

# Kubernetes Fundamentals



Ganesh Palnitkar

PlusForum

[www.plusforum.in](http://www.plusforum.in)

# **Index**

- 1) Introduction
- 2) Kubernetes Architecture
- 3) Installing and configuring Kubernetes
- 4) Working with Kubernetes cluster
- 5) Deploying a simple application to cluster.

## External References:

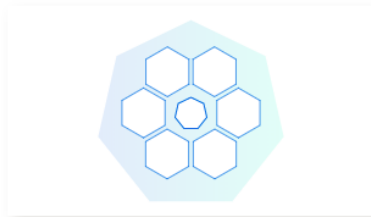
<https://kubernetes.io/docs/reference/kubectl/kubectl>

<https://kubernetes.io/docs/reference/kubectl/cheatsheet>

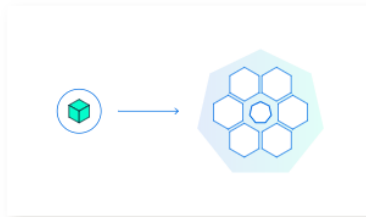
**\*\*\*\* Use Bash-completion in all tools while working on commandline\*\*\*\***

# What is Kubernetes?

## Kubernetes Basics Modules



1. [Create a Kubernetes cluster](#)



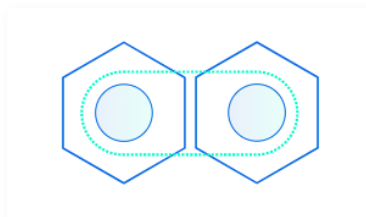
2. [Deploy an app](#)



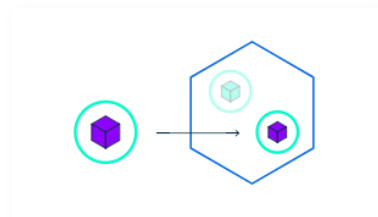
3. [Explore your app](#)



4. [Expose your app publicly](#)

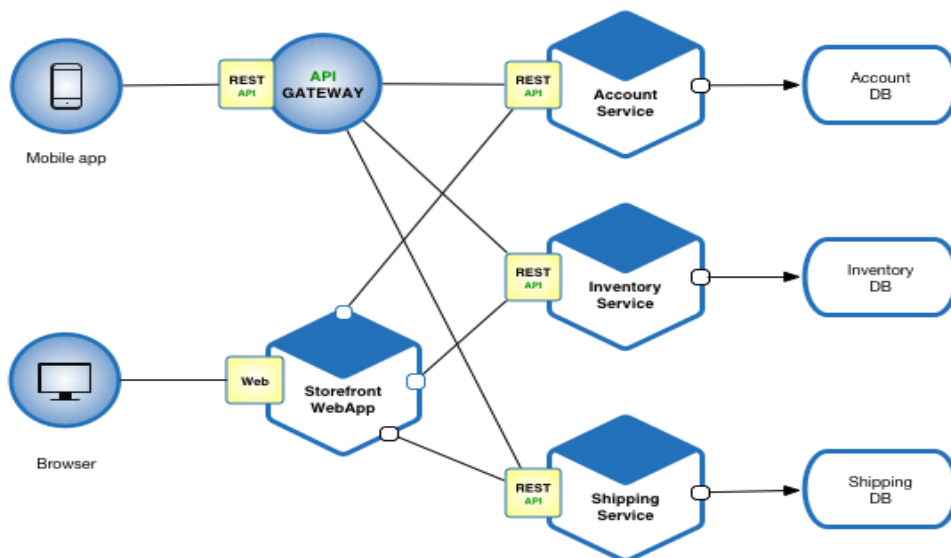


5. [Scale up your app](#)



6. [Update your app](#)

## What's a micro-service application?



## Kubernetes setup

Kubernetes can be installed using multiple ways.

- Minikube – for MacOS and Windows.... Works good for testing purpose that spins up a VM and creates all Kubernetes components in it.
- Search the page for [minikube](https://github.com/kubernetes/minikube/releases) latest release on page <https://github.com/kubernetes/minikube/releases>

- Google Container Engine.. this is integrated with google cloud and has Kubernetes available as a service. (SAAS)
  - On AWS, also available as a service (SAAS) as 'Kubernetes operations (kops)'. We can install all Kube componenets using 'kops' on AWS EC2 instances.
  - Manual install method, usually used while working with local datacentre or cloud. This uses the 'kubeadm'. With using commands like 'Kubeadm init' and 'kubeadm join -token <token>' we can create a kubernetes cluster and join nodes to the cluster.
- 

We will look at K8S using the last method of manual installs on cloud environment.

Spin up three instances on google cloud with one installed as Master and other two as nodes.

