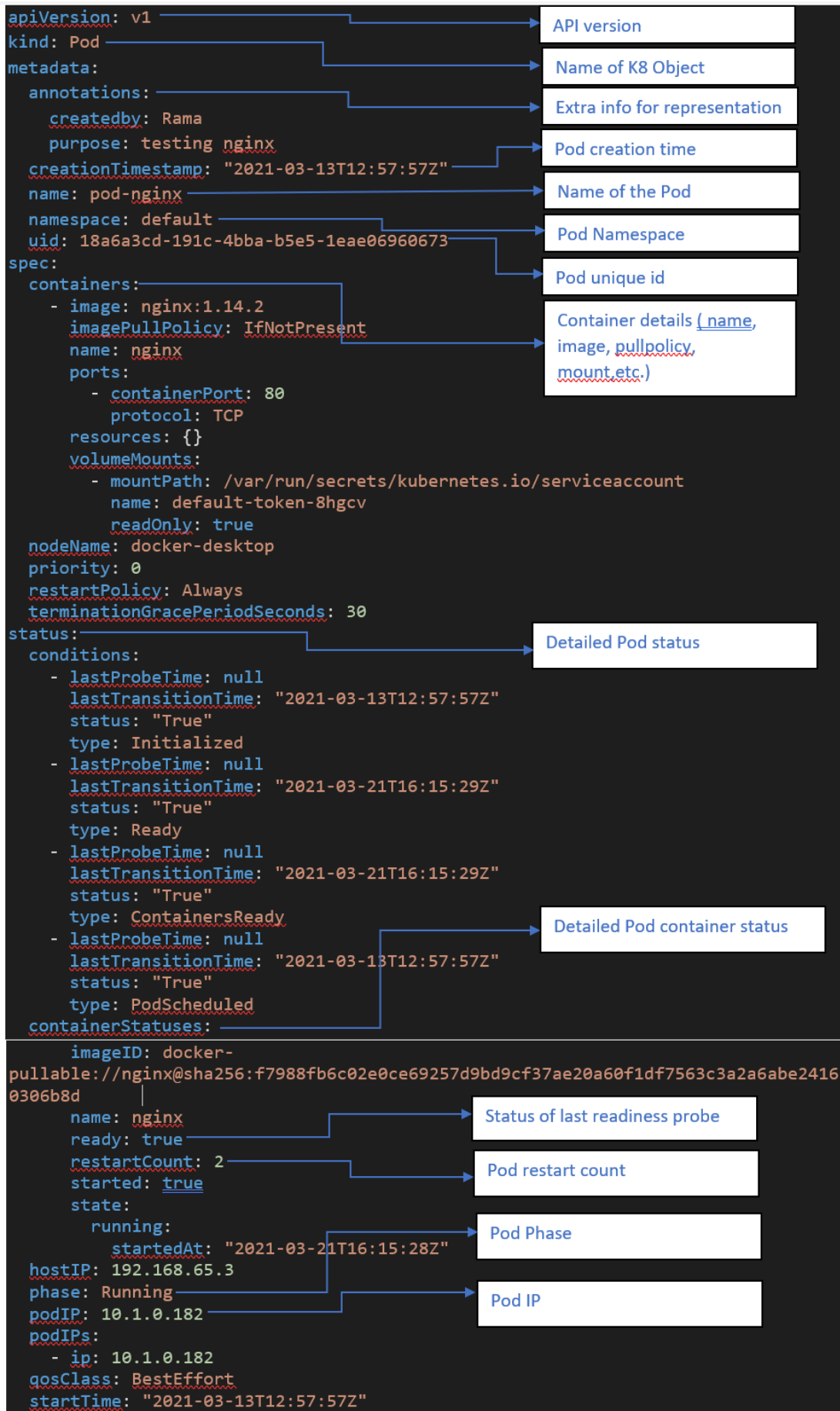
There are two labels i.e. **app:nginx**, **environment:production** are assigned to Pod

### Running Pod Manifest:

execute following command to get running pod manifest details

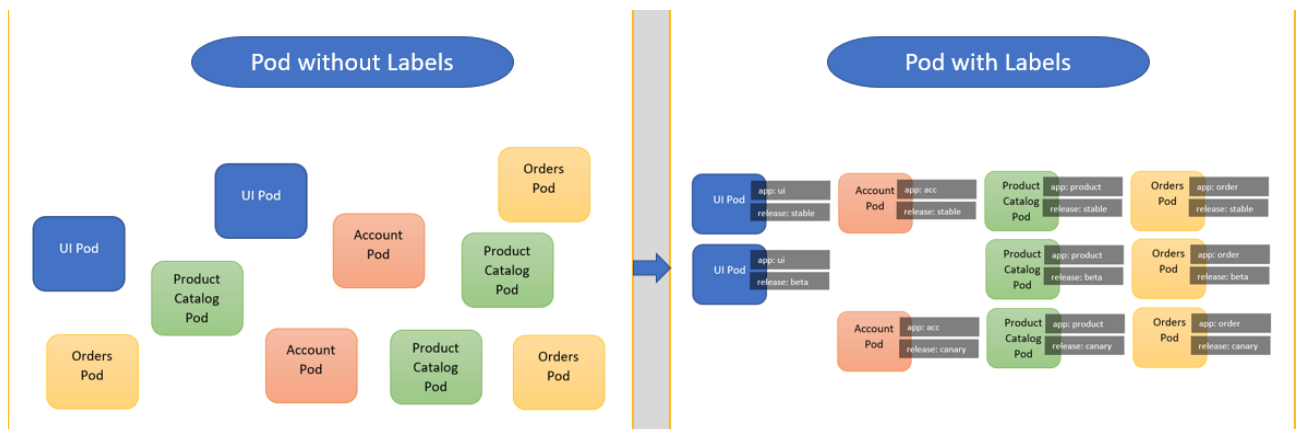
```
// show all labels on each Deployment
> kubectl get pod pod-nginx -o yaml
```





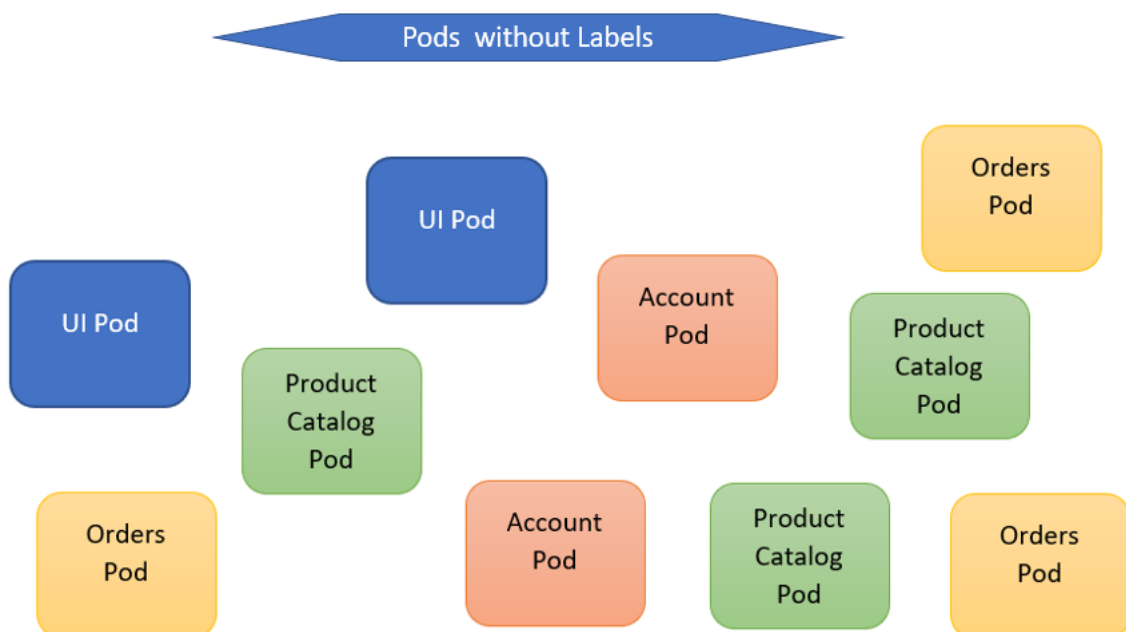
## 11. Pod Organization using Labels

In Microservice architecture, the number of deployed Pods will be replicated more than ones and multiple releases (i.e., stable, beta, canary) will run concurrently. This can lead to hundreds of pods within no time. So, organizing pods are crucial in microservice architecture.



### Pods without Labels:

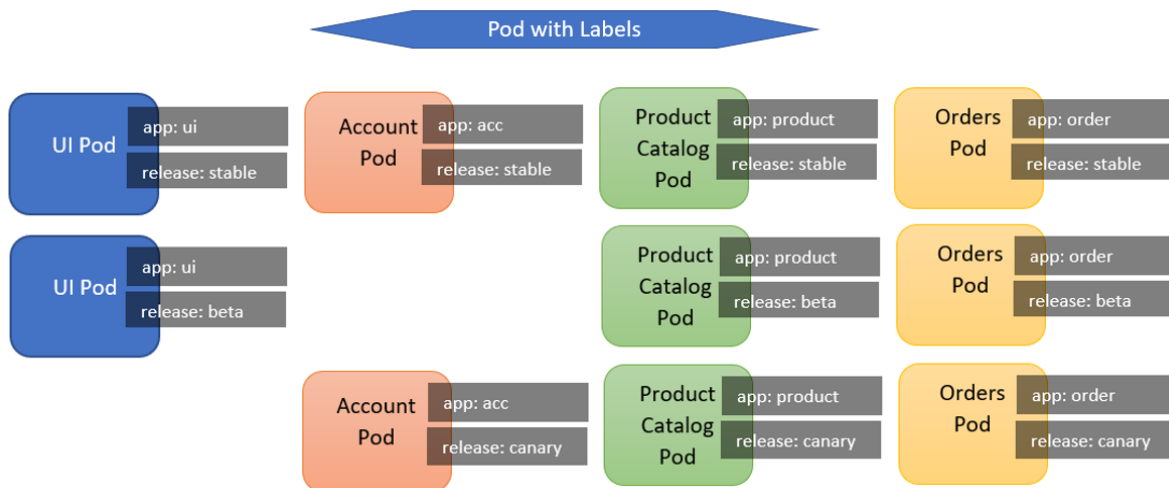
As per below image, we are running multiple microservices with multiple replicas and multiple releases without Pod labels. It is evident that we need to find a way to organize the Pods into smaller groups based on arbitrary criteria. Here, Pod Labels help us to organizing Pods into different groups.



## Pods with Labels:

As per below image, we are running multiple microservices with multiple replicas and multiple releases with Pod labels. Each Pod is labeled with two labels

- **app**, it specifies Pod belongs to which app, component, or service
- **release**, it specifies Pod running under stable, beta or a canary release



## Labels

Label is a key-value pair which is attached to pods, deployments, etc.

- It acts as an identifier on K8 object (Ex: Pod or Deployment). So, the other Kubernetes objects (Ex: Service, DaemonSet) can communicate by matching same label names.
- It can be attached to Kubernetes objects at creation time or directly on live objects by using Imperative way
- Same label key/value can be assigned to multiple Kubernetes objects
- Each object in the label can have set of key/value and each key must be unique for a given object
- Label key should not exceed 63 characters and allowed characters are alphanumeric, dash (-), underscore (\_), dot(.)
- Label value should not exceed 63 characters and allowed characters are alphanumeric, dash (-), underscore (\_), dot(.)

```
metadata:
  name: pod-label-demo
labels:
  release: stable
  app: ui
```

### Example: Creating labels on Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-label-demo
  labels:
    release: stable
    app: ui
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

There are two labels i.e., **app: ui**, **release: stable** are assigned to Pod

Use following kubectl commands to find each k8 object and respective labels

```
// show all labels on each Pod
> kubectl get pods --show-labels

// show all labels on each Deployment
> kubectl get deployments --show-labels

// show all Pods where label app: ui
> kubectl get pods -l app=ui

// add label to running Pod
// syntax
kubectl label Pod <podname> <key>=<value>
// example
> kubectl label Pod pod-label-demo release=stable

// delete Pods using label
> kubectl delete pod -l app=ui
```

## 12. Effective way of using K8 Liveness Probe

In General, Replication controller will keep running specified number of pods if application is crashed abruptly inside the Pod. However, there are situations where application might have crashed or deadlocked without terminating process. This is the place, where Kubernetes Health Probes comes into the picture.

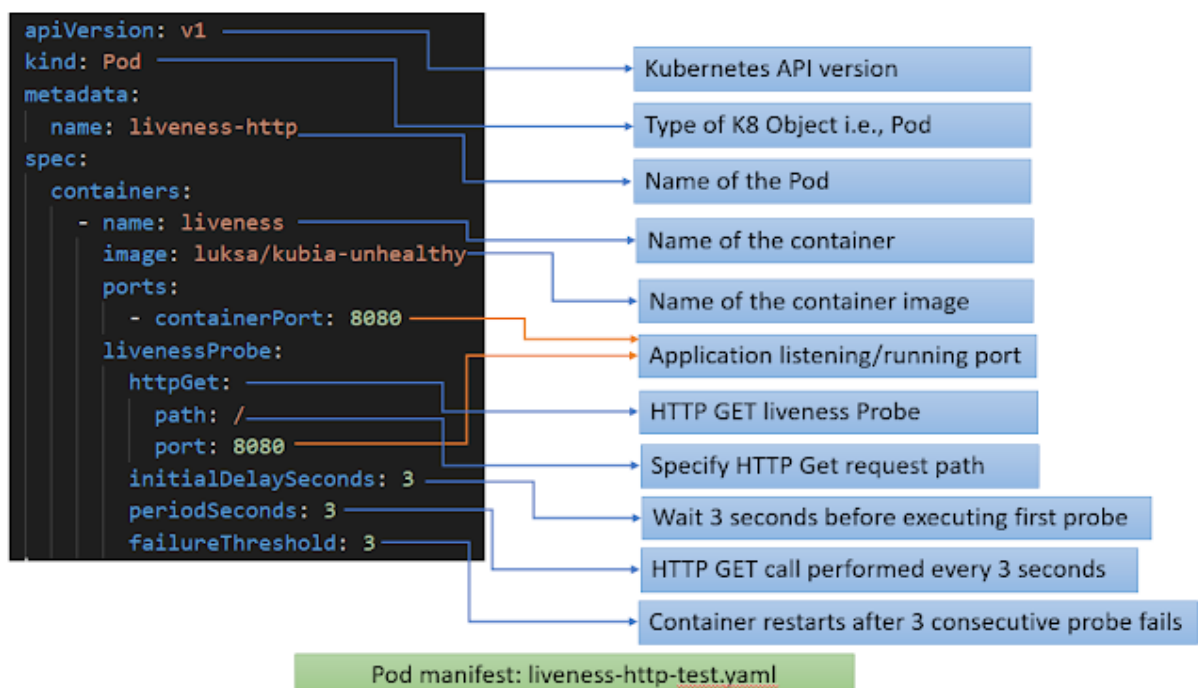
### Liveness Probe

Kubernetes checks if the container is alive through Liveness Probe and it will let kubelet to know when to restart a container. We can specify a liveness probe for each container in the pod specification. Kubernetes will periodically execute the liveness probe and restart the container if the probe fails.

Kubernetes can probe the container in three different ways.

#### 1. HTTP GET probe

It performs an HTTP GET request on the container's REST API resource. If the probe receives a response code between 2xx to 3xx is considered successful. If Http server returns different response code (i.e., other than 2xx to 3xx) or if it doesn't respond at all, the probe is considered a failure and the container will be restarted by kubelet.



```
apiVersion: v1
kind: Pod
metadata:
  name: liveness-http
```

```
spec:
  containers:
  - name: liveness
    image: luksa/kubia-unhealthy
    ports:
    - containerPort: 8080
    livenessProbe:
      httpGet:
        path: /
        port: 8080
      initialDelaySeconds: 3
      periodSeconds: 3
      failureThreshold: 3
```

As per code written in the specified node.js app, Http GET request returns response code as 500 after each fifth request. So, Pod will restart automatically after 3 consecutive failures.

```
// run the pod yaml file
> kubectl apply -f ./liveness-http-test.yaml

// view running pod details
> kubectl get po liveness-http-test

// view Pod complete details
> kubectl describe po liveness-http-test

// view previous terminated container logs
> kubectl logs liveness-http-test -previous
```

Always set an initialDelaySeconds based on your application startup time. If you do not set the initial delay, liveness probe check starts immediately before application accept the requests and which leads to probe failing, eventually Pod restarts in infinite loop.

## 2. TCP Socket probe

It opens a TCP connection to the specified port of the container. If the connection is established successfully, the probe is successful. Otherwise, the container will be restarted by kubelet.

```
apiVersion: v1
kind: Pod
metadata:
  name: liveness-tcp
spec:
  containers:
  - name: goproxy
    image: k8s.gcr.io/goproxy:0.1
```

```

ports:
  - containerPort: 8080
livenessProbe:
  tcpSocket:
    port: 8080
  initialDelaySeconds: 3
  periodSeconds: 3

```

### 3. Exec probe

It executes a specified command inside the container and checks the command's exit status code. If the status code is 0, the probe is successful. Otherwise, the container will be restarted by kubelet.

```

apiVersion: v1
kind: Pod
metadata:
  name: liveness-exec
spec:
  containers:
    - name: liveness
      image: k8s.gcr.io/busybox
      args:
        - /bin/sh
        - -c
        - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600
      livenessProbe:
        exec:
          command:
            - cat
            - /tmp/healthy
          initialDelaySeconds: 5
          periodSeconds: 5

```

When the container starts, it executes this command:

```
/bin/sh -c "touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600"
```

For the first 30 seconds of the container's life, there is a /tmp/healthy file. So, during the first 30 seconds, the command `cat /tmp/healthy` returns a success code. After 30 seconds, `cat /tmp/healthy` returns a failure code.