Runs only on Linux.

Each machine to be installed with below apps.

- 1) Docker or Rocket : Container runtime
- 2) Kubelet: kubernetes node agent (running on each node)
- 3) Kubeadm: Tool to build the cluster
- 4) Kubectl: Kubernetes client
- 5) CNI: Support for CNI networking (Container network interface)

## Kubernetes...



Enables developers to deploy their applications themselves and as often as they want



Enables the ops team by automatically monitoring and rescheduling those apps in the event of a hardware failure



The focus from supervising individual apps to mostly supervising and managing Kubernetes and the rest of the infrastructure



Abstracts away the hardware infrastructure and exposes your whole datacenter as a single gigantic computational resource

# Kubernetes Architecture:

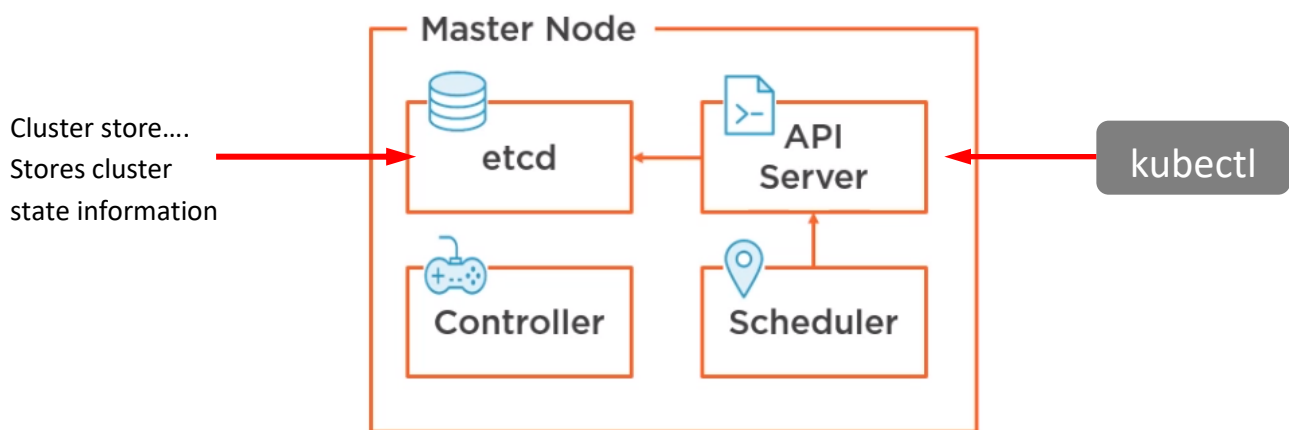
## Master and Node:

### K8S Master:

A single K8S Master or multi Master environment is definitely a possibility.

The Master has components like, API server, Proxy server, Container controller, cluster store etc.

**Nodes** are the VMs on which we will install three major components as the 'kubelet', 'Container runtime' and the 'kube-proxy'.



Functionalities of every control plane component.

API Server	etcd	Scheduler	Controller Manager
Central	Persists State	Watches API Server	Controller Loops
Simple	API Objects	Schedules Pods	Lifecycle functions and desired state
RESTful	Key-value	Resources	Watch and update the API Server
Updates etcd		Respects constraints	ReplicaSet

## Windows nodes in Kubernetes

To enable the orchestration of Windows containers in Kubernetes, include Windows nodes in your existing Linux cluster. Scheduling Windows containers in Pods on Kubernetes is similar to scheduling Linux-based containers.

In order to run Windows containers, your Kubernetes cluster must include multiple operating systems. While you can only run the control plane on Linux, you can deploy worker nodes running either Windows or Linux depending on your workload needs.

**Windows nodes are supported provided that the operating system is Windows Server 2019.**

### RESTful API Verbs:

GET	Get the data for a specified resource(s)
POST	Create a resource
DELETE	Delete a resource
PUT	Create or update entire existing resource
PATCH	Modify the specified fields of a resource

The kubectl commands helps to interact with the API server by using above mentioned methods, like get, post etc.