## Effective liveness probe check

you should always define a liveness probe. Without one, Kubernetes has no way of knowing whether your app is responding to HTTP requests. So, liveness probe helps us to restart the container when the application is unresponsive state.

- Configure the liveness probe to perform request on a specific URL path (i.e. /health)
- Make sure HTTP GET /health does not require validation or authentication
- Always check only internals of the app and should not depend on external services/dependencies.
- liveness probe should not return a failure when the server cannot connect to the database. If the underlying cause is in the database itself, restarting the web server container will not fix the problem.
- HTTP GET /health probe should not use many computational resources and should not take too long to complete. It supposed get response back in one second.
- liveness Prob failure Threshold default value is 3. So, it is not mandatory to specify again in pod yaml file.
- Do not set the same specification for Liveness and Readiness Probe
- Try to avoid "exec" probes as there are known problems with them
- Exit the process when uncaught exception occurs (i.e. kubelet will restart the container automatically) instead of signalling to liveness probe
- Liveness probe should be used as a recovery mechanism only when the process is not responsive.



---

## 13. Effective way of using K8 Readiness Probe

In General, Replication controller will keep running specified number of pods if application is crashed abruptly inside the Pod. However, there are situations where application might have crashed or deadlocked without terminating process. This is the place, where Kubernetes Health Probes comes into the picture.

### Liveness Probe

The complete details written @[Kubernetes for Developers #12: Effective way of using K8 Liveness Probe](#)

### Readiness Probe

The readiness probe determines when a container is ready to start accepting traffic. It invokes the probe periodically and If a pod reports that it is not ready, it is removed from the service. If the Pod then becomes ready again, it will be re-added automatically.

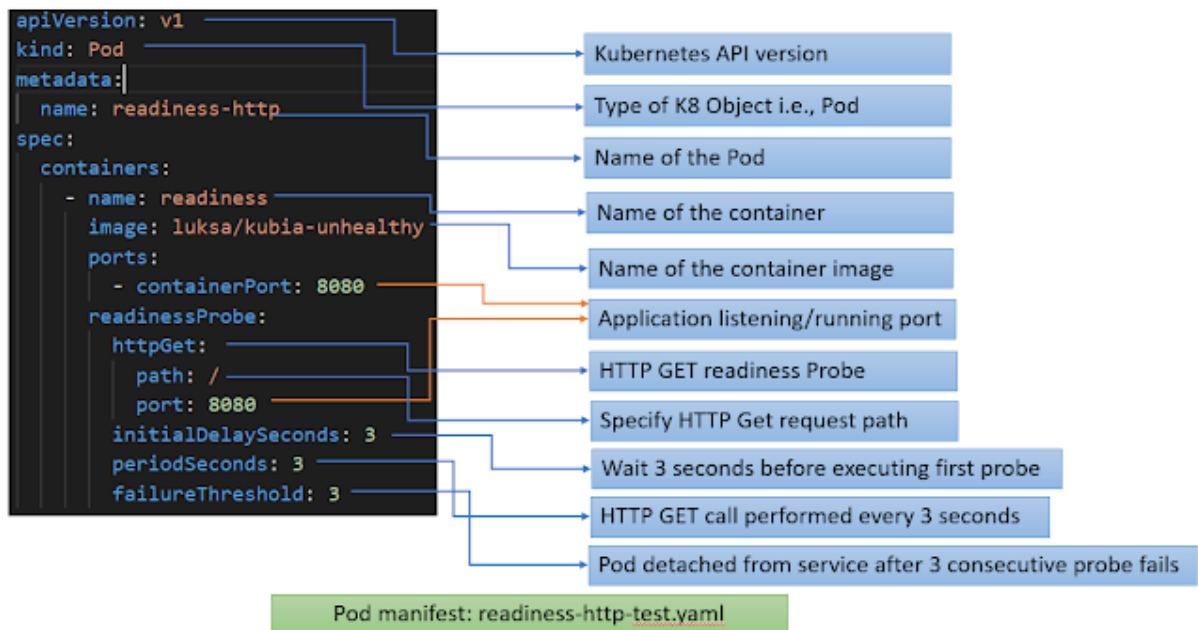
Unlike liveness probes, if a container fails the readiness check, it will not be restarted. The Pod will be detached from the service endpoint. Whenever Pod is ready to accept request, it will be attached to the Service endpoint again.

Liveness probes keep pods healthy by restarting unhealthy containers. whereas readiness probes make sure that only pods that are ready to serve requests receive them.

Kubernetes can probe the container in three different ways.

#### 1. HTTP GET probe

It performs an HTTP GET request on the container's REST API resource. If the probe receives a response code between 2xx to 3xx is considered successful. If Http server returns different response code (i.e. other than 2xx to 3xx) or if it doesn't respond at all, the probe is considered a failure and pod will be detached from the service endpoint.



```
apiVersion: v1
kind: Pod
metadata:
  name: readiness-http
spec:
  containers:
    - name: readiness
      image: luksa/kubia-unhealthy
      ports:
        - containerPort: 8080
      readinessProbe:
        httpGet:
          path: /
          port: 8080
        initialDelaySeconds: 3
        periodSeconds: 3
        failureThreshold: 3
```

```
// run the pod yaml file
> kubectl apply -f ./liveness-http-test.yaml

// view running pod details
> kubectl get po liveness-http-test

// view Pod complete details
> kubectl describe po liveness-http-test
```

```
// view previous terminated container logs
> kubectl logs liveness-http-test -previous
```

## 2. TCP Socket probe

It opens a TCP connection to the specified port of the container. If the connection is established successfully, the probe is successful. Otherwise, the pod will be detached from the service endpoint.

```
apiVersion: v1
kind: Pod
metadata:
  name: readiness-tcp
spec:
  containers:
    - name: goproxy
      image: k8s.gcr.io/goproxy:0.1
      ports:
        - containerPort: 8080
      readinessProbe:
        tcpSocket:
          port: 8080
        initialDelaySeconds: 5
        periodSeconds: 3
```

## 3. Exec probe

It executes a specified command inside the container and checks the command's exit status code. If the status code is 0, the probe is successful. Otherwise, the pod will be detached from the service endpoint.

```
apiVersion: v1
kind: Pod
metadata:
  name: readiness-exec
spec:
  containers:
    - name: readiness
      image: k8s.gcr.io/busybox
      args:
        - /bin/sh
        - -c
        - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600
      readinessProbe:
        exec:
          command:
```

```
- cat
- /tmp/healthy
initialDelaySeconds: 5
periodSeconds: 5
```

When the container starts, it executes this command:

```
/bin/sh -c "touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600"
```

For the first 30 seconds of the container's life, there is a /tmp/healthy file. So, during the first 30 seconds, the command `cat /tmp/healthy` returns a success code. After 30 seconds, `cat /tmp/healthy` returns a failure code.

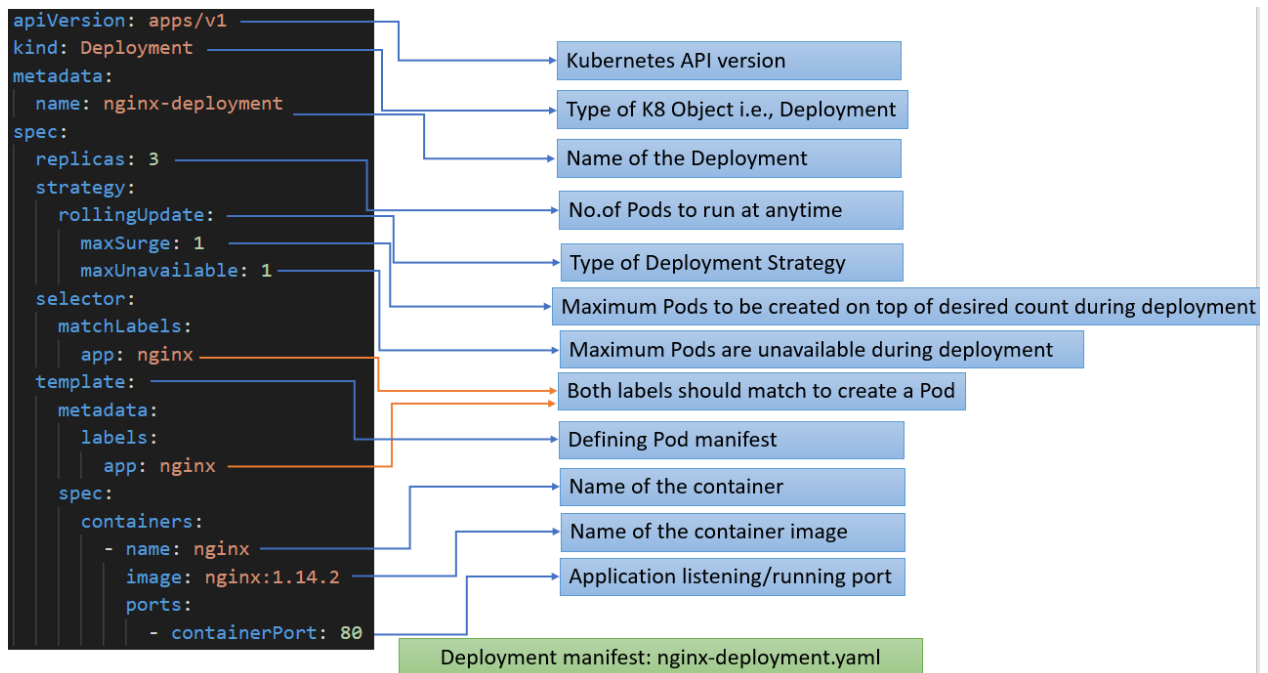
### Effective Readiness probe check

you should always define a Readiness probe. If your application takes too long to start listening for incoming connections, all the client requests hitting to this Pod will fail until an application is ready to accept.

- Configure the Readiness probe to perform request on a specific URL path (i.e. /ready)
- Make sure HTTP GET /ready does not require validation or authentication
- Liveness probes do not wait for readiness probes to succeed. Both probes run parallel. If you want to wait before executing a liveness probe you should use `initialDelaySeconds`.
- It is highly useful if your application needs to load lot of configuration/data during startup.
- Readiness and liveness probes can be used in parallel for the same container. Using both can ensure that traffic does not reach a container that is not ready for it, and that containers are restarted when they fail.
- liveness Prob `failureThreshold` default value is 3. So, it is not mandatory to specify again in pod yaml file.
- Do not set the same specification for Liveness and Readiness Probe
- Try to avoid "exec" probes as there are known problems with them

## 14. Kubernetes Deployment YAML manifest in-detail

In previous chapters, we have learned how to create or manage Pod manually. However, in real-world use cases, The Pod creation/re-creation should be done automatically and remain healthy without manual intervention, that's where K8 Deployment comes into the picture.



- It helps us to manage set of identical pods without creating/updating/deleting pods manually.
- It makes desired number of identical pods always stay up and running automatically without user intervention.
- It is a high-level resource, the actual pods are created and managed by the ReplicaSets, not by the Deployment directly.
- ReplicaSet constantly monitors the list of running pods and makes sure the actual number of pods are always matching the desired number of pods. If there are only few pods are running, it creates new replicas from a pod template. If there are many pods are running than desired number, it removes the excess replicas.
- Deployment YAML file contains both desired replica count and Pod template in it. Therefore, it will not depend on external Pod YAML file to create new Pod replicas.
- If one cluster node fails, Deployment will create desired number of replicas on another node automatically without manual intervention.

## Deployment Strategies

**1. RollingUpdate:** This is the default strategy for the Deployment. It removes old pods one by one, while adding new ones at the same time. This strategy helps us to serve the application without downtime.

**maxSurge** and **maxUnavailable** are two properties used to control the rate of rollout.

**maxSurge:** It determines how many maximum number of Pods can be created on top of the desired number of Pods. The number can be declared as absolute number (i.e. 1) or percentage (i.e. 20%). The default value is 25%

**maxUnavailable:** It determines maximum number of Pods that can be unavailable during deployment process. The number can be declared as absolute number (i.e. 1) or percentage (i.e. 20%). The default value is 25%

**Note:** Use this strategy only when your application can handle running both the old and new version at the same time.

**2. Recreate:** This strategy deletes all the old pods at once before creating new ones. This requires short period of complete application downtime.

**Note:** Use this strategy when your application does not support running multiple versions in parallel and requires the old version to be stopped completely before the new one is started.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
```

```
ports:
  - containerPort: 80
```

```
// Create a Deployment based on the YAML file
$ kubectl apply -f ./nginx-deployment.yaml
deployment.apps/nginx-deployment created
```

```
// Display information about all deployments
$ kubectl get deployments
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
nginx-deployment	2/3	2	2	64s

**NAME:** name of the deployment  
**READY:** How many Pods are available to access  
**UP-TO-DATE:** How many Pods are updated as per latest Pod template  
**AVAILABLE:** How many Pods are available to access  
**AGE:** Since how long application has been running

```
// Display ReplicaSet created by the Deployment
$ kubectl get rs
```

NAME	DESIRED	CURRENT	READY	AGE
nginx-deployment-6b474476c4	3	3	3	52m

**NAME:** name of the ReplicaSet i.e. [DEPLOYMENT NAME]-[RANDOM STRING]  
**DESIRED:** Display desired number of Pods which specified in the deployment.yaml file  
**CURRENT:** how many Pods are currently running  
**READY:** How many Pods are available to access  
**AGE:** Since how long application has been running

```
// Display information about the Pods
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
nginx-deployment-6b474476c4-b6j4j	1/1	Running	0	3h31m
nginx-deployment-6b474476c4-dk6wm	1/1	Running	0	3h31m
nginx-deployment-6b474476c4-xfdd9	1/1	Running	0	3h31m

```
// Display complete information about the Deployment
$ kubectl describe deployment nginx-deployment

// Filter Pods using labels
$ kubectl get pods -l app=nginx
```

```
// display deployment rollout history
$ kubectl rollout history deployment nginx-deployment
```

```
// undo deployment to previous revision
$ kubectl rollout undo deployment nginx-deployment
```

```
// undo deployment to specific revision
$ kubectl rollout undo deployment nginx-deployment --to-revision=2
```

```
// Delete the deployment
$ kubectl delete deployment nginx-deployment
```



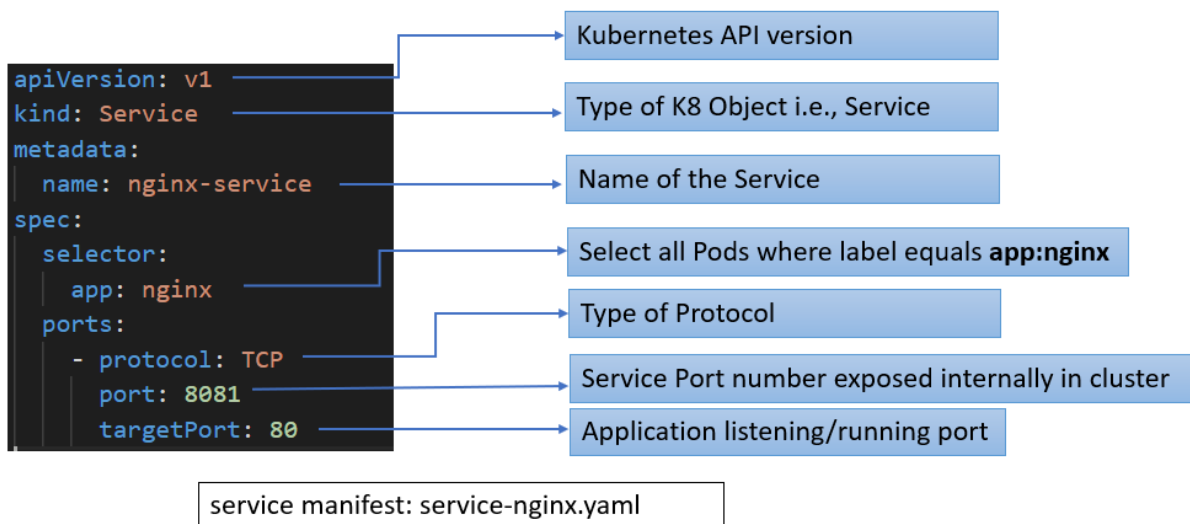
## 15. Kubernetes Service YAML manifest in-detail

In general, we use application IP address and Port to give access to the external world. However, In Kubernetes world, we are deploying our applications as K8 Pods and it has following limitations.

- Pods are designed to be ephemeral in nature which means they can be destroyed and re-created with new IP address at any time. Hence, relying on Pod IP address will not help us.
- Kubernetes assigns an IP address to a Pod after the pod has been scheduled to a node and before getting started. Hence, we cannot know the Pod IP address prior.
- As we run multiple Pods of same application, all Pods should be accessible through a single IP address to the external world.

Kubernetes Service comes into the picture to solve the above problems.

**Kubernetes Service** provides a single entry or network access to a group of Pods. Each service has an IP address and port that never change while the service exists. All the Pods can communicate with other Pods within the cluster or outside cluster using K8 Service.



- Kubernetes Service find the Pods based on their labels which is similar to K8 Deployment matchLabels.
- Once service is created, All the Pods in the cluster can communicate with each other using K8 Service IP address or Service name

- K8 Service can be created using YAML file or 'kubectl expose' command
- Internally K8 Endpoints keep track of all selected Pod IPs and these endpoints will be attached to the Service
- Kube-proxy module is used to assign virtual IP address for all K8 services.
- K8 Service is responsible for enabling network access to set of Pods whereas K8 Deployment is responsible for keeping set of Pods running
- K8 Service can be created with or without a Pod selector
- Once service is created, Kubernetes controller will create a resource called "Endpoints" with the same service name automatically. This "Endpoints" resource will keep track all the selected Pod IPs.
- K8 Service supports exposing more than one port from the application/container
- K8 Service **port** can be any number used to map **targetPort**. Try to use same number for both **port** and **targetPort** to avoid confusion.

### Create Kubernetes Deployment

Before creating service, first create K8 Deployment and make nginx Pods are up and running with label **app:nginx**

Read complete article on k8 deployment @ [Kubernetes for Developers #14: Kubernetes Deployment YAML manifest in-detail](#)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80
```

```
// Create a Deployment based on YAML file
$ kubectl apply -f ./nginx-deployment.yaml
deployment.apps/nginx-deployment created
```

```
// Display information about all deployments
$ kubectl get deployments
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
nginx-deployment	2/3	2	2	64s