### Special API requests

LOG	Retrieve logs from a container in a Pod
EXEC	Exec a command in a container get the output
WATCH	Change notifications on a resource with streaming output

### API Resource Location (API Paths)

#### Core API (Legacy)

`http://apiserver:port/api/$VERSION/$RESOURCE_TYPE`

`http://apiserver:port/api/$VERSION/namespaces/$NAMESPACE/$RESOURCE_TYPE/$RESOURCE_NAME`

#### API Groups

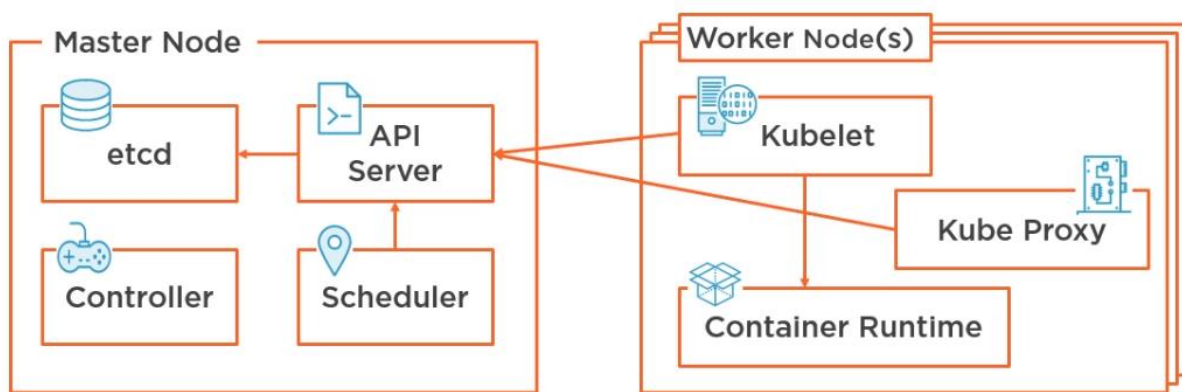
`http://apiserver:port/apis/$GROUPNAME/$VERSION/$RESOURCE_TYPE`

`http://apiserver:port/apis/$GROUPNAME/$VERSION/namespaces/$NAMESPACE/$RESOURCE_TYPE/$RESOURCE_NAME`

### Response code from API server

Success (2xx)	Client Errors (4xx)	Server Errors (5xx)
200 - OK	401 - Unauthorized	500 - Internal Server Error
201 - Created	403 - Access Denied	
202 - Accepted	404 - Not Found	

## Cluster structure and components:



Components running on all nodes (including Control plane) and their responsibilities.

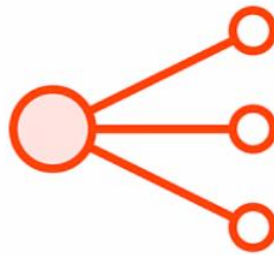
Kubelet	kube-proxy	Container Runtime
Monitors API Server for changes	iptables	Downloads images & runs containers
Responsible for Pod Lifecycle	Implements Services	Container Runtime Interface (CRI)
Reports Node & Pod state	Routing traffic to Pods	containerd
Pod probes	Load Balancing	Many others...



## Add-on PODs



DNS



Ingress



Dashboard

**Kubelet** is the main Kubernetes agent on the node, in fact we can say, the 'kubelet' is the node.

- It's the main kubernetes agent
- Registers node with cluster
- And watches the 'apiserver' on the master for work assignments.
- Instantiate PODs
- In case of any error, reports back to master.
- Exposes port 10255 to inspect the status of PODs
- Any stopped POD, kubelet on the node does not take any action to restart or recreate it, but simply reports back to master. Thus talks to the container runtime (Docker / rkt) for container management.

**Container Runtime** on the node is responsible for container management on the node,

- Pulling images
- Starting stopping container.
- Thus pluggable and uses 'Docker' or 'CoreOS rkt'.

**Kube-proxy** on the node is responsible for networking within PODs on the node.

- POD IP addresses
  - o All container inside a POD share a single IP
- Load balancing across all PODs in a service inside a node.

## Working with Kubernetes cloud service:



#### Elastic Kubernetes Service (EKS)

<https://aws.amazon.com/getting-started/projects/deploy-kubernetes-app-amazon-eks/>



#### Google Kubernetes Engine (GKE)

<https://cloud.google.com/kubernetes-engine/docs/how-to/>



#### Azure Kubernetes Services (AKS)

<https://docs.microsoft.com/en-us/azure/aks/kubernetes-walkthrough>

## Declarative Model and Desired State:

In Kubernetes we do not specify how to create a cluster and containers inside a POD but we just define what we want to achieve. So that becomes our 'Desired State Configuration' of the Container and related services in it. This is stated using a YAML or JSON as the kubernetes manifest files.

So if we want to have 10 PODS running always with a particular version of a file deployed to it, this is defined in the manifest and K8S will always make sure to achieve this state.

## Working with Pods

PODs theory:

1. POD is the smallest atomic unit of scheduling in Kubernetes.
2. One POD can contain multiple containers in it with same or multiple services running in it.
3. But one POD can only be running on one node and can never be split on multiple nodes.
4. PODs are declarative via Manifest files.

A simple comparison:

POD can be termed as a simple ringed fenced environment with Network stack, Kernel namespaces, volume mounts and containers in it.

If two containers need to share same volume inside a POD, those can reside inside a single POD as well.

Hypervisor / Virtualization

**VMs**

(Atomic Unit of Scheduling)

Docker

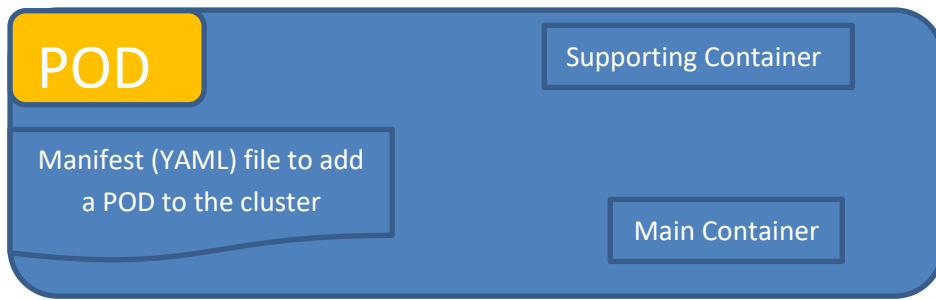
**Containers**

(Atomic Unit of Scheduling)

Kubernetes

**PODs**

(Atomic Unit of Scheduling)

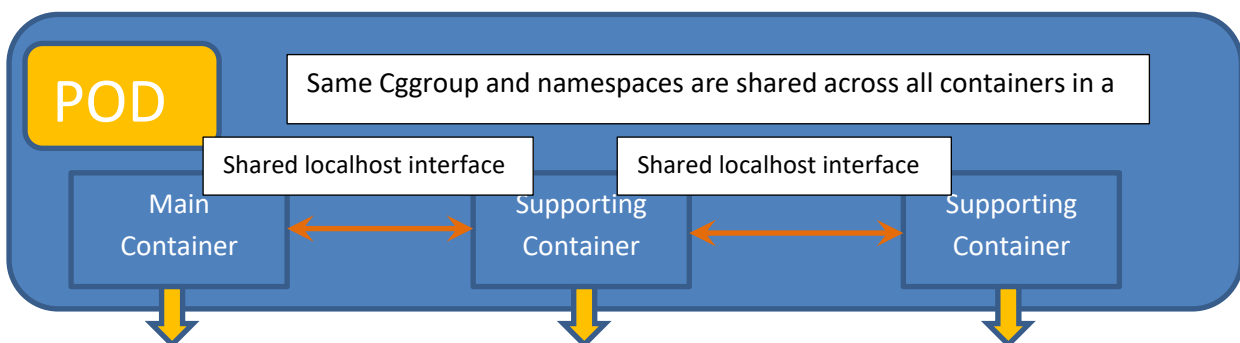


```
[root@my-node1 ~] $ mkdir /etc/kubelet.d/
[root@my-node1 ~] $ cat <<EOF >/etc/kubelet.d/static-web.yaml
apiVersion: v1
kind: Pod
metadata:
  name: static-web
  labels:
    role: myrole
spec:
  containers:
  - name: web
    image: nginx
    ports:
    - name: web
      containerPort: 80
      protocol: TCP
EOF
```

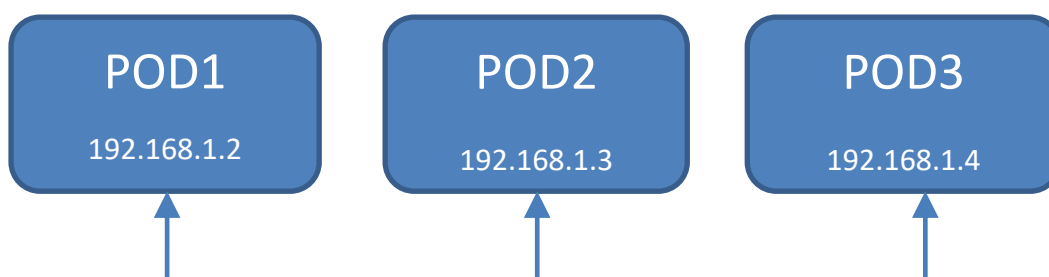
This file is then fed to the API server and the scheduler deploys this to a Kube node.