```
// Display information about Pods with labels
$ kubectl get pods --show-labels
```

NAME	READY	STATUS	RESTARTS	AGE	LABELS
nginx-deployment-6b474476c4-8mx72	1/1	Running	0	30h	app=nginx
nginx-deployment-6b474476c4-cdhvb	1/1	Running	0	30h	app=nginx
nginx-deployment-6b474476c4-gw975	1/1	Running	0	30h	app=nginx

## Create Kubernetes Service

K8 Service will be created in following two ways

### 1. Create Service using YAML manifest

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
```

```
port: 8081
targetPort: 80
```

```
// Create a Service using YAML file
$ kubectl apply -f ./nginx-service.yaml
service/nginx-service created
```

```
// Display information about Services
$ kubectl get services
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
nginx-service	ClusterIP	10.105.224.36	<none>	8081/TCP	112s

K8 Endpoint object will be created automatically when the Service is created with same service name. It will track all the Pod IPs where label as **app:nginx**

```
// Display information about the Endpoints
$ kubectl get endpoints
```

NAME	ENDPOINTS	AGE
nginx-service	10.1.1.68:80,10.1.1.69:80,10.1.1.70:80	19m

NAME: Name of the Endpoint which is same as Service name  
ENDPOINTS: list all the Pod IPs where label as app:nginx

By default, Kubernetes Service IP address (i.e. Cluster-IP) and Port will be accessed within the cluster only. So, enter into one of the nginx Pod and make **curl** using service IP address/name and Port to view nginx webpage

```
// Enter into nginx Pod shell
$ kubectl exec -it nginx-deployment-6b474476c4-8mx72 -- bin/bash

// update packages
# apt-get update
```

```
// install curl
# apt-get install curl

// make curl using service name(or IP ) and port to access nginx webpage
# curl http://nginx-service:8081
(or)
# curl http://10.105.224.36:8081
```

There is another way to test service by port-forwarding from service port to local computer port

```
// use kubectl port-forward to test the service in local computer
$ kubectl port-forward service/nginx-service 8081:8081
Forwarding from 127.0.0.1:8081 -> 80
Forwarding from [::1]:8081 -> 80

// Go to browser and hit http://localhost:8081 to view nginx webpage
```

## 2. Create Service using kubectl expose command

```
$ kubectl expose deployment nginx-deployment --name=nginx-service --
port=8081 --target-port=80
service/nginx-service exposed
```

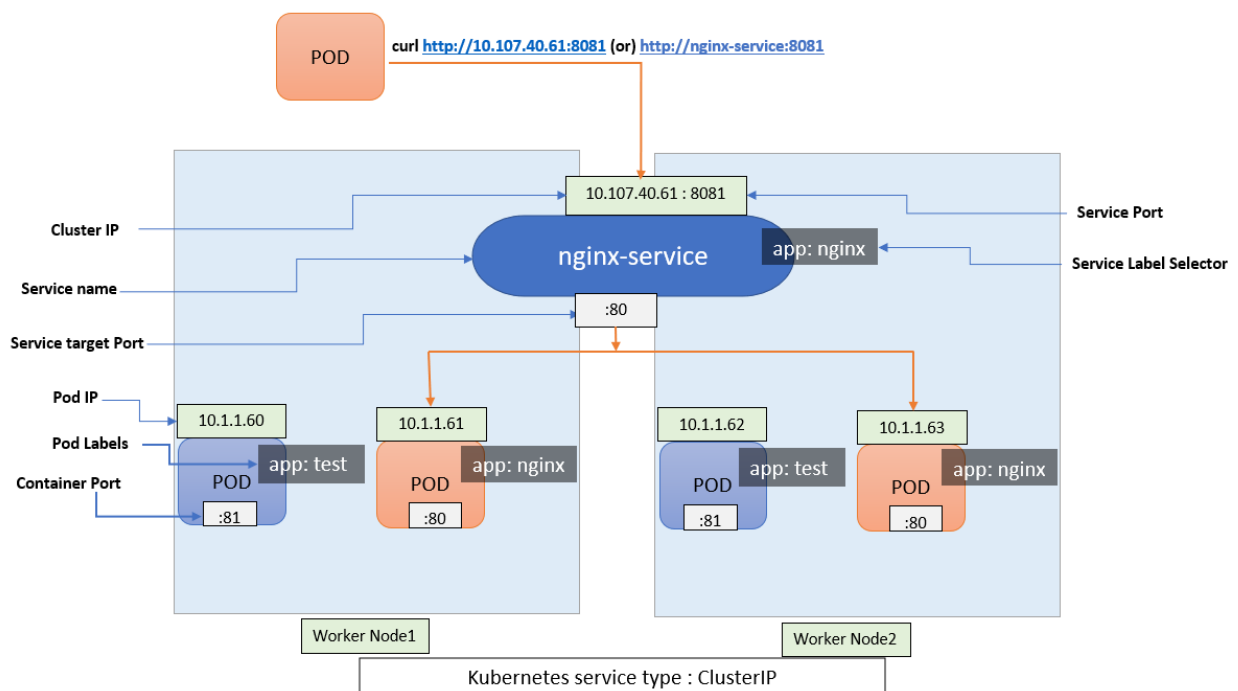
## 16. Kubernetes Service Types - ClusterIP, NodePort, LoadBalancer and ExternalName

In the previous chapter ([Kubernetes for Developers #15: Kubernetes Service YAML manifest in-detail](#)) we have seen how to create K8 Service and access the Pods inside the cluster.

There are four types of services available to access the Pods in and outside cluster.

### 1. ClusterIP

- It exposes the service within the Kubernetes cluster only.
- Only Pods within the K8 Cluster can communicate using this service. You cannot access directly from the browser using Service IP.
- Kubernetes controller creates unique virtual IP address (i.e., which is called ClusterIP) whenever the service is created.
- This is the default **serviceType**



As per above diagram,

- The K8 service "**nginx-service**" has been created with "ClusterIP" type.
- nginx-service mapped to Pod labels **app:nginx**
- nginx-service mapped service port 8081(i.e. **Port** ), container port 80 (i.e. **targetPort** )
- As it is "ClusterIP" type, we can't access the service outside the cluster. So, enter into any Pod and make curl using service IP or name or FQDN (<servicename>.<namespace>.svc.cluster.local) i.e.

```
curl http://10.107.40.61:8081 (or) curl http://nginx-service:8081 (or)
curl http://nginx-service.default.svc.cluster.local:8081
```

## Create Kubernetes Deployment

Before creating service, first create K8 Deployment and make nginx Pods are up and running with label **app:nginx**

Read complete article on k8 deployment @ [Kubernetes for Developers #14: Kubernetes Deployment YAML manifest in-detail](#)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80
```

```
// Create a Deployment based on YAML file
$ kubectl apply -f ./nginx-deployment.yaml
deployment.apps/nginx-deployment created
```

```
// Display information about all deployments
$ kubectl get deployments
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
nginx-deployment	2/2	2	2	64s

```
// Display information about Pods with labels
```

```
$ kubectl get pods --show-labels
```

NAME	READY	STATUS	RESTARTS	AGE	LAB
ELS					
nginx-deployment-6b474476c4-8mx72	1/1	Running	0	30h	app=nginx
nginx-deployment-6b474476c4-cdhvb	1/1	Running	0	30h	app=nginx

### Create ClusterIP Service using YAML manifest

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  type: ClusterIP
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 8081
      targetPort: 80
```

```
// Create a Service using YAML file
```

```
$ kubectl apply -f ./nginx-service-cluserIP.yaml
service/nginx-service created
```

```
// Display information about Services
```

```
$ kubectl get services
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
nginx-service	ClusterIP	10.105.224.36	<none>	8081/TCP	112s



K8 Endpoint object will be created automatically when the Service is created with same service name. It will track all the Pod IPs where label as **app:nginx**

```
// Display information about the Endpoints
$ kubectl get endpoints
NAME                ENDPOINTS                                AGE
nginx-service       10.1.1.61:80,10.1.1.63:80              19m

NAME: Name of the Endpoint which is same as Service name
ENDPOINTS: list all the Pod IPs where label as app:nginx
```

By default, Kubernetes Service IP address (i.e. Cluster-IP) and Port will be accessed within the cluster only. So, enter one of the nginx Pod and make curl using service IP address/name and Port to view nginx webpage

```
// Enter into nginx Pod shell
$ kubectl exec -it nginx-deployment-6b474476c4-8mx72 -- bin/bash

// update packages
# apt-get update

// install curl
# apt-get install curl

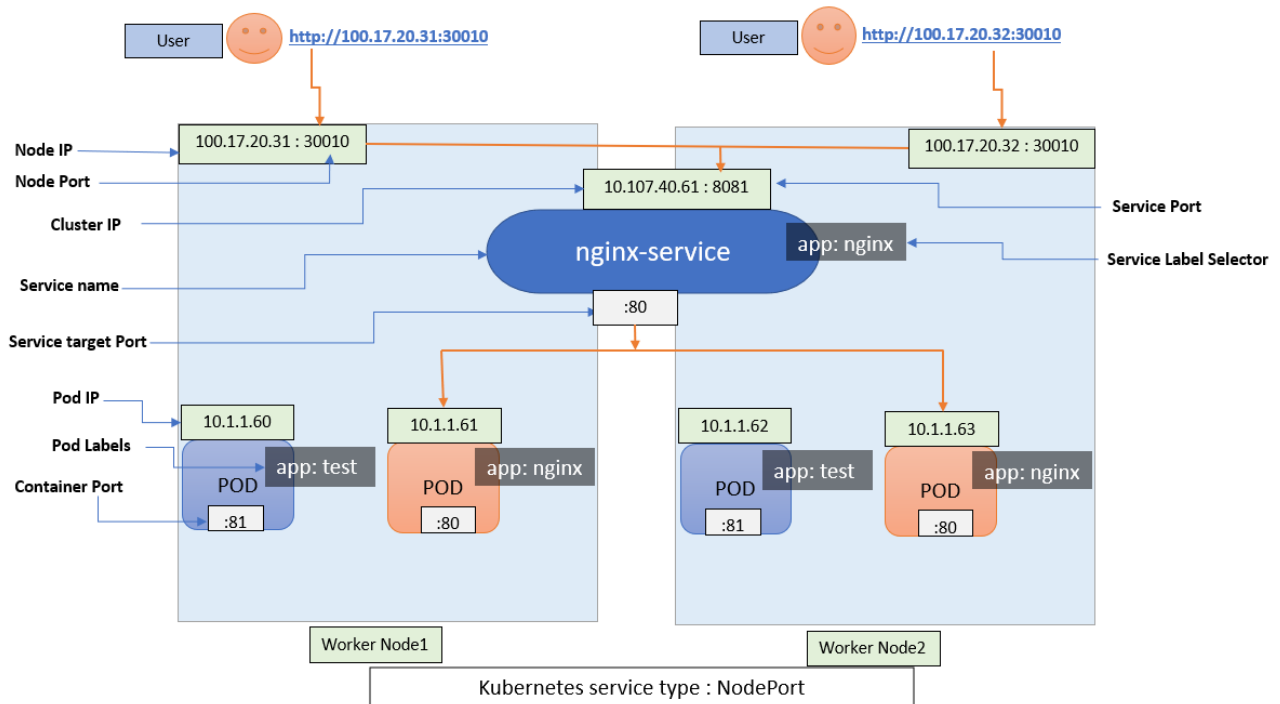
// make curl using service name(or IP ) and port to access nginx webpage
# curl http://nginx-service:8081
(or)
# curl http://10.105.224.36:8081
```

There is another way to test service by port-forwarding from service port to local computer port

```
// use kubectl port-forward to test the service in local computer
$ kubectl port-forward service/nginx-service 8081:8081
Forwarding from 127.0.0.1:8081 -> 80
Forwarding from [::1]:8081 -> 80
// Go to browser and hit http://localhost:8081 to view nginx webpage
```

## 2. NodePort

- It exposes the service both in and outside the cluster
- It exposes the service on each Worker Node's IP at a static port (i.e., which is called NodePort).
- A ClusterIP Service will be created automatically whenever the NodePort service is created. It means, the external traffic enter into the cluster using <NodeIP>:<NodePort> (ex: 100.72.40.61: 30010) and it direct traffic to the ClusterIP
- NodePort must be within the range from 30000-32767



As per above diagram,

- The K8 service "**nginx-service**" has been created with "NodePort" type.
- nginx-service mapped to Pod labels **app:nginx**
- nginx-service mapped service port 8081(i.e. **Port**), container port 80 (i.e. **targetPort** ) and node Port 30010
- As it is "NodePort" type, we can access the service both inside and outside the cluster.
- To access inside cluster, enter into any Pod and make curl using service IP or name or FQDN (<servicename>.<namespace>.svc.cluster.local) i.e.  
curl <http://10.107.40.61:8081> (or) curl <http://nginx-service:8081> (or)  
curl <http://nginx-service.default.svc.cluster.local:8081>
- To access outside cluster, go to browser and hit <http://<nodeIP>:<nodePort>>  
i.e. <http://100.17.20.31:30010>

## Create NodePort Service using YAML manifest

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  type: NodePort
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 8081
      targetPort: 80
      nodePort: 30010
```

```
// Create a Service using YAML file
$ kubectl apply -f ./nginx-service-nodeport.yaml
service/nginx-service created
```

```
// Display information about Services
$ kubectl get services
```

NAME	TYPE	CLUSTER-IP	EXTERNAL -
IP	PORT(S)	AGE	
service/nginx-			
service	NodePort	10.105.224.36	<none> 8081:30010/TCP 15h

K8 Endpoint object will be created automatically when the Service is created with same service name. It will track all the Pod IPs where label as **app:nginx**

```
// Display information about the Endpoints
$ kubectl get endpoints
```

NAME	ENDPOINTS	AGE
nginx-service	10.1.1.61:80,10.1.1.63:80	19m

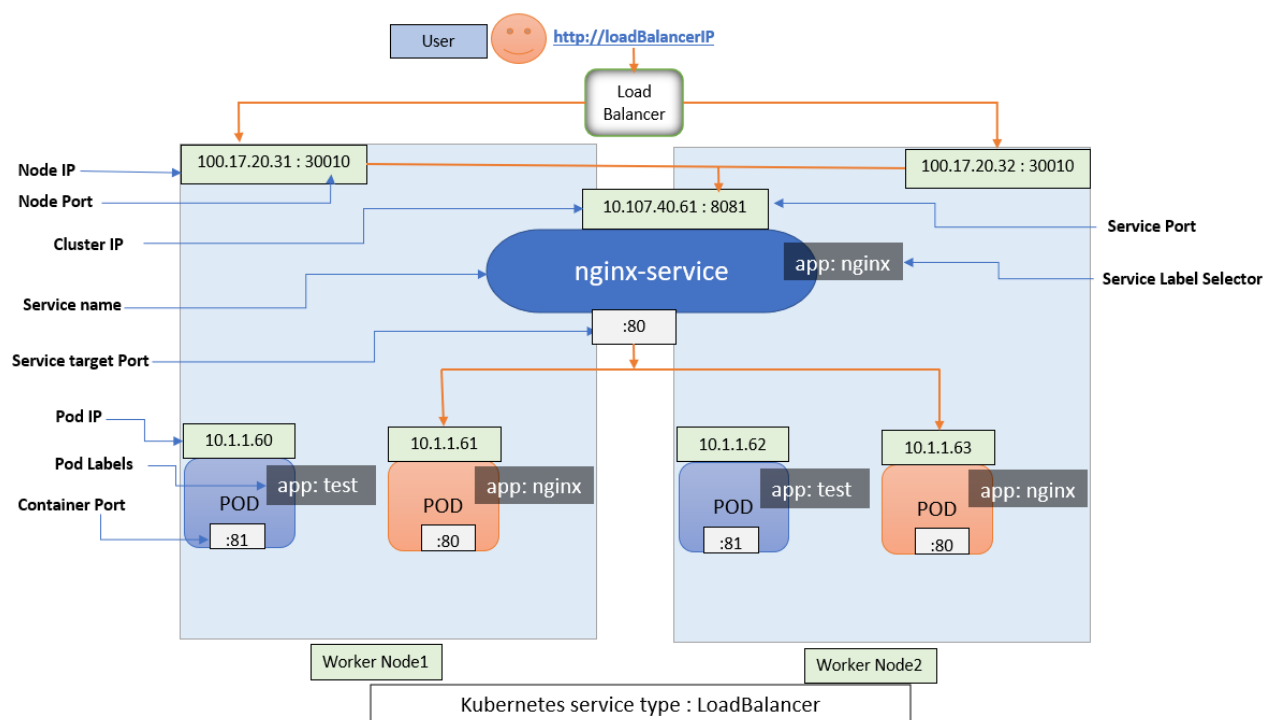
NAME: Name of the Endpoint which is same as Service name  
ENDPOINTS: list all the Pod IPs where label as app:nginx

Go to browser and enter <http://100.17.20.31:30010> to view nginx webpage.

Go to browser and enter <http://localhost:30010> if you are testing locally using docker-desktop or minikube.