In a POD, even if there are multiple containers, each POD has only one IPAddress. So the containers inside each POD is accessed over different port like, as shown below,

## POD Networking: Intra-POD communication



## Inter POD communication: POD network

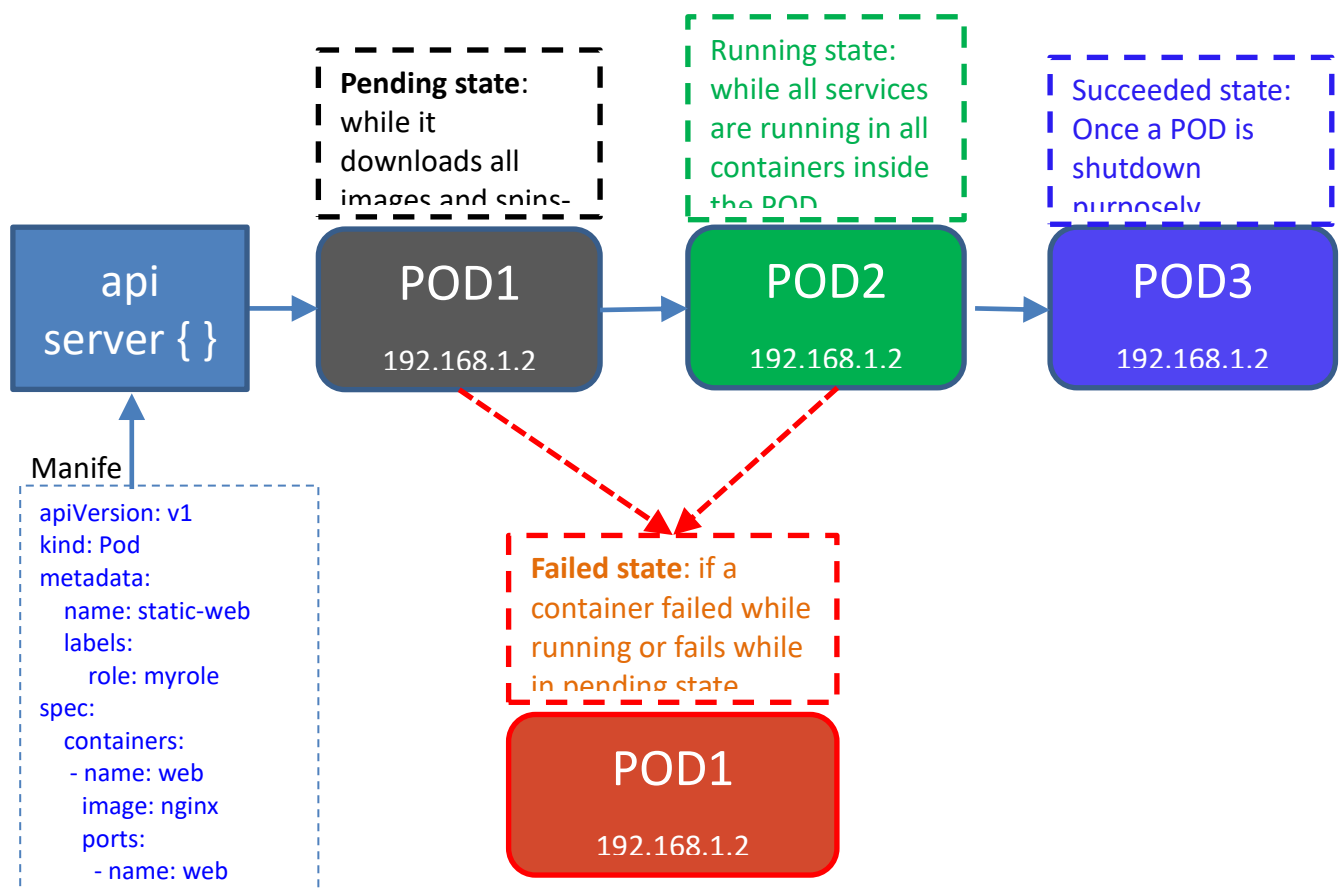


Like Container in Docker, PODs are mortal units in Kubernetes. We never get into the mode of repairing a POD.

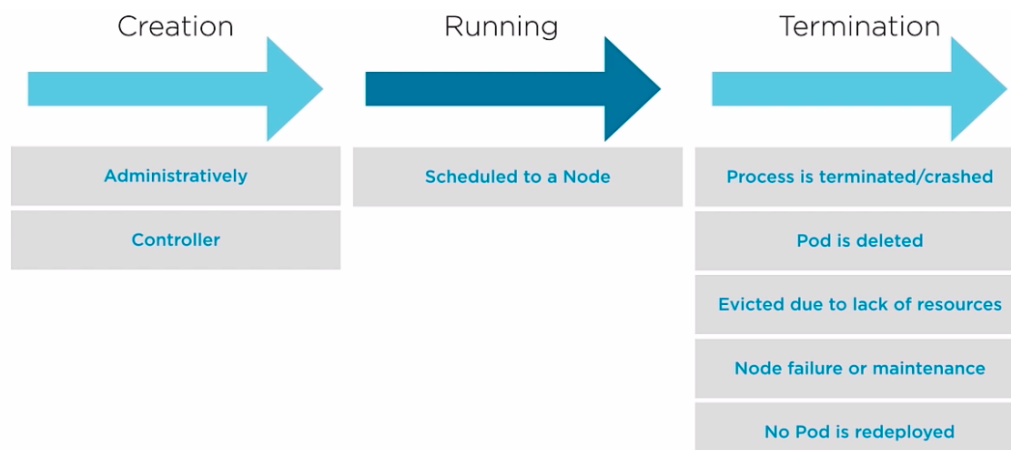
## Kubernetes Networking Fundamentals:



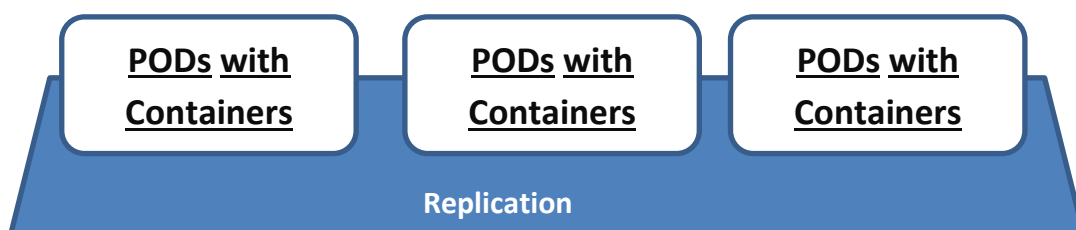
## PODs Lifecycle phases:



## How does a POD progress in its lifecycle?



## Replication Controller:

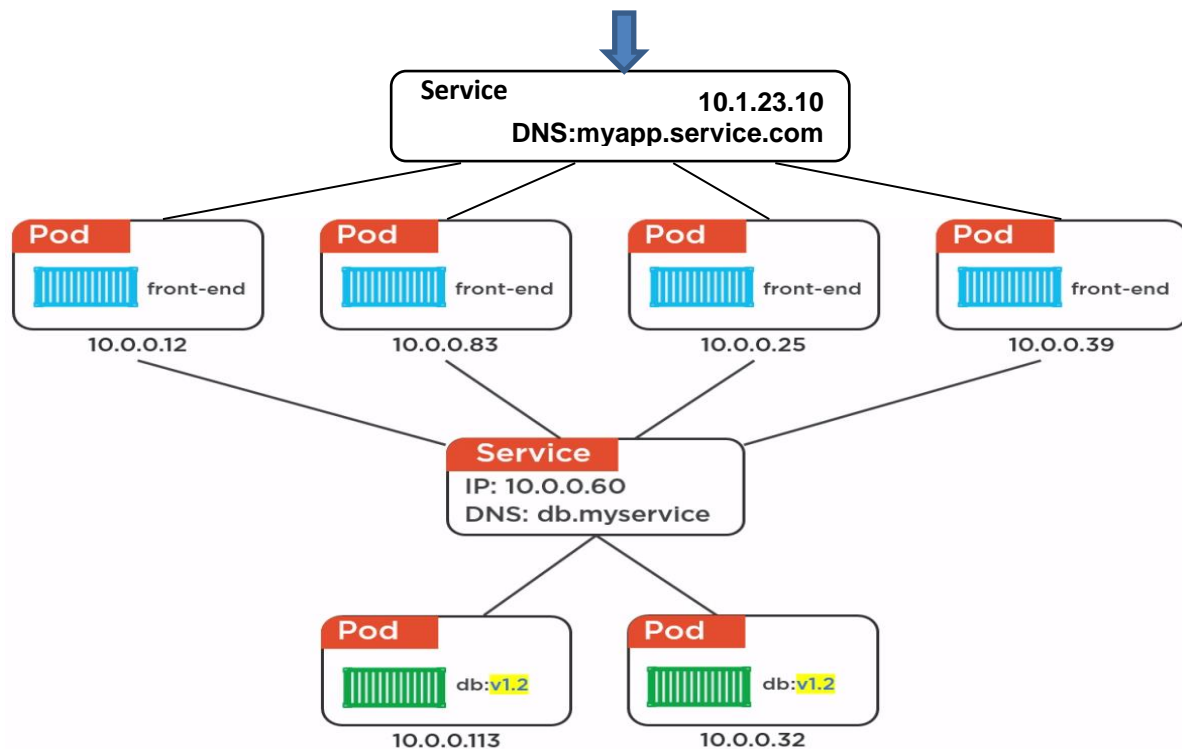


## Kubernetes Services:

### Theory:

'Service' is a REST object in the Kubernetes API. Services are defined in YAML files.