### 3. LoadBalancer

- It exposes the service both in and outside the cluster
- It exposes the service externally using cloud providers load balancer.
- NodePort and ClusterIP services will be created automatically whenever the LoadBalancer service is created.
- The LoadBalancer service redirects traffic to the node port across all the nodes.
- The external clients connect to the service through load balancer IP
- This is the most preferable approach to expose service outside the cluster



As per above diagram,

- The K8 service "**nginx-service**" has been created with "LoadBalancer" type.
- nginx-service mapped to Pod labels **app:nginx**
- nginx-service mapped service port 8081(i.e. **Port**), container port 80 (i.e. **targetPort**) and node Port 30010
- As it is "LoadBalancer" type, we can access the service both inside and outside the cluster.
- To access inside cluster, enter into any Pod and make curl using service IP or name or FQDN (<servicename>.<namespace>.svc.cluster.local) i.e.

curl <http://10.107.40.61:8081> (or) curl <http://nginx-service:8081> (or)  
curl <http://nginx-service.default.svc.cluster.local:8081>

- To access outside cluster, go to browser and hit <http://<loadBalancerIP>>  
i.e. <http://2343260762.us-east-1.elb.amazonaws.com>

### Create LoadBalancer Service using YAML manifest

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  type: LoadBalancer
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 8081
      targetPort: 80
      nodePort: 30010
```

```
// Create a Service using YAML file
$ kubectl apply -f ./nginx-service-lb.yaml
service/nginx-service created
```

```
// Display information about Services
```

NAME	TYPE	CLUSTER-IP	EXTERNAL -
IP	PORT(S)	AGE	
service/nginx-service	LoadBalancer	10.105.224.36	2343260762.us-east-
1.elb.amazonaws.com	8081:30010/TCP	2h	

Go to browser and enter <http://2343260762.us-east-1.elb.amazonaws.com> to view nginx webpage.  
Go to browser and enter <http://localhost:30010> if you are testing locally using docker-desktop or minikube.

#### 4. ExternalName

- It maps service to a DNS name , typically domain/subdomain name (ex: foo.example.com )
- It will redirect a request to domain specified in the externalName
- There won't be any difference while accessing service inside Pod. (i.e it should be external-service.default.svc.cluster.local )

```
apiVersion: v1
kind: Service
metadata:
  name: external-service
spec:
  type: ExternalName
  externalName: example.api.com
```

```
// Create a Service using YAML file
$ kubectl apply -f ./external-service.yaml
service/external-service created
```

```
// Display information about Services
```

NAME	TYPE	CLUSTER-IP	EXTERNAL -
IP	PORT(S)	AGE	
service/external-			
service	ExternalName	<none>	example.api.com
9m31s			<none>

## 17. Expose service using Kubernetes Ingress

In the previous chapter([Kubernetes for Developers #16: Kubernetes Service Types - ClusterIP, NodePort, LoadBalancer and ExternalName](#)) we discussed “LoadBalancer” service type would be the most preferable technique to expose service outside the cluster. However, it requires one LoadBalancer per service with public IP address. It is cost inefficient if you have bunch of services needs to be exposed outside the cluster.

We can solve this problem by implementing single Kubernetes Ingress resource with multiple service mapping.

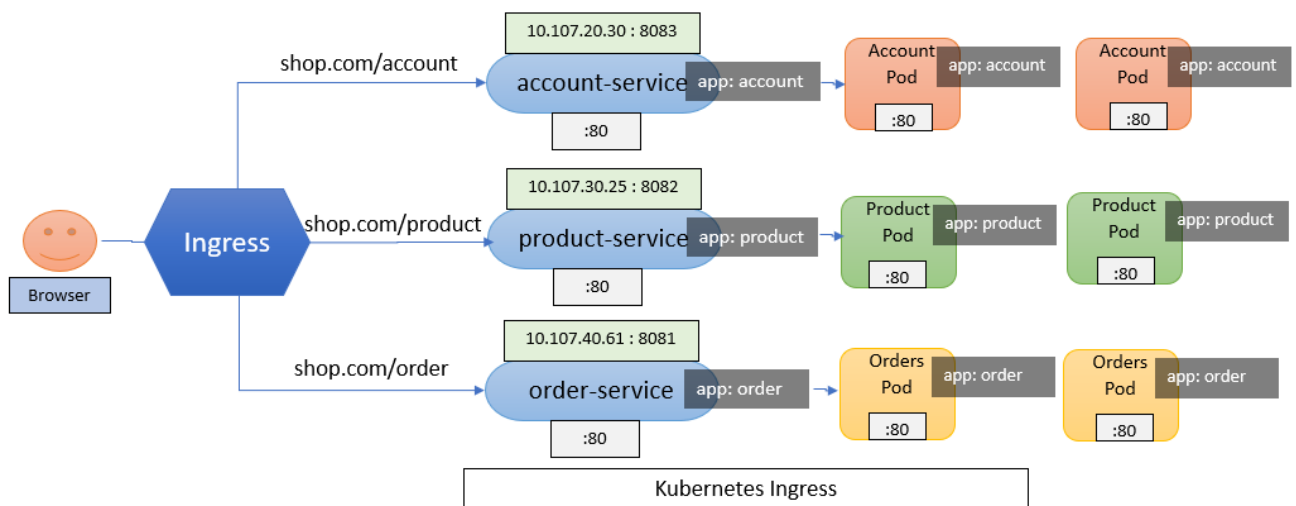
K8 Ingress works based on URL based HTTP routing. So, we can map multiple services to multiple HTTP URLs. When a client sends an HTTP request to the Ingress, the URL host and path in the request determine which service the request is supposed to navigate.

Kubernetes Ingress configuration has two parts.

**1. Ingress Resource:** It is YAML manifest file which contain set of HTTP routing rules mapped to services

**2. Ingress Controller:** It is K8 Pod running inside the cluster. It is responsible to process Ingress Resource YAML file and route the request to appropriate service. Ingress controller Pods are not started automatically with a cluster. Different K8 environments use different implementations of the controller.

**Note:** DNS name should resolve to the IP address of Ingress controller to access K8 service from the ingress controller.



As per diagram,

- when you initiate a http call from the browser (i.e. shop.com/cart), it performs DNS lookup and returns a IP address of the Ingress controller. So, make sure that DNS name should map to IP address of the Ingress controller.
- Browser sends HTTP request to the Ingress controller; it determines the service which is mapped to the given http URL based on HTTP headers
- Ingress controller find the Pod IPs through Endpoints associated with the service
- HTTP request will be forwarded to one of the Pod.
- Ingress controller uses the service to find Pod IPs, then the controller will process the request.

### Enable the Nginx Ingress controller

1. run the following kubectl command to enable Nginx Ingress controller on Docker Desktop

```
$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v0.47.0/deploy/static/provider/cloud/deploy.yaml
```

run the following command if you are using minikube

```
$ minikube addons enable ingress
```

2. run the following kubectl command to check the “ingress-nginx-controller” Pod running status

```
$ kubectl get pods -n ingress-nginx
```

NAME	READY	STATUS	RESTARTS
ingress-nginx-controller-84dcb9867d-dqc9h	1/1	Running	0



## Create Kubernetes Deployment

Before creating an Ingress, first create two different versions of helloworld app using K8 Deployment and service

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-v1-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: v1-web
  template:
    metadata:
      labels:
        app: v1-web
    spec:
      containers:
        - name: hello-v1
          image: gcr.io/google-samples/hello-app:1.0
          ports:
            - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: hello-v1-service
spec:
  type: ClusterIP
  selector:
    app: v1-web
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
```

```
// save above yaml manifest as hello-v1.yaml and run below command
$ kubectl apply -f hello-v1.yaml
deployment.apps/hello-v1-deployment created
service/hello-v1-service created
```

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-v2-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: v2-web
  template:
    metadata:
      labels:
        app: v2-web
    spec:
      containers:
        - name: hello-v2
          image: gcr.io/google-samples/hello-app:2.0
          ports:
            - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: hello-v2-service
spec:
  type: ClusterIP
  selector:
    app: v2-web
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080

```

```

// save above yaml manifest as hello-v2.yaml and run below command
$ kubectl apply -f hello-v2.yaml
deployment.apps/hello-v2-deployment created
service/hello-v2-service created

```

## Create an Ingress YAML manifest to expose multiple services on single host

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: hello-ingress
spec:
  rules:
    - host: shop.com
      http:
        paths:
          - path: /product
            backend:
              serviceName: hello-v1-service
              servicePort: 8080
          - path: /order
            backend:
              serviceName: hello-v2-service
              servicePort: 8080
```

```
// create an ingress resource
$ kubectl apply -f hello-ingress.yaml
ingress.extensions/hello-ingress created
```

```
// get an ingress details
$ kubectl get ingress
NAME           CLASS    HOSTS      ADDRESS    PORTS    AGE
hello-ingress  <none>   shop.com   localhost  80       73s
```

```
// open windows host file (C:\Windows\System32\drivers\etc) and map shop.com to localhost
127.0.0.1 shop.com
```

```
// make curl command to shop.com/product to get hello-v1 app details
$ curl shop.com/product
```

```
Hello, world!
Version: 1.0.0
Hostname: hello-v1-deployment-855c95579b-d99pp

// make curl command to shop.com/order to get hello-v2 app details
$ curl shop.com/order
Hello, world!
Version: 2.0.0
Hostname: hello-v2-deployment-5fc58f68df-hwgxx
```

### Create an Ingress YAML manifest to expose multiple services on sub domains

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: hello-ingress
spec:
  rules:
    - host: product.shop.com
      http:
        paths:
          - path: /
            backend:
              serviceName: hello-v1-service
              servicePort: 8080
    - host: order.shop.com
      http:
        paths:
          - path: /
            backend:
              serviceName: hello-v2-service
              servicePort: 8080
```

```
// create an ingress resource to map multiple sub domains
$ kubectl apply -f hello-multi-domain-ingress.yaml
ingress.extensions/hello-ingress created
```

```
// get an ingress details
$ kubectl get ingress
```

NAME	CLASS	HOSTS	ADDRESS	PORTS
hello-ingress	<none>	product.shop.com,order.shop.com	80	6s

```
// open windows host file (C:\Windows\System32\drivers\etc) and map shop.com to localhost
127.0.0.1 product.shop.com
127.0.0.1 order.shop.com
```

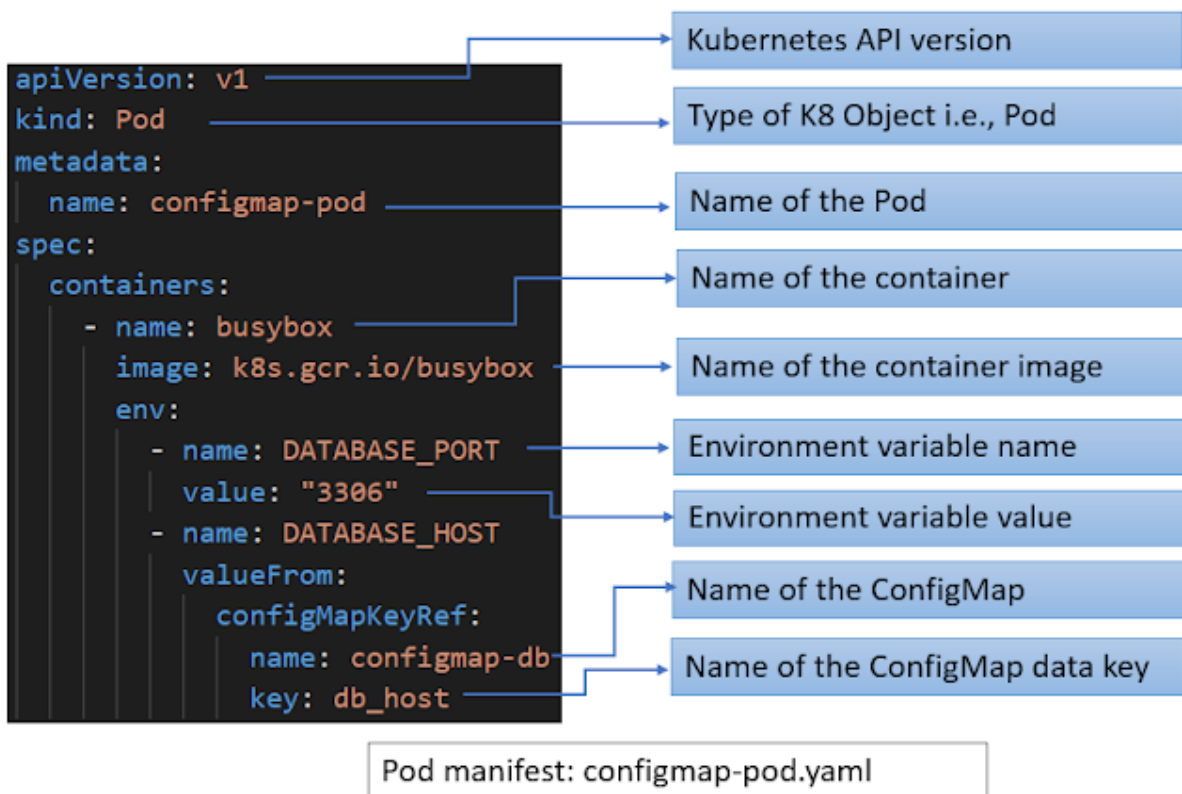
```
// make curl command to product.shop.com to get hello-v1 app details
$ curl product.shop.com
Hello, world!
Version: 1.0.0
Hostname: hello-v1-deployment-855c95579b-d99pp
```

```
// make curl command to order.shop.com to get hello-v2 app details
$ curl order.shop.com
Hello, world!
Version: 2.0.0
Hostname: hello-v2-deployment-5fc58f68df-hwgxx
```

## 18. Manage app settings using ConfigMap

In general, we use configuration files (i.e., web.config and .rc) or environment files (.env) for separating application settings from the code. The same application settings can be implemented using Kubernetes ConfigMap Object in the K8 world.

- It stores data as key-value pair.
- It helps us to separate application environment specific configurations from the code.
- Each Pod uses ConfigMap values as environment variables, volumes or command-line arguments.
- Key name must contain alphanumeric, dash (-), underscore (\_) and dot(.) only
- It is not suitable for storing confidential data.



### Create ConfigMap

There are multiple ways to create K8 ConfigMap object

#### 1. from YAML file

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: configmap-db
```

```
data:
  db_host: "mydb.rds.amazonaws.com"
  db_port: "3306"
  db_name: "testuser"
```

Here, we should specify key-value pair as **data** property in the K8 ConfigMap object

Save above yaml content as "configmapdb.yaml" and run following kubectl command

```
// create configmap object from yaml file
$ kubectl apply -f configmapdb.yaml
configmap/configmap-db created
```

```
// display all configmap objects
$ kubectl get configmap
NAME          DATA    AGE
configmap-db  3        2m
```

```
// view configmap object details using describe command
$ kubectl describe configmap configmap-db

// view configmap object details as yaml file
$ kubectl get configmap configmap-db -o yaml
```

## 2. from literals

In this approach, we pass key-value pair as part of kubectl command directly.

```
// create configmap object from literals
// syntax
kubectl create configmap <configmap-name> --from-
literal=<key_name>=<key_value>
```

```
$ kubectl create configmap configmap-ui --from-literal=app_name=mytestapp --from-literal=app_theme=bluetheme
configmap/configmap-ui created
```

```
// display all configmap objects
$ kubectl get configmap
NAME          DATA   AGE
configmap-db   3       134m
configmap-ui    2       19s
```

```
// view configmap object details using describe command
$ kubectl describe configmap configmap-ui

// view configmap object details as yaml file
$ kubectl get configmap configmap-ui -o yaml
```

### 3. from an env-file

In this approach, we create an env file with list of environment variables and values

```
db_host=mydb.rds.amazonaws.com
db_port=3306
db_name=testuser
```

save above content as "configmap-db-env.txt" or "configmap-db-env.properties" and run following kubectl command

```
// create configmap object from env file
$ kubectl create configmap configmap-db-env --from-env-file=configmap-db-env.txt
configmap-db-env created
```



```
// display all configmap objects
```

```
$ kubectl get configmap
```

NAME	DATA	AGE
configmap-db	3	153m
configmap-db-env	3	6s
configmap-ui	2	18m

```
// view configmap object details using describe command
```

```
$ kubectl describe configmap configmap-db-env
```

```
// view configmap object details as yaml file
```

```
$ kubectl get configmap configmap-db-env -o yaml
```

## Passing ConfigMap values to Pod

### 1. passing ConfigMap values to container environment variables

In the below example, we are setting Pod container individual environment variables from “configmap-db” object

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-pod
spec:
  containers:
    - name: busybox
      image: k8s.gcr.io/busybox
      command: ["/bin/sh", "-c", "env"]
      env:
        - name: DATABASE_PORT
          value: "3306"
        - name: DATABASE_HOST
          valueFrom:
            configMapKeyRef:
              name: configmap-db
              key: db_host
        - name: DATABASE_NAME
          valueFrom:
            configMapKeyRef:
              name: configmap-db
```

```
key: db_name
```

save above yaml content as "configmap-pod-valuefrom.yaml" and run following kubectl command

```
// create busybox Pod
$ kubectl apply -f configmap-pod-valuefrom.yaml
pod/configmap-pod created
```

```
// check environment values from busybox Pod
$ kubectl logs pod/configmap-pod | grep DATABASE
DATABASE_PORT=3306
DATABASE_NAME=testuser
DATABASE_HOST=mydb.rds.amazonaws.com
```

## 2. passing complete ConfigMap Object as environment variables to the container

In the below example, we are passing all "configmap-db" object key-value pairs as container environment variables.

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-pod-env
spec:
  containers:
    - name: busybox
      image: k8s.gcr.io/busybox
      command: ["/bin/sh", "-c", "env"]
      envFrom:
        - configMapRef:
            name: configmap-db
```

save above yaml content as "configmap-pod-envfrom.yaml" and run following kubectl command

```
// create busybox Pod
$ kubectl apply -f configmap-pod-envfrom.yaml
pod/configmap-pod-env created
```

```
// check environment values from busybox Pod
$ kubectl logs pod/configmap-pod-env | grep db
db_port=3306
db_name=testuser
db_host=mydb.rds.amazonaws.com
```

### 3. passing ConfigMap defined environment variables as container command arguments

In the below example, we are passing environment variables as container command arguments with `$(environment_variable)` syntax

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-pod-command
spec:
  containers:
  - name: busybox
    image: k8s.gcr.io/busybox
    command:
    [
      "/bin/echo",
      "dbhostname: $(DATABASE_HOST) and dbport: $(DATABASE_PORT)",
    ]
    env:
    - name: DATABASE_HOST
      valueFrom:
        configMapKeyRef:
          name: configmap-db
          key: db_host
    - name: DATABASE_PORT
      valueFrom:
        configMapKeyRef:
          name: configmap-db
          key: db_port
```

save above yaml content as "configmap-pod-command.yaml" and run following kubectl command

```
// create busybox Pod
$ kubectl apply -f configmap-pod-command.yaml
pod/configmap-pod-command created
```

```
// check logs from busybox Pod
$ kubectl logs pod/configmap-pod-command
dbhostname: mydb.rds.amazonaws.com and dbport: 3306
```

#### 4. Attaching ConfigMap as container volume

In the below example,

- we are first creating **volumes** by setting configMap name property as "configmap-db"
- we are accessing this volume by setting **volumeMounts** property

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-pod-volume
spec:
  containers:
    - name: busybox
      image: k8s.gcr.io/busybox
      command: ["/bin/sh", "-c", "ls /etc/configmap/"]
      volumeMounts:
        - name: configmap-volume
          mountPath: /etc/configmap
  volumes:
    - name: configmap-volume
      configMap:
        name: configmap-db
```

save above yaml content as "configmap-pod-volume.yaml" and run following kubectl command

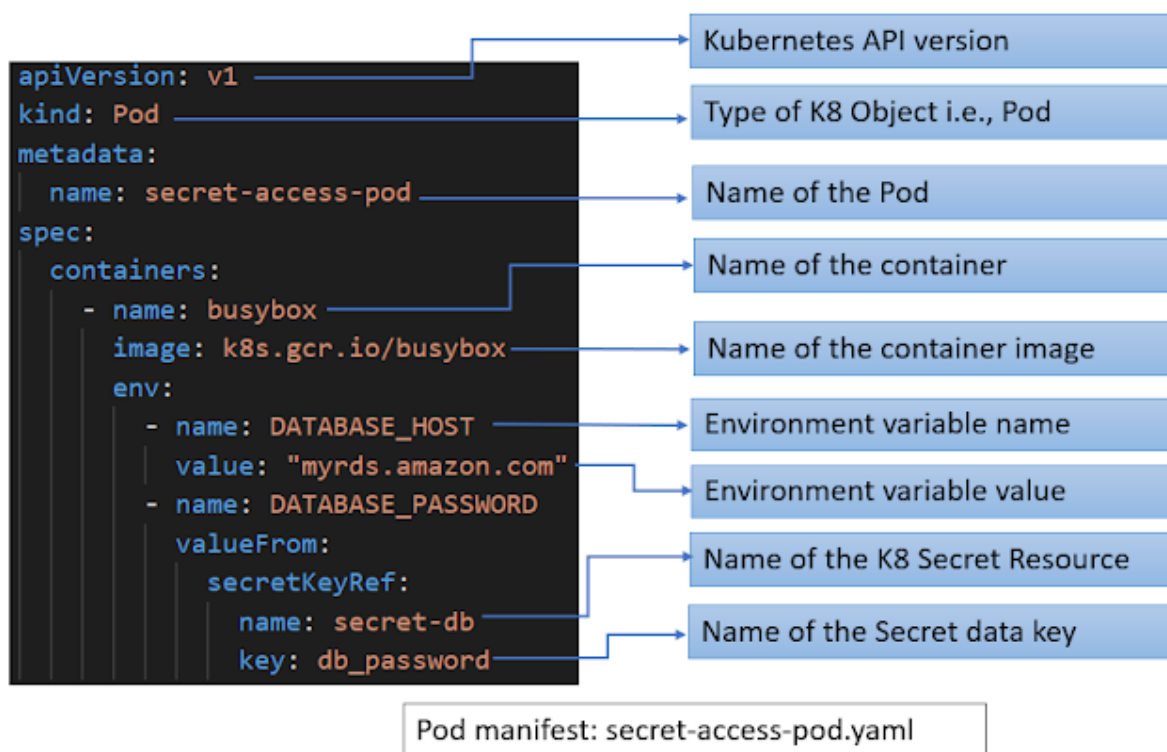
```
// create busybox Pod
$ kubectl apply -f configmap-pod-volume.yaml
pod/configmap-pod-volume created
```

```
// check logs from busybox Pod
$ kubectl logs pod/configmap-pod-volume
db_host
db_name
db_port
```

## 19. Manage app credentials using Secrets

In the previous chapter ([Kubernetes for Developers #18: Manage app settings using Kubernetes ConfigMap](#)) we discussed about Kubernetes Configmap which is used for storing non-sensitive information. Whereas Kubernetes Secrets used for storing application sensitive information i.e., application encryption keys, certificates, database credentials etc.

- It stores data as key-value pair.
- It stored in memory and never saved to physical disk.
- It helps us to separate application specific configurations from the code.
- Each Pod uses Secrets values as environment variables or volumes.
- It supports passing key value as base64 string or plain text.
- Key-value pair must be part of “**data**” or “**stringData**” attribute
- Key name must contain alphanumeric, dash (-), underscore (\_), dot(.) only



### base64 encoded key-value pair

We should use “**data**” attribute if we want to pass all values of keys as base64 encoded strings. This is useful if the key value has binary data. These key values will be converted to plain text automatically when the pod is consuming.

```

apiVersion: v1
kind: Secret
metadata:
  name: secret-db
data:
  db_name: "cmFtYQo="
  db_password: "cGFzc3dvcmQK"

```

### Plain text key-value pair

We should use “stringData” attribute if we want to pass all values of keys as plain strings. However, internally it converted to based64 encoded string while creating.

```

apiVersion: v1
kind: Secret
metadata:
  name: secret-db
stringData:
  db_name: "rama"
  db_password: "password"

```

### Types of Secret

Kubernetes provides different built-in secret types for handling common use cases. We use “Opaque” type for storing application related credentials.

Type	Description
Opaque	used for creating user-defined data
kubernetes.io/service-account-token	used for mapping service account token
kubernetes.io/dockercfg	used for storing credentials for accessing private docker registry images
kubernetes.io/basic-auth	used for storing credentials for basic authentication
kubernetes.io/ssh-auth	Used for storing SSH privatekey for SSH tunneling
kubernetes.io/tls	Used for storing TLS certificate details.

## Create Secret

There are multiple ways to create K8 Secret object

### 1. from YAML file

```
apiVersion: v1
kind: Secret
type: Opaque
metadata:
  name: secret-db
data:
  db_name: "cmFtYQo="
  db_password: "cGFzc3dvcmQK"
```

As we are using **data** attribute, we should pass key values in the base64 encoded string format

Save above yaml content as "secret-db.yaml" and run following kubectl command

```
// create secret object from yaml file
$ kubectl apply -f secret-db.yaml
secret/secret-db created
```

```
// display all secret objects
$ kubectl get secret
```

NAME	TYPE	DATA	AGE
secret-db	Opaque	2	18s

```
// view secret object details using describe command
$ kubectl describe secret secret-db
```

```
// view secret object details as yaml file
$ kubectl get secret secret-db -o yaml
```

```
// decode base64 string
$ echo "cmFtYQo=" | base64 --decode
rama
```

## 2. from literals

In this approach, we pass key-value pair as part of kubectl command directly.

```
// create secret object from literals
// syntax
// kubectl create secret <secret-type> <secret-name> --from-
literal=<key_name>=<key_value>

// we should use "generic" for "Opaque" secret type
$ kubectl create secret generic secret-db-test --from-
literal=db_name=rama --from-literal=db_password=password
secret/secret-db-test created
```

```
// display all secret objects
$ kubectl get secret
```

NAME	TYPE	DATA	AGE
secret-db-test	Opaque	2	6s

```
// view secret object details using describe command
$ kubectl describe secret secret-db-test

// view secret object details as yaml file
$ kubectl get secret secret-db-test -o yaml
```

## Passing Secret values to Pod

### 1. passing Secret values to container environment variables

In the below example, we are setting Pod container individual environment variables from “secret-db” object

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-pod
spec:
  containers:
    - name: busybox
      image: k8s.gcr.io/busybox
      command: ["/bin/sh", "-c", "env"]
```



```
env:
  - name: DATABASE_HOST
    value: "myrds.amazon.com"
  - name: DATABASE_USER_NAME
    valueFrom:
      secretKeyRef:
        name: secret-db
        key: db_name
  - name: DATABASE_PASSWORD
    valueFrom:
      secretKeyRef:
        name: secret-db
        key: db_password
```

save above yaml content as "secret-pod-valuefrom.yaml" and run following kubectl command

```
// create busybox Pod
$ kubectl apply -f secret-pod-valuefrom.yaml
pod/secret-pod created
```

```
// check environment values from busybox Pod
// secret values will be decoded to plaintext automatically when Pod is
consuming
$ kubectl logs pod/secret-pod | grep DATABASE
DATABASE_USER_NAME=rama
DATABASE_PASSWORD=password
DATABASE_HOST=myrds.amazon.com
```

## 2. passing complete Secret Object as environment variables to the container

In the below example, we are passing all "secret-db" object key-value pairs as container environment variables.

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-pod-env
spec:
  containers:
    - name: busybox
      image: k8s.gcr.io/busybox
      command: ["/bin/sh", "-c", "env"]
```

```
envFrom:
  - secretRef:
      name: secret-db
```

save above yaml content as "configmap-pod-envfrom.yaml" and run following kubectl command

```
// create busybox Pod
$ kubectl apply -f secret-pod-envfrom.yaml
pod/secret-pod-env created
```

```
// check environment values from busybox Pod
$ kubectl logs pod/secret-pod-env | grep db
db_password=password
db_name=rama
```

### 3. Attaching Secret as container volume

In the below example,

- we are first creating **volumes** by setting secretName property as "secret-db"
- we are accessing this volume by setting **volumeMounts** property

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-pod-volume
spec:
  containers:
    - name: busybox
      image: k8s.gcr.io/busybox
      command: ["/bin/sh", "-c", "ls /etc/secret/"]
      volumeMounts:
        - name: secret-volume
          mountPath: /etc/secret
  volumes:
    - name: secret-volume
      secret:
        secretName: secret-db
```

save above yaml content as "secret-pod-volume.yaml" and run following kubectl command

```
// create busybox Pod
```

```
$ kubectl apply -f secret-pod-volume.yaml
pod/secret-pod-volume created

// check logs from busybox Pod
$ kubectl logs pod/secret-pod-volume
db_name
db_password
```

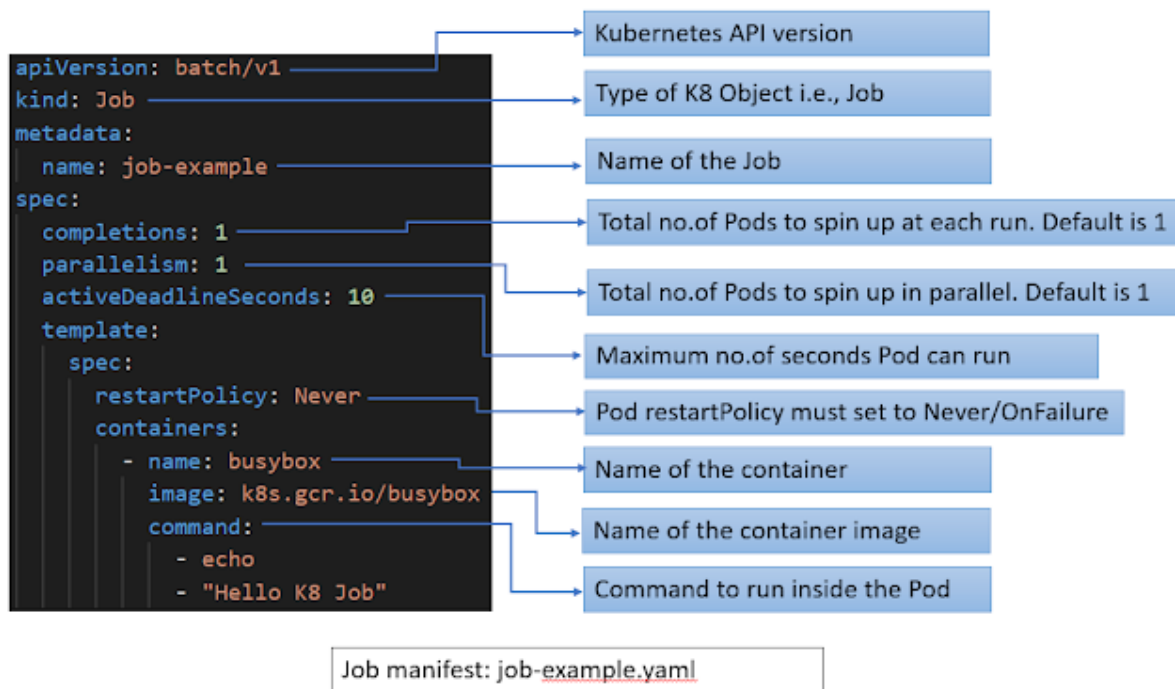
---

## 20. Create Automated Tasks using Jobs and CronJobs

In general, we use K8 Deployment object for creating Pods and running continuously without stopping. However, there are cases where we want to run a task and terminate once it is completed. such as data backups, exporting logs, batch process, sending email etc.

Kubernetes Job object is best suitable to do these kinds of tasks. It creates a Pod for given task and terminates successfully once task is completed.

- Kubernetes Job creates a Pod to run the task
- Pod should be stopped once the task is completed and must not run always as K8 Deployment. Hence, Pod **restartPolicy** must be set to “Never” or “OnFailure”
- Kubernetes Job can create multiple Pods by configuring parallelism
- Job will be scheduled to another worker node automatically in case of current worker node failure
- Set Job “**completions**” attribute to create total number of pods at each run. Default value is 1
- Set Job “**parallelism**” attribute to spin up total number of pods in parallel at each run. Default value is 1
- Set Job “**activeDeadlineSeconds**” attribute to specify how long Job should wait for the Pod to finish. Pod will be terminated automatically if it is running beyond specified time.



### Create Job with Single Pod

As per below yaml, single Pod will be triggered as we are setting completions and parallelism attribute values to 1

```
apiVersion: batch/v1
kind: Job
metadata:
  name: job-example
spec:
  completions: 1
  parallelism: 1
  activeDeadlineSeconds: 10
  template:
    spec:
      restartPolicy: Never
      containers:
        - name: busybox
          image: k8s.gcr.io/busybox
          command:
            - echo
            - "Hello K8 Job"
```

Save above yaml content as "job.yaml" and run following kubectl command

```
// create job object from yaml file
$ kubectl apply -f job.yaml
job.batch/job-example created
```

```
// display all job objects
$ kubectl get jobs
NAME                COMPLETIONS  DURATION  AGE
job-example         1/1           6s        1m
```

```
// check the pod which is created by job
$ kubectl get po
NAME                READY  STATUS      RESTARTS  AGE
job-example-pdchw   0/1    Completed   0          1m
```

```
// view logs from the pod
$ kubectl logs job-example-pdchw
Hello K8 Job
```

```
// delete the job
$ kubectl delete job/job-example
job.batch "job-example" deleted
```

### Create Job with Multiple Pods Sequentially

As per below yaml, multiple Pods gets triggered sequentially one after another (i.e after completion of each Pod) until it reaches to 3 as we are setting **completions: 3**

First creates one pod, and when the pod completes, it creates second pod and so on, until it completes all 3 pods.

```
apiVersion: batch/v1
kind: Job
metadata:
```

```

  name: job-example-sequential
spec:
  completions: 3
  parallelism: 1
  template:
    spec:
      restartPolicy: Never
      containers:
      - name: busybox
        image: k8s.gcr.io/busybox
        command:
        - echo
        - "Hello K8 Job"

```

Save above yaml content as "job-sequential.yaml" and run following kubectl command

```

// create job object from yaml file
$ kubectl apply -f job-sequential.yaml
job.batch/job-example-sequential created

```

```

// display all job objects
$ kubectl get job

```

NAME	COMPLETIONS	DURATION	AGE
job-example-sequential	1/3	18s	25s

```

// 2nd Pod is creating after completion of 1st Pod
$ kubectl get pod

```

NAME	READY	STATUS	RESTARTS
pod/job-example-sequential-h15zn	0/1	ContainerCreating	0
pod/job-example-sequential-srf62	0/1	Completed	0

```

// 3rd Pod is creating after completion of 2nd Pod
$ kubectl get pod

```

NAME	READY	STATUS	RESTARTS
pod/job-example-sequential-671zn	0/1	ContainerCreating	0

```
pod/job-example-sequential-
h15zn      0/1      Completed      0      11s
pod/job-example-sequential-
srf62      0/1      Completed      0      17s
```

```
// All three Pods are completed successfully
$ kubectl get pod
NAME                                     READY   STATUS    RESTARTS
AGE
pod/job-example-sequential-
671zn      0/1      Completed      0      5s
pod/job-example-sequential-
h15zn      0/1      Completed      0     13s
pod/job-example-sequential-
srf62      0/1      Completed      0     19s
```

### Create Job with Multiple Pods in Parallel

As per below yaml, two Pods gets triggered in parallel as we are setting **parallelism** : 2

First creates two pods in parallel, and when any one pod completes, it creates third pod and so on, until it completes all 3 pods.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: job-example-parallel
spec:
  completions: 3
  parallelism: 2
  template:
    spec:
      restartPolicy: Never
      containers:
      - name: busybox
        image: k8s.gcr.io/busybox
        command:
        - echo
        - "Hello K8 Job"
```

Save above yaml content as "job-parallel.yaml" and run following kubectl command

```
// create job object from yaml file
$ kubectl apply -f job-parallel.yaml
job.batch/job-example-parallel created
```

```
// display all job objects
$ kubectl get job
NAME                                COMPLETIONS  DURATION  AGE
job-example-parallel              2/3           18s       25s
```

```
// Two Pods are triggered in parallel at same time
$ kubectl get pod
NAME                                READY  STATUS              RESTARTS  AGE
pod/job-example-parallel-g7jgx     0/1    ContainerCreating   0          1s
pod/job-example-parallel-vjl9w     0/1    ContainerCreating   0          1s
```

```
// 3rd Pod is creating after completion of one of the pod
$ kubectl get pod
NAME                                READY  STATUS              RESTARTS  AGE
pod/job-example-parallel-vjl9w     0/1    Completed           0          21s
pod/job-example-parallel-g7jgx     0/1    Completed           0          21s
pod/job-example-parallel-58v8j     0/1    ContainerCreating   0          11s
```

## Create CronJob

CronJob is used for creating periodic and recurring tasks. It runs a job periodically on a given schedule, written in Cron format.