With IP addresses flushed in case a new node is brought or a POD is recreated a new IP address is allocated, in such case we use the service in between the front-end pods of an application to talk to the back-end pods in the application. Here the Service component is important to manage the request routing to the backend Pods. In a way it is the load balancing for the backend Pods.



The way a Pod belongs to a service is done via assigning labels.

Service types:

- 1) ClusterIP: Stable Internet cluster IP
  - 2) NodePort: exposes the app outside of the cluster by adding a cluster-wide port on top of the ClusterIP.
  - 3) LoadBalancer: Integrates NodePort with load based load balancers.
- - Services provide reliable network endpoint
    - o IP address
    - o DNS Name
    - o Port
  - Exposes Pods to the outside world.
    - o NodePort provide cluster-wide port

## Kubernetes Deployments:

In Kubernetes, how deployments are managed?... it is done by defining a manifest written in JSON format and sent to the 'apiserver'.

Deployments are REST objects defined in JSON / YAML format and help in below challenges..

- 1) Deployment Manifests are self-documenting
- 2) Specify once and deploy multiple times.
- 3) Easy versioning
- 4) Simple rolling updates and rollbacks

```

apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: hello-deploy
spec:
  replicas: 5
  minReadySeconds: 2
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  template:
    metadata:
      labels:
        app: hello-world-updated
    spec:
      containers:
        - name: hello-pod
          image: ganeshhp/maven-petclinic-project:latest
          ports:
            - containerPort: 8080

```

JSON / YAML files are deployed using API server on master. Uses replica sets to define number of PODS to be deployed.

## Installing and getting started with Master and Nodes on Ubuntu:

### Where to Install?

- Cloud
  - o IAAS – Virtual Machines
  - o PAAS – Kubernetes Managed service on cloud (AKS, EKS)
- On-Premises
  - o Bare Metal
  - o Virtual Machines

### Which one do you choose?

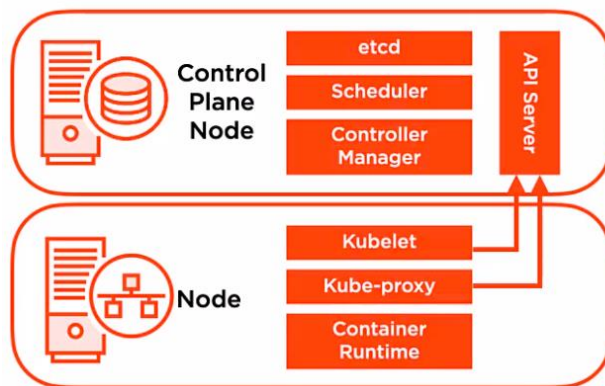
We will first install and configure cluster on local virtual machines, in a way choosing Infra as a Service (IAAS)

### Installation requirements:

System Requirements	Container Runtime	Networking
Linux - Ubuntu/RHEL	Container Runtime Interface (CRI)	Connectivity between all Nodes
2 CPUs	containerd	Unique hostname
2GB RAM	Docker (Deprecated 1.20)	Unique MAC address
Swap Disabled	CRI-O	

## Setting up Network Port

### Cluster Network Ports



Component	Ports (tcp)	Used By
API	6443	All
etcd	2379-2380	API/etcd
Scheduler	10251	Self
Controller Manager	10252	Self
Kubelet	10250	Control Plane
Kubelet	10250	Control Plane
NodePort	30000-32767	All

## Installing Kubernetes standalone cluster

### Required Packages:

- Container runtime- containerd / Docker
- kubelet – client service running on all nodes
- kubeadm – cluster administrative utility.
- Kubectl – command line utility to communicate with cluster.