CronJob creates Job object from the **jobTemplate** property configured in the Cronjob yaml

```
# |----- minute (0 - 59)
# |----- hour (0 - 23)
# |----- day of the month (1 - 31)
# |----- month (1 - 12)
# |----- day of the week (0 - 6) (Sunday to Saturday;
# |                          7 is also Sunday on some systems)
#
# * * * * *
```

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: cronjob-example
spec:
  schedule: "*/2 * * * *"
  jobTemplate:
```



```
spec:
  completions: 1
  parallelism: 1
  template:
    spec:
      restartPolicy: Never
      containers:
      - name: busybox
        image: k8s.gcr.io/busybox
        command:
        - echo
        - "Hello K8 Job"
```

As per above yaml, job is scheduled for every 2 minutes.

```
// create cronjob object from yaml file
$ kubectl apply -f cronjob.yaml
cronjob.batch/cronjob-example created
```

```
// display all cronjob objects
$ kubectl get cronjob
```

NAME	SCHEDULE	SUSPEND	ACTIVE	LAST SCHEDULE	AGE
cronjob-example	*/2 * * * *	False	0	11s	3m6s

```
// display pods created by cronjob
$ kubectl get pod
```

NAME	READY	STATUS	RESTARTS
pod/cronjob-example-27119250-khqs8	0/1	Completed	0
pod/cronjob-example-27119251-4c5kc	0/1	Completed	0

---

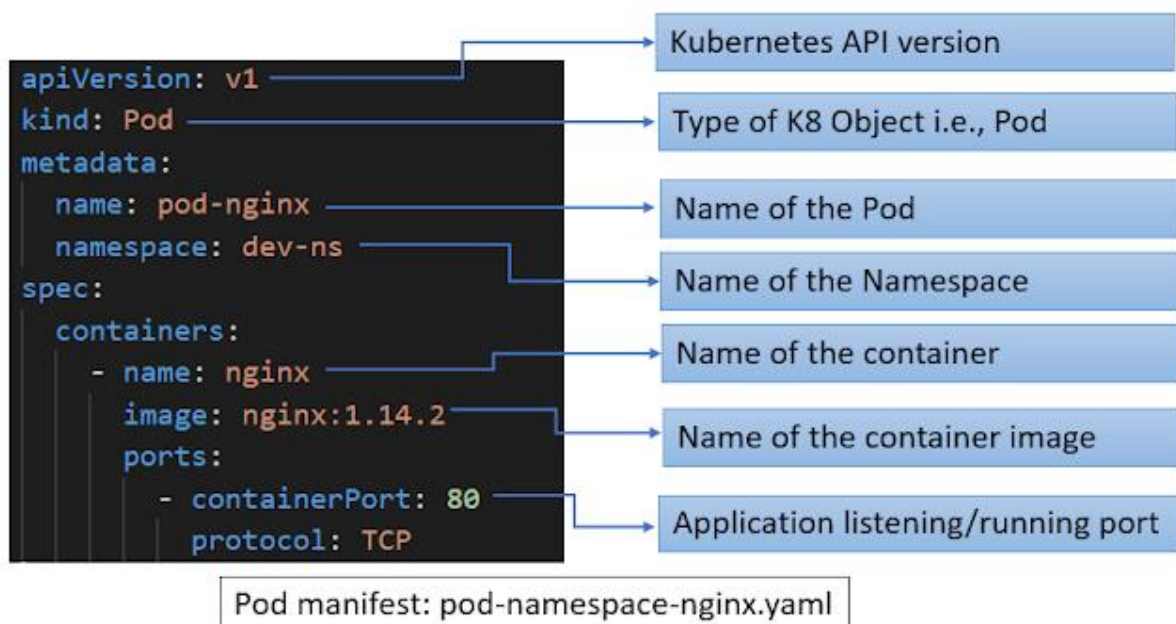
## 21. Kubernetes Namespace in-detail

Kubernetes supports multiple virtual clusters by using Namespaces. It helps when multiple teams using same cluster and want to setup separate Roles, Binding and Environments for each team.

- Kubernetes resource name should be unique within a namespace, but not across namespaces. It means, we can use same Pod/Deployment name in different namespaces
- Namespaces cannot be nested
- Each Kubernetes resource can only be in one namespace. It means, we cannot attach same Pod/Deployment to multiple namespaces
- Try to avoid creating namespace with prefix "kube-" as it is reserved for K8 internal system
- We can restrict resource usage limit (i.e. CPU, Memory) for each namespace
- We can restrict users to access only selected namespaces and it's Kubernetes objects
- By default, all the k8 objects created under "default" namespace

The following four namespaces will be created automatically when the cluster is configured.

Namespace	description
default	By default, all the resources created under "default" namespace without CPU and Memory restrictions
kube-system	This namespace for objects created by the Kubernetes system
kube-public	It is reserved for cluster usage, and it is accessible for all users. Use this only when k8 objects should be visible to publicly throughout the cluster
kube-node-lease	It is for lease objects associated with each node to improve performance of the node heartbeats



## Create Namespace

### 1. from YAML file

```
apiVersion: v1
kind: Namespace
metadata:
  name: dev-ns
```

Save above yaml content as "dev-ns.yaml" and run following kubectl command

```
// create namespace resource from yaml file
$ kubectl apply -f dev-ns.yaml
namespace/dev-ns created
```

```
// display all namespaces
$ kubectl get ns
NAME                STATUS    AGE
default             Active    30d
kube-node-lease     Active    30d
kube-public         Active    30d
kube-system         Active    30d
dev-ns              Active    17m
```

```
// view namespace details using describe command
$ kubectl describe ns dev-ns
```

```
// view namespace details as yaml file
$ kubectl get ns dev-ns -o yaml
```

## 2. imperative way

```
// syntax $kubectl create ns <namespace-name>
$ kubectl create ns dev2-ns
namespace/dev2-ns created

// display all namespaces
$ kubectl get ns
```

NAME	STATUS	AGE
default	Active	30d
kube-node-lease	Active	30d
kube-public	Active	30d
kube-system	Active	30d
dev-ns	Active	17m
dev2-ns	Active	1m

## Create Pod in selected Namespace

### 1. from YAML file

we can achieve this by specifying **namespace** attribute in the metadata section

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-nginx
  namespace: dev-ns
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
```

Save above yaml content as "pod-dev-ns.yaml" and run following kubectl command

```
// create pod under "dev-ns" namespace using yaml file
$ kubectl apply -f pod-dev-ns.yaml
pod/pod-nginx created
```

```
// display all pods from dev-ns namespace
// syntax
$ kubectl get po -n <namespace-name>

$ kubectl get po -n dev-ns
NAME          READY   STATUS    RESTARTS   AGE
pod-nginx     1/1     Running   0           4m4s
```

```
// delete namespace
// all pods gets deleted automatically when namespace is deleted
$ kubectl delete ns dev-ns
namespace "dev-ns" deleted

// display all pods under dev-ns namespace
$ kubectl get po -n dev-ns
No resources found in dev-ns namespace.

// delete all pods without deleting namespace
//syntax:
kubectl delete po --all -n <namespace-name>

$ kubectl delete po --all -n dev2-ns
pod "pod-nginx" deleted

// display all namespaces
$ kubectl get ns
NAME           STATUS    AGE
default        Active    31d
dev2-ns        Active    47m
kube-node-lease Active    31d
kube-public    Active    31d
kube-system    Active    31d
```

## 2. imperative way

```
// syntax
// kubectl run <pod-name> --image <image-name> -n <namespace-name>

$ kubectl run pod-nginx --image nginx -n dev2-ns
pod/pod-nginx created
```

```
// display all pods from dev2-ns namespace
// syntax
kubectl get po -n <namespace-name>
```

```
$ kubectl get po -n dev2-ns
```

NAME	READY	STATUS	RESTARTS	AGE
pod-nginx	1/1	Running	0	4m4s

we must use Fully Qualified Domain Name (FQDN) to access Pods from one namespace to another namespace i.e.

<servicename>.<namespace>.svc.cluster.local

---

## 22. Access to Multiple Clusters or Namespaces using kubectl and kubeconfig

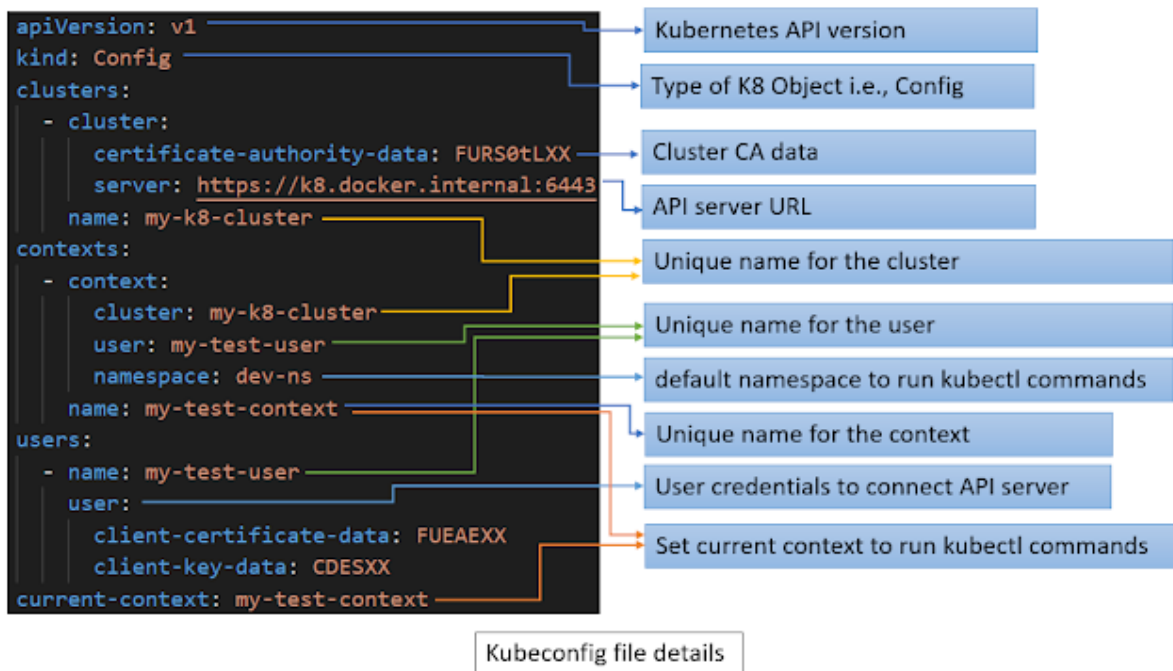
In the previous chapter ([Kubernetes for Developers #21: Kubernetes Namespace in-detail](#)), we added name of the namespace in all kubectl commands to create or display kubernetes resources. Because kubeconfig current context referring to **default** namespace.

### Kubeconfig file

**kubeconfig** file gets created automatically when you install k8 in the machine. It is usually located in the `~/.kube/config`. This file contains all the necessary information to connect k8 cluster. kubectl CLI uses this file to execute commands against k8 cluster.

kubeconfig file contains four major sections

1. **clusters**: it contains list of cluster details like name of the cluster, API server URL, Certificate Authority (CA) file, etc.
2. **users**: it contains list of user credentials (like name, password, user token, certificate) for connecting an API server
3. **contexts**: it contains combination of cluster name, username and namespace. It is used for kubectl CLI when executing commands against k8
4. **current-context**: Always one combination of context (i.e. name of cluster, user, namespace) must be set as current context. (i.e. cluster name + user name + namespace name)



run following command to view kubeconfig file details

```
$ kubectl config view
```

```
apiVersion: v1
kind: Config
clusters:
- cluster:
  certificate-authority-data: FURS0tLXX
  server: https://k8.docker.internal:6443
  name: my-k8-cluster
contexts:
- context:
  cluster: my-k8-cluster
  user: my-test-user
  namespace: dev-ns
  name: my-test-context
users:
- name: my-test-user
  user:
  client-certificate-data: FUEAEXX
  client-key-data: CDESXX
current-context: my-test-context
```

```
// display all contexts
$ kubectl config get-contexts
```

CURRENT	NAME	CLUSTER	AUTHINFO	NAMESPACE
*	my-test-context	my-k8-cluster	my-test-user	dev-ns

```
// display current context
$ kubectl config current-context
my-test-context
```

The following kubectl command display all pods from **dev-ns** namespace as current-context (i.e. my-test-context) referring to **dev-ns** namespace

```
// display pods
$ kubectl get po
```

NAME	READY	STATUS	RESTARTS	AGE
pod-nginx-dev-ns	1/1	Running	0	4m4s

run following command to modify namespace for the context

```
// modify namespace for the given context
$ kubectl config set-context my-test-context --namespace=dev2-ns
Context "my-test-context" modified.
```

```
// the below command display all pods from the dev2-ns namespace
$ kubectl get po
```

NAME	READY	STATUS	RESTARTS	AGE
pod-nginx-dev2-ns	1/1	Running	0	1m4s

## Create new Context

As said earlier, every context consists of three parts

### 1. Cluster details:

use following kubectl command to configure new cluster details

```
// syntax
// kubectl config set-cluster <cluster-map-name> --server=<cluster-url> --
certificate-authority=<ca-file>
```



```
$ kubectl config set-cluster my-new-cluster \  
  --server=https://mycluster.k8.com:6443 \  
  --certificate-authority=/path/cafile  
Cluster "my-new-cluster" set.
```

## 2. User details:

use following kubectl command to configure new user details

```
// syntax  
// kubectl config set-credentials <user-map-name> --token=<user-token-  
info> (or)  
// kubectl config set-credentials <user-map-name> --username=<username> --  
password=<password>  
$ kubectl config set-credentials my-new-user --token=token1234abcd  
User "my-new-user" set.
```

## 3. Namespace details:

use following kubectl command to view namespace details

```
// display all namespaces  
$ kubectl get ns  
NAME           STATUS    AGE  
default        Active    31d  
dev2-ns        Active    47m  
dev-ns         Active    47m
```

Use following kubectl command to combine all above three details for **creating new context**

```
// syntax  
// kubectl config set-context <context-name> \  
  --cluster=<cluster-name> \  
  --user=<user-name> \  
  --namespace=<namespace-name>  
$ kubectl config set-context my-new-context \  
  --cluster=my-new-cluster \  
  --user=my-new-user \  
  --namespace=default
```

```
--user=my-new-user \  
--namespace=dev-ns  
Context "my-new-context" created.
```

```
//display all contexts  
$ kubectl config get-contexts  
CURRENT  NAME                CLUSTER             AUTHINFO            NAMESPACE  
*         docker-desktop      docker-desktop      docker-desktop      dev-ns  
          my-new-context     my-new-cluster      my-new-user  
  
// display current context  
$ kubectl config current-context  
docker-desktop
```

Use following kubectl command to switch the context

```
// change the context  
$ kubectl config use-context my-new-context  
Switched to context "my-new-context".
```

By default, the following kubectl command display all the pods from **dev-ns** namespace because CLI using "my-new-context" as current-context

```
// display pods  
$ kubectl get po  
NAME                READY   STATUS    RESTARTS   AGE  
pod-nginx-dev-ns    1/1     Running   0           4m4s
```

Use following kubectl command to delete the context

```
// delete context  
$ kubectl config delete-context my-new-context  
deleted context my-new-context from ~/.kube/config
```

---

## 23. Kubernetes Volume emptyDir in-detail

Containers are ephemeral. It means, any container generated data gets stored into its own filesystem and will be deleted automatically if the container is deleted or restarted.

In Docker world, docker volumes provide a way to store container data into the host machine as permanent storage. However, it is less managed and limited for multi-node cluster.

Kubernetes volumes provide a way for containers to access external disk storage or share storage among containers.

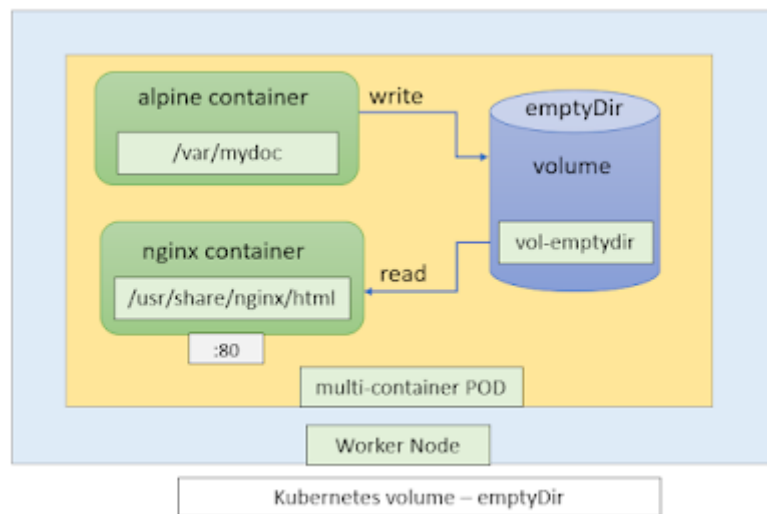
Kubernetes volumes are not top-level object as like Pod, Deployment etc., however these are component of a pod and defined as part of pod YAML specification. K8 Volumes are available to all containers in the pod and must be mounted in each container specific file location.

Kubernetes supports many types of volumes like,

- **emptyDir**: Used for mounting temporary empty directory from worker node Disk/RAM
- **awsElasticBlockStore**: Used for mounting AWS EBS volume into the pod
- **azureDisk**: Used for mounting Microsoft Azure data disk into the pod
- **azureFile**: Used for mounting Microsoft Azure File volume into the pod
- **gcePersistentDisk**: Used for mounting Google PD into the pod
- **hostPath**: Used for mounting Worker node filesystem into the pod
- **nfs**: Used for mounting existing NFS (Network file system) into the pod
- **configMap/secret**: Used for mounting these values into the pod
- **persistentVolumeClaim**: Used for mounting dynamically provisioned storage into the pod

A Pod can use any number of volume types simultaneously to persist container data.

## emptyDir volume



An empty directory created when a Pod is assigned to a node and remains active until pod is running. All containers in the pod can read/write the contents to the emptyDir volume. An emptyDir volume will be erased automatically once the pod is terminated from the node.

A container crashing does not remove a Pod from a node. The data in an emptyDir volume is safe across container crashes. It only erased when Pod is deleted from the node.

- It is useful for sharing files between containers which are running in the same pod
- It is useful for doing disk-based merge sort on large dataset where memory is low
- It is useful when container filesystem is read-only and wants to write data temporarily.

```
apiVersion: v1
kind: Pod
metadata:
  name: volume-emptydir
spec:
  containers:
  - name: alpine
    image: alpine
    command:
    [
      "sh",
      "-c",
      'mkdir var/mydoc; while true; do echo "random message text `date`" >> var/mydoc/index.html;sleep 10;done',
    ]
  volumeMounts:
  - name: vol-emptydir
    mountPath: /var/mydoc
```

```

- name: nginx
  image: nginx:alpine
  ports:
    - containerPort: 80
      protocol: TCP
  volumeMounts:
    - name: vol-emptydir
      mountPath: /usr/share/nginx/html
volumes:
- name: vol-emptydir
  emptyDir: {}

```

As per above yaml ,

1. A multi-container pod gets created with volume type “emptyDir” named as “vol-emptydir”
2. “vol-emptydir” volume gets created automatically when a pod is assigned to a worker-node.
3. As name says, volume contains empty files/directories at initial stage.
4. First “alpine” container creates random text message for every 10 seconds and appends to /var/mydoc/index.html file.
5. First “alpine” container mounted a volume at ‘/var/mydoc’. So, all the files under this directory copied into volume (i.e., index.html file).
6. Second “nginx” container mounted a same volume at ‘/usr/share/nginx/html’ (this is the default directory for nginx to serve index.html file). As we mounted same volume which has “index.html”, nginx web server serves the file (i.e., index.html) which is created by the first container.
7. As first container adds new random message to index.html file for every 10 seconds, we see different message each time when we request index.html from nginx webserver.
8. Volume and its contents get deleted automatically when the Pod is deleted
9. By default, Volume contents get stored on the worker node disk. However, “emptyDir” volume contents can be stored into the memory (RAM) by setting “medium” attribute

save above yaml content as "**pod-vol-emptydir.yaml**" and run the following kubectl command

```

// create pod
$ kubectl apply -f pod-vol-emptydir.yaml
pod/pod-vol-emptydir created

```

```
// display pods
$ kubectl get po
NAME                READY   STATUS    RESTARTS   AGE
pod-vol-emptydir    2/2     Running   0           2m39s
```

run the following kubectl command to forward a port from local machine to the pod

```
// syntax
// kubectl port-forward <pod-name> <local-port>:<container-port>
$ kubectl port-forward pod-vol-emptydir 8081:80
Forwarding from 127.0.0.1:8081 -> 80
Forwarding from [::1]:8081 -> 80
```

run the following curl command to check random messages which are appending after every 10 seconds

```
$ curl http://localhost:8081
random message text Tue Nov  2 15:48:50 UTC 2021
```

```
$ curl http://localhost:8081
random message text Tue Nov  2 15:48:50 UTC 2021
random message text Tue Nov  2 15:49:00 UTC 2021
random message text Tue Nov  2 15:49:10 UTC 2021
```

An emptyDir volume does not persist data after pod termination. So, delete the Pod and recreate all above steps to check any existing data is printing while doing curl command

```
// delete pod
$ kubectl delete pod/pod-vol-emptydir
pod/pod-vol-emptydir deleted

// create pod
$ kubectl apply -f pod-vol-emptydir.yaml
pod/pod-vol-emptydir created
```

```
// display pods
$ kubectl get po
NAME                READY   STATUS    RESTARTS   AGE
pod-vol-emptydir    2/2     Running   0           1m10s

// syntax
// kubectl port-forward <pod-name> <local-port>:<container-port>
$ kubectl port-forward pod-vol-emptydir 8081:80
Forwarding from 127.0.0.1:8081 -> 80
Forwarding from [::1]:8081 -> 80

$ curl http://localhost:8081
random message text Tue Nov  2 16:01:50 UTC 2021
```

It is confirmed that only new data is printing after pod recreation when using emptyDir volume.

By default, Volume contents get stored on the worker node disk. However, “emptyDir” volume contents can be stored into the memory (RAM) by setting “medium” attribute

```
volumes:
- name: vol-emptydir
  emptyDir:
    medium: Memory.
```

---

## 24. Kubernetes Volume hostPath in-detail

In the previous chapter ([Kubernetes for Developers #23: Kubernetes Volume emptyDir in-detail](#)), we discussed about emptyDir volume for storing and sharing data among multiple/single container(s) in the pod. However, emptyDir volume and its contents get deleted automatically when the Pod is deleted from the worker node.

Kubernetes hostPath volume helps us to persist volume contents even after pod deleted from the worker node.

K8 hostPath volume mounts a file or directory from the worker node filesystem into the pod.

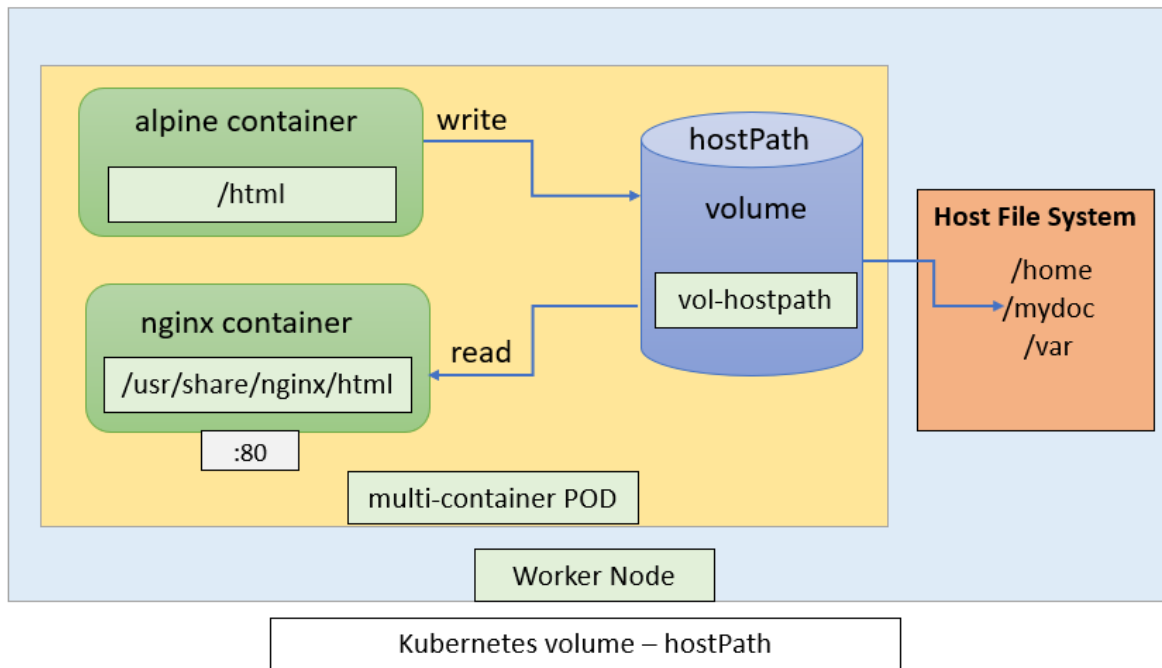
A pod running on the same worker node can only mount to the file/directory of that node.

- It is useful when the container wants to access docker system files from the host (i.e., /var/lib/docker)
- It is useful when the container needs to access kubeconfig file (or) CA certificates (or) /var/logs from the host
- It is useful when the container needs to access host /sys files for cAdvisor
- It is useful when the container wants to check given path exists in the host before running

Kubernetes hostPath volume supports following types while mounting

Type	Description
Directory	A directory must exist in the specified path on the host
DirectoryOrCreate	An empty directory will be created when the specified path does not exist on the host
File	A file must exist in the specified path on the host
FileOrCreate	An empty file will be created when the specified path does not exist on the host
Socket	A UNIX socket must exist in the specified path





```

apiVersion: v1
kind: Pod
metadata:
  name: pod-vol-hostpath
spec:
  containers:
    - name: alpine
      image: alpine
      command:
        [
          "sh",
          "-c",
          'while true; do echo "random message text `date`" >>
html/index.html;sleep 10;done',
        ]
      volumeMounts:
        - name: vol-hostpath
          mountPath: /html
    - name: nginx
      image: nginx:alpine
      ports:
        - containerPort: 80
          protocol: TCP
      volumeMounts:
        - name: vol-hostpath
          mountPath: /usr/share/nginx/html
  volumes:
    - name: vol-hostpath

```

```
hostPath:
  path: /mydoc
  type: DirectoryOrCreate
```

As per above yaml ,

1. A multi-container pod gets created with volume type "hostPath" named as "vol-hostpath" and mounted on "/mydoc" directory from the host filesystem
2. "mydoc" directory gets created automatically when a pod is assigned to a worker-node if not exists on the host filesystem as we specified volume type "DirectoryOrCreate"
3. First "alpine" container creates random text message for every 10 seconds and appends to /html/index.html file.
4. First "alpine" container mounted a volume at '/html'. So, all the new/modified files under this directory referring to "/mydoc" host filesystem
5. Second "nginx" container mounted a same volume at '/usr/share/nginx/html' (this is the default directory for nginx to serve index.html file ). As we mounted same volume which has "index.html", nginx web server serves the file (i.e., index.html) which is created by the first container.
6. As first container adds new random message to index.html file for every 10 seconds, we see different message each time when we request index.html from nginx webserver.
7. Volume contents won't be deleted on Pod termination. So, whenever the new pod is scheduled on the same node with same hostpath will see all the previous contents.

save above yaml content as "**pod-vol-hostpath.yaml**" and run the following kubectl command

```
// create pod
$ kubectl apply -f pod-vol-hostpath.yaml
pod/pod-vol-hostpath created
```

```
// display pods
$ kubectl get po
NAME                READY   STATUS    RESTARTS   AGE
pod-vol-hostpath    2/2     Running   0           1m10s
```

run the following kubectl command to forward a port from local machine to the pod

```
// syntax
// kubectl port-forward <pod-name> <local-port>:<container-port>
```

```
$ kubectl port-forward pod-vol-hostpath 8081:80
Forwarding from 127.0.0.1:8081 -> 80
Forwarding from [::1]:8081 -> 80
```

run the following curl command to check random messages which are appending after every 10 seconds

```
$ curl http://localhost:8081
random message text Tue Nov 7 12:01:10 UTC 2021
```

```
$ curl http://localhost:8081
random message text Tue Nov 7 12:01:10 UTC 2021
random message text Tue Nov 7 12:01:20 UTC 2021
random message text Tue Nov 7 12:01:30 UTC 2021
```

Volume contents won't be deleted on Pod termination. So, whenever the new pod is scheduled on the same node with same hostpath will see all the previous contents.

delete the Pod and recreate all above steps to check existing data is printing while doing curl command

```
// delete pod
$ kubectl delete pod/pod-vol-hostpath
pod/pod-vol-hostpath deleted

// create pod
$ kubectl apply -f pod-vol-hostpath.yaml
pod/pod-vol-hostpath created

// display pods
$ kubectl get po
```

NAME	READY	STATUS	RESTARTS	AGE
pod-vol-hostpath	2/2	Running	0	1m10s

```
// syntax
// kubectl port-forward <pod-name> <local-port>:<container-port>
$ kubectl port-forward pod-vol-hostpath 8081:80
Forwarding from 127.0.0.1:8081 -> 80
Forwarding from [::1]:8081 -> 80

$ curl http://localhost:8081
random message text Tue Nov 7 12:01:10 UTC 2021
random message text Tue Nov 7 12:01:20 UTC 2021
random message text Tue Nov 7 12:01:30 UTC 2021
random message text Tue Nov 7 14:12:40 UTC 2021

// first 3 lines are generated by the previous pod
```

It is confirmed that curl command showing both previous pod generated contents and new pod contents.

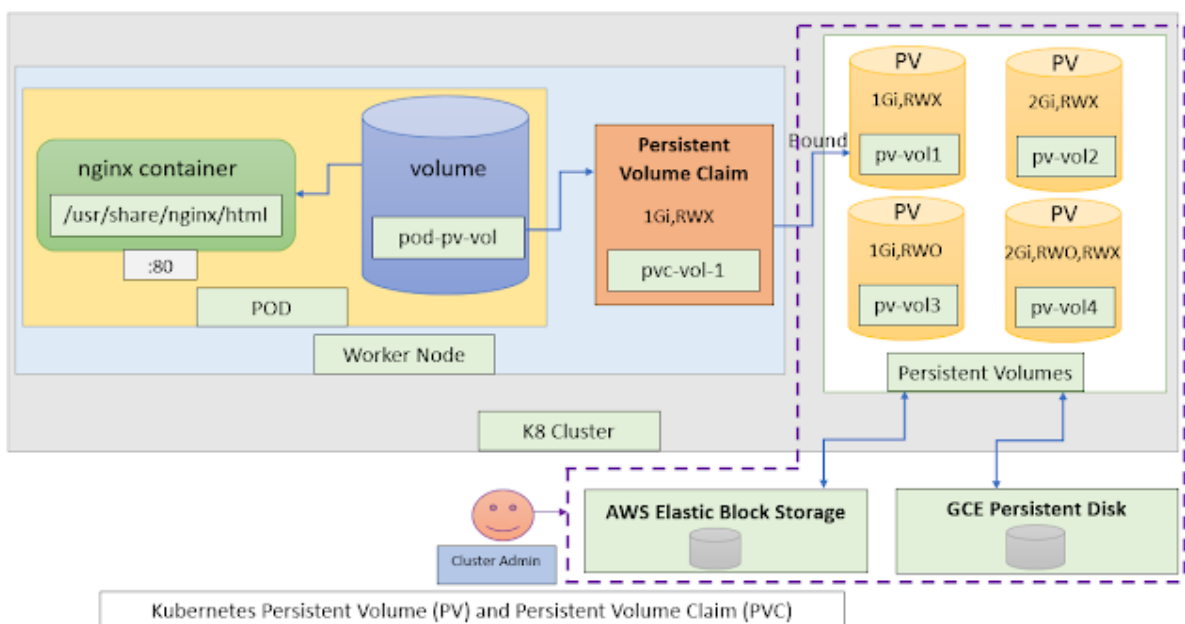
## 25. PersistentVolume and PersistentVolumeClaim in-detail

In the previous chapter ([Kubernetes for Developers #24: Kubernetes Volume hostPath in-detail](#)), we discussed about hostPath volume for persisting container data in the worker node file system. However, this data is available only to the pods which are scheduled on the same worker node. This is not a feasible solution for multi-node cluster.

This problem can be solved by using external storage volumes like awsElasticBlockStore, azureDisk, GCE PD, nfs etc. However, developer must have knowledge on the network storage infrastructure details to use in the pod definition.

It means, when the developer wants to use awsEBS volume in the Pod, the developer should know the details of EBS ID and file type. If there is a change in the storage details, developer must make changes in all the pod definitions.

Kubernetes solves the above problem by using PersistentVolume and PersistentVolumeClaim. It decouples underlying storage details from the application pod definitions. Developers don't have to know the underlying storage infrastructure which is being used. It is more of cluster administrator responsibility.



As per diagram,

- PersistentVolumes(PV) are cluster-level resources like worker nodes. It not belonging to any namespace.

- PersistentVolumeClaims(PVC) can be created in a specific namespace only and it can be used by pods within same namespace only.
- Cluster Administrator sets up cloud storage infrastructure i.e., AWS Elastic Block Storage and GCE Persistent Disk as per the need.
- Cluster Administrator creates Kubernetes PersistentVolumes (PV) with different size and access modes by referring AWS EBS/GCE PD as per application requirements.
- Whenever pod requires persistent storage, Kubernetes Developer creates PersistentVolumeClaim (PVC) with minimum size and access mode, and Kubernetes finds an adequate Persistent Volume with same size and access mode and binds volume (PV) to the claim (PVC).
- Pod refers PersistentVolumeClaim (PVC) as volume whenever it is required.
- Once PersistentVolume is bound to PVC, it cannot be used by others until it is released (i.e., we must delete PVC to reuse PV by others).
- Kubernetes Developer don't have to know the underlying storage details. They just have to create PersistentVolumeClaim (PVC) whenever pod requires persistent storage.

## Access Modes

The following access modes are supported by PersistentVolume(PV)

- **ReadWriteOnce (RWO):** Only single worker node can mount the volume for reading and writing at the same time.
- **ReadOnlyMany (ROX):** Multiple worker nodes can mount the volume for reading at the same time.
- **ReadWriteMany (RWX):** Multiple worker nodes can mount the volume for reading and writing at the same time.

## Reclaim Policy

Reclaim Policy tell us what happens to PersistentVolume(PV) when the PersistentVolumeClaim(PVC) is deleted.

- **Delete:** It deletes volume contents and makes the volume available to be claimed again as soon as PVC is deleted.
- **Retain:** PersistentVolume(PV) contents will be persisted after PVC is deleted and it cannot be re-used until Cluster Administrator reclaim the volume manually.

In general, Cluster Administrator creates multiple PersistentVolumes(PV) by using any one of cloud storages i.e. AWS EBS or GCE PD

```
// Ex: creating aws EBS from cli
$ aws ec2 create-volume \
```

```
--availability-zone=eu-east-1a
--size=10 --volume-type=gp2 ebs-data-id
```

Cluster Administrator creates following PV by using ebs-id

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-vol1
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: ""
  awsElasticBlockStore:
    volumeID: ebs-data-id
    fsType: ext4
```

For local testing, lets use hostPath PersistentVolume. Create a directory called “/mydata” and “index.html” file under “mydata” directory.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-vol1
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: ""
  hostPath:
    path: "/mydata"
```

As per above yaml, Volume configured at “/mydata” host directory with the size of 1Gi and an access mode of “ReadWriteOnce(RWO)”

save above yaml content as “**pv-vol1.yaml**” and run the following kubectl command

```
// create persistentvolume(pv)
$ kubectl apply -f pv-vol1.yaml
persistentvolume/pv-vol1 created
```

```
// display pv
$ kubectl get pv
```

NAME	CAPACITY	ACCESS MODES	RECLAIM POLICY	STATUS
pv-vol1	1Gi	RWO	Retain	Available

Here, status showing "Available". It means, PV is not yet bound to a PersistentVolumeClaim (PVC)

Next step is to create persistentvolumeclaim(pvc) to request physical storage for the pod

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-vol-1
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 500Mi
  storageClassName: ""
```

save above yaml content as "**pvc-vol-1.yaml**" and run the following kubectl command

```
// create pvc
$ kubectl apply -f pvc-vol-1.yaml
persistentvolumeclaim/pvc-vol-1 created
```

```
// display pvc
$ kubectl get pvc
```

NAME	STATUS	VOLUME	CAPACITY	ACCESS MODES
pvc-vol-1	Bound	pv-vol1	1Gi	RWO

Here, PersistentVolumeClaim is bound to PersistentVolume i.e. pv-vol1

Next step is to create a pod to use persistentvolumeclaim as a volume

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-pv-pvc
spec:
  containers:
```



```

- name: nginx
  image: nginx:alpine
  ports:
    - containerPort: 80
      protocol: TCP
  volumeMounts:
    - name: pod-pv-vol
      mountPath: /usr/share/nginx/html
volumes:
- name: pod-pv-vol
  persistentVolumeClaim:
    claimName: pvc-vol-1

```

save above yaml content as "**pod-pv-pvc.yaml**" and run the following kubectl command

```

// create pod
$ kubectl apply -f pod-pv-pvc.yaml
pod/pod-pv-pvc created

```

```

// display pods
$ kubectl get po

```

NAME	READY	STATUS	RESTARTS	AGE
pod-pv-pvc	1/1	Running	0	1m

run the following kubectl command to forward a port from local machine to the pod

```

// syntax
// kubectl port-forward <pod-name> <local-port>:<container-port>
$ kubectl port-forward pod-pv-pvc 8081:80
Forwarding from 127.0.0.1:8081 -> 80
Forwarding from [::1]:8081 -> 80
$ curl http://localhost:8081
text message text Tue Nov 16 12:01:10 UTC 2021

```

We have successfully configured a Pod to use PersistentVolumeClaim as physical storage.

run the following kubectl commands to delete the resources

```

$ kubectl delete pod pod-pv-pvc
$ kubectl delete pvc pvc-vol-1
$ kubectl delete pv pv-vol1

```

---

## 26. Managing Container CPU, Memory Requests and Limits

By default, Kubernetes doesn't put any restrictions on using CPU and Memory for the Pod. It means, a single container in the pod can consume entire node resources. However, this makes other CPU intensive containers will slow down, Kubernetes services may become unresponsive and worker node may go down with NotReady state in worst case scenario.

So, setting up CPU and Memory limits for the containers in the Pod will helps us that only fair share of resources will be allocated by Kubernetes Cluster and will not affect other Pods performance in the Node.

Kubernetes uses following YAML request and limit structure to control container CPU, Memory resources

```
resources:
  requests:
    cpu: 100m
    memory: 50Mi
  limits:
    cpu: 150m
    memory: 100Mi
```

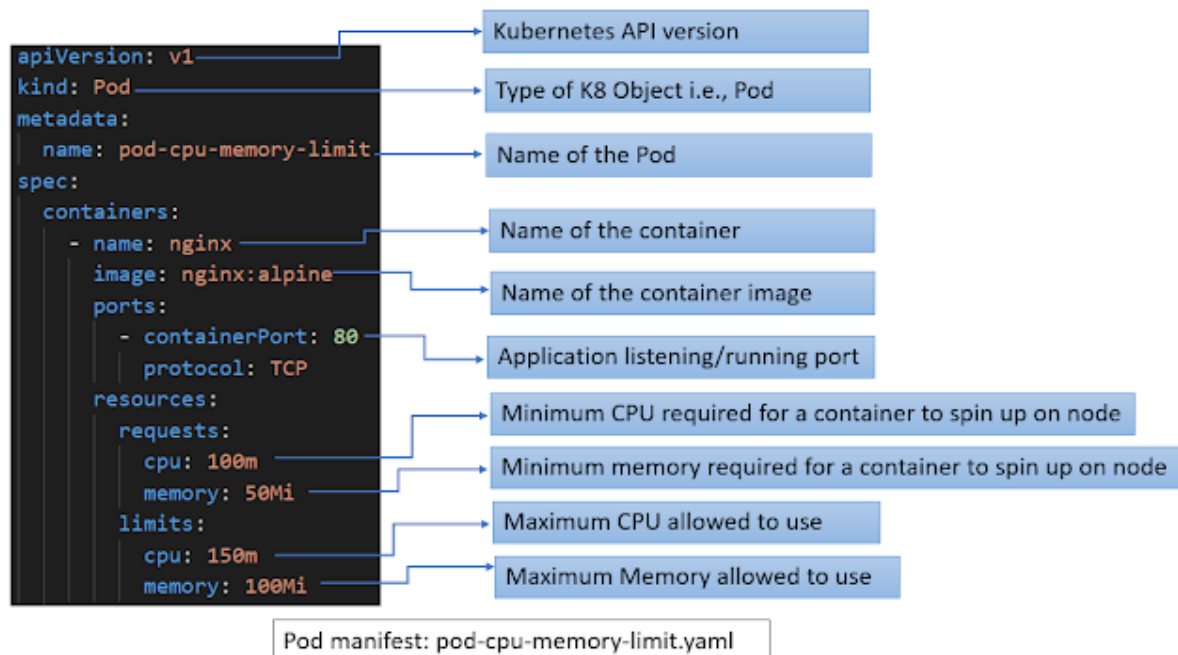
### requests:

- This is the place to specify how much CPU and Memory required for a container. Kubernetes will only schedule it on a node that can give the specified request resources.

### limits:

- This is the place to specify maximum CPU and Memory allowed to use by a single container. So, the running container is not allowed to use more than specified limits.
- limits can never be lower than the requests. If you try this, Kubernetes will throw an error and won't let you run the container.
- CPU is a “**compressible**” resource. It means when a container start hitting max CPU limit, it won't be terminated from the node, but it will throttle it and gives worse performance.
- Memory is a “**non-compressible**” resource. It means when a container start hitting max Memory limit, it will be terminated from the node (Out of Memory killed).

**requests** and **limits** are on a per-container basis. It means, we must specify for each container in the Pod. The pod's resource requests and limits are the sum of the requests and limits of all its containers.



## Kubernetes CPU Resource Units

Limits and requests for CPU resources measured in cpu units. One cpu in Kubernetes equivalent to 1 AWS vCPU (or) 1 GCP Core (or) 1 Azure vCore (or) 1 Hyperthread on a bare-metal processors.

CPU resources can be specified in both fractional and milli cores. i.e., 1 Core = 1000 milli cores

Ex: 0.2 equivalent to 200m

## Kubernetes Memory Resource Units

Limits and requests for Memory are measured in bytes. So, it can be used as plain integer number or suffixing with Mi, Pi, Ti, Gi, Mi, Ki

For example, check the below configuration where container has a request of (0.5 cpu and 200Mi bytes of memory) and limit of (1 cpu and 400Mi bytes of memory)

```
resources:
  requests:
    cpu: 500m
    memory: 200Mi
  limits:
    cpu: 1000m
    memory: 400Mi
```

create following yaml content and save as "pod-cpu-memory-limit.yaml"

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-cpu-memory-limit
spec:
  containers:
    - name: nginx
      image: nginx:alpine
      ports:
        - containerPort: 80
          protocol: TCP
      resources:
        requests:
          cpu: 100m
          memory: 50Mi
        limits:
          cpu: 150m
          memory: 100Mi
    - name: alpine
      image: alpine
      command:
        [
          "sh",
          "-c",
          "while true; do echo date;sleep 10;done"
        ]
      resources:
        requests:
          cpu: 50m
          memory: 30Mi
        limits:
          cpu: 60m
          memory: 50Mi
```

The above pod has two containers.

- First container (i.e. nginx) has request of 100m or 0.1 CPU and 50Mi memory, max limit of 150m CPU and 100Mi
- Second container (i.e. alpine) has request of 50m CPU and 30Mi memory, max limit of 60m CPU and 50Mi

So, Pod has total request of 150m CPU and 80Mi of memory, total max limit of 210m CPU and 150Mi of memory.

run the following query to get node **capacity**(i.e. total cpu and memory of the node) and **allocatable**(i.e. total resources allocatable for pods by the scheduler)

```
// check all worker nodes capacity
$ kubectl describe nodes
Capacity:
  cpu:          4
  memory:       7118488Ki
Allocatable:
  cpu:          4
  memory:       7016088Ki
```

```
// create pod
$ kubectl apply -f pod-cpu-memory-limit.yaml
pod/pod-cpu-memory-limit created
```

```
// display pods
$ kubectl get po
NAME                                READY   STATUS    RESTARTS
pod-cpu-memory-limit               2/2     Running   0
```

```
// view CPU and Memory limits for all containers in the Pod
$ kubectl describe pod/pod-cpu-memory-limit
Name:          pod-cpu-memory-limit
Namespace:     default
Containers:
  nginx:
    Image:      nginx:alpine
    Limits:
      cpu:      150m
      memory:   100Mi
    Requests:
      cpu:      100m
      memory:   50Mi
  alpine:
    Image:      alpine
    Limits:
      cpu:      60m
      memory:   50Mi
    Requests:
      cpu:      50m
      memory:   30Mi
```

```
// Pod will be in Pending state when we specify requests are bigger than node capacity
```

```
// Create Pod using imperative style
$ kubectl run requests-bigger-pod --image=busybox--restart Never \
--requests='cpu=8000m,memory=200Mi'
```

```
// display pods
$ kubectl get po
NAME                READY   STATUS    RESTARTS
requests-bigger-pod  0/1     Pending   0

// Pod in pending status due to insufficient CPU. So, check Pod details
$ kubectl describe pod/requests-bigger-pod
Name:                requests-bigger-pod
Namespace:           default
Events:
  Type      Reason             Age   From                  Message
  ----      -
Warning    FailedScheduling   28s   default-scheduler    0/1 nodes are available:
1 Insufficient cpu.
```

```
// Pod created successfully when limits are bigger than node capacity

// Create Pod using imperative style
$ kubectl run limits-bigger-pod --image=busybox --restart Never \
--requests='cpu=100m,memory=50Mi' \
--limits='cpu=8000m,memory=200Mi'
```

```
// If you specify limits but do not specify requests then k8 creates
requests
which is equal to limits

$ kubectl run no-requests-pod --image=busybox --restart Never \
--limits='cpu=100m,memory=50Mi'

// view CPU and Memory limits
$ kubectl describe pod/no-requests-pod
Name:                no-requests-pod
Namespace:           default
Containers:
  busybox:
    Image:            nginx:alpine
    Limits:
      cpu:            100m
      memory:         50Mi
    Requests:
      cpu:            100m
      memory:         50Mi
```

---

## 27. Configure LimitRange for setting default Memory/CPU for a Pod

In the previous article ([Kubernetes for Developers #26: Managing Container CPU, Memory Requests and Limits](#)), we have successfully configured requests and limits for containers in the pod. However, there is a possibility for developers to forget setting up resources and eventually containers may consume more than fair share of resources in the cluster.

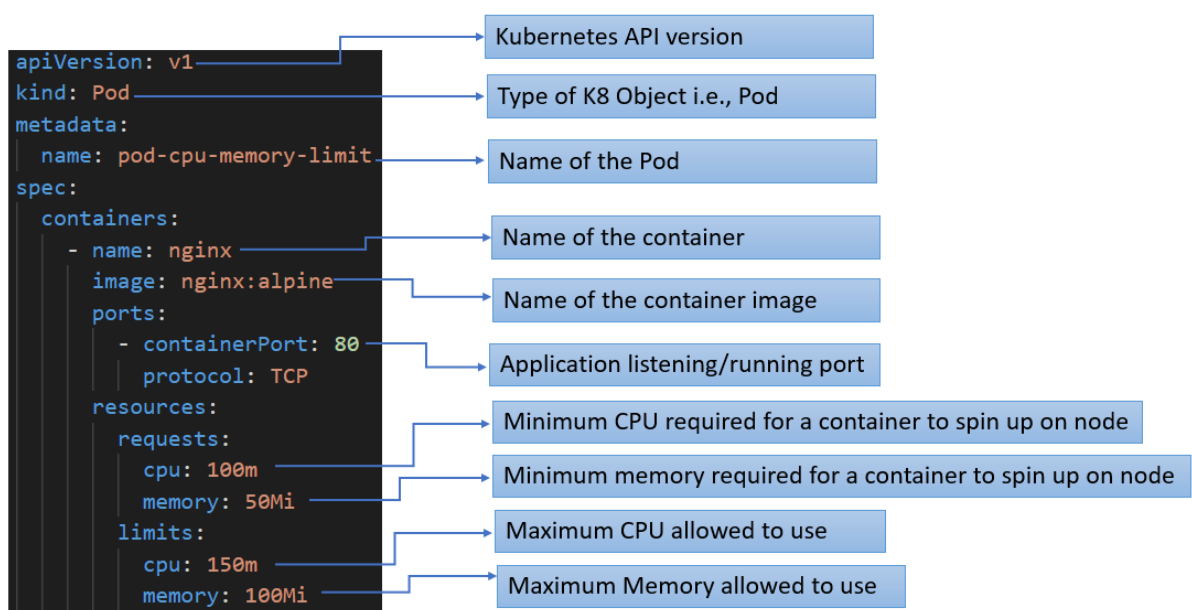
We can solve this problem by setting default requests and limits for container/pod per namespace using K8 LimitRange resource.

### LimitRange

Instead of setting requests and limits for each container explicitly in the pod, we can create Kubernetes LimitRange resource per namespace with default, min, and max request/limit. This LimitRange settings will be added to each container in the pod automatically when pod is created using same namespace.

Another benefit of LimitRange resource is, Pod will not be scheduled on node when developer set the requests and limits of container bigger than LimitRange min and max limits.

LimitRange helps developer to stop creating too tiny or too big container as it validates against LimitRange min and mix limit while creating the pod.



Pod manifest: pod-cpu-memory-limit.yaml

```

apiVersion: v1
kind: LimitRange
metadata:
  name: cpu-memory-limit-range
spec:
  limits:
    - type: Container
      defaultRequest:
        cpu: 100m
        memory: 100Mi
      default:
        cpu: 200m
        memory: 300Mi
      min:
        cpu: 30m
        memory: 30Mi
      max:
        cpu: 1000m
        memory: 600Mi

```

As per YAML,

- **type:** It specifies whether LimitRange settings are applicable to each container or entire Pod. Acceptable values are Container/Pod
- **defaultRequest:** These values will be added to a container automatically when container doesn't have its own CPU request and Memory request.
- **default:** These values will be added to a container automatically when container doesn't have its own CPU limit and Memory limit.
- **min:** It sets up the minimum Requests that a container in a Pod can set. The defaultRequest section cannot be lower than these values. Pod can't be created when its CPU and Memory requests are less than these values.
- **max:** It sets up the maximum limits that a container in a Pod can set. The default section cannot be higher than these values. Pod can't be created when its CPU and Memory limits are higher than these values.

save above yaml content as "cpu-memory-limit-range.yaml" and run the following kubectl commands

```

// create namespace
$ kubectl create ns limit-range-ns
namespace/limit-range-ns created

```



```
// create k8 limitrange resource under limit-range-ns namespace
$ kubectl apply -f cpu-memory-limit-range.yaml --namespace=limit-range-ns
limitrange/cpu-memory-limit-range created
```

```
//create pod with single container without specifying cpu/memory requests
and limits
$ kubectl run test-pod --image=nginx --restart=Never -n limit-range-ns
pod/test-pod created
```

```
// Check pod details where requests and limits are added automatically
based on K8 LimitRange
$ kubectl describe -n limit-range-ns pod/test-p
Name:          test-pod
Namespace:     limit-range-ns
Containers:
  test-pod:
    Container ID:  docker://e58640bb6eec
    Image:         nginx
    Limits:
      cpu:         200m
      memory:      300Mi
    Requests:
      cpu:         100m
      memory:      100Mi
```

try to create tiny container which values are less than LimitRange min settings.

```
apiVersion: v1
kind: Pod
metadata:
  name: cpu-memory-min-test
  namespace: limit-range-ns
spec:
  containers:
  - name: test-pod-2
    image: nginx
    resources:
      requests:
        cpu: 10m
        memory: 10Mi
```

save above yaml content as "cpu-memory-min-test.yaml" and run the following kubectl commands

```
$ kubectl apply -f cpu-memory-min-test.yaml
Error from server (Forbidden): error when creating "cpu-memory-min-test.yaml":
pods "cpu-memory-min-test" is forbidden: [minimum cpu usage per Container is
30m, but request is 10m, minimum memory usage per Container is 30Mi, but
request is 10Mi]
```