Sequence of commands:

```
sudo apt-get install -y containerd

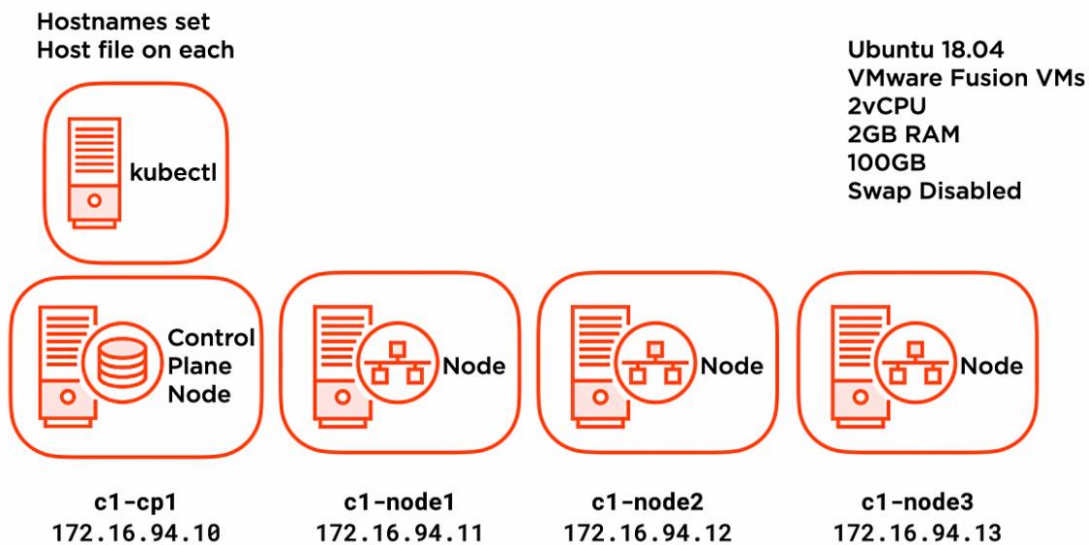
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -

cat <<EOF >/etc/apt/sources.list.d/kubernetes.list
deb https://apt.kubernetes.io/ kubernetes-xenial main
EOF

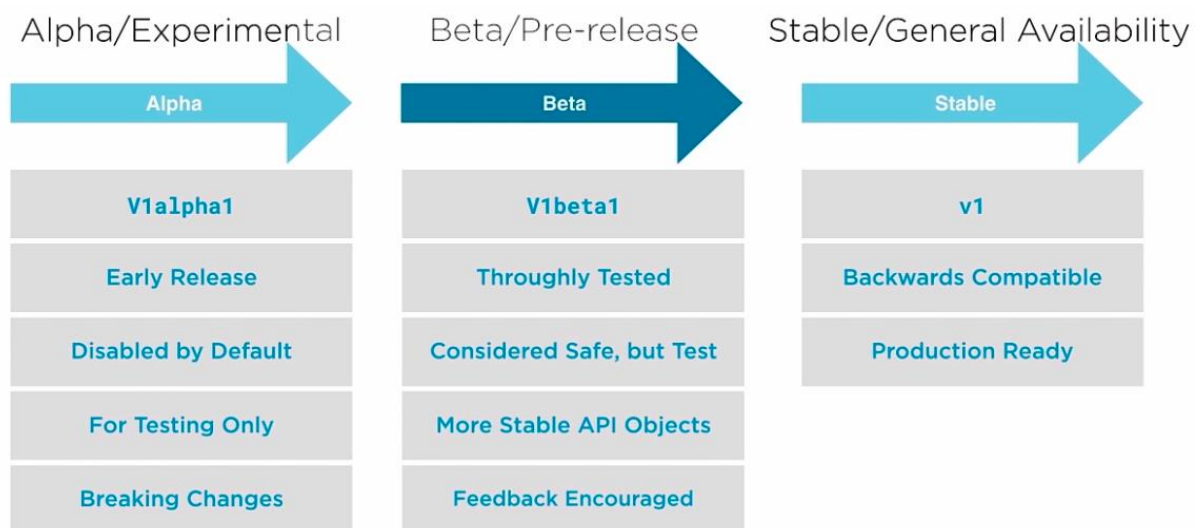
apt-get update
apt-get install -y kubelet kubeadm kubectl
apt-mark hold kubelet kubeadm kubectl containerd
```



## Sample cluster configuration:



## API Versioning:



## Starting cluster installation:

```
$ swapoff -a
```

To implement swapoff, open file at [/etc/fstab](#) and comment out 2nd line in the file that reads about swap memory.

On the **Ubuntu VM** that's going to be the Kubernetes Master, follow below commands, to start.

```
$ apt-get update && apt-get install -y apt-transport-https  
$ curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg |  
apt-key add -  
$ cat <<EOF >/etc/apt/sources.list.d/kubernetes.list
```

```

deb http://apt.kubernetes.io/ kubernetes-xenial main
EOF
$ apt-get update
$ apt-get install docker.io kubeadm kubect1 kubelet kubernetes-
cni

```

### For Centos / RHEL:

Install docker-ce using below steps first,

```

$ sudo yum install -y yum-utils device-mapper-persistent-data lvm2
$ sudo yum-config-manager --add-repo
https://download.docker.com/linux/centos/docker-ce.repo
$ sudo yum install docker-ce
$ sudo systemctl start docker
$ sudo systemctl enable docker
$ cat <<EOF > /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-
x86_64
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg
https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
EOF
$ yum makecache fast
$ yum update
$ yum install -y kubelet kubeadm kubect1 -y
$ source <(kubect1 completion bash)
$ kubect1 completion bash > /etc/bash_completion.d/kubect1

```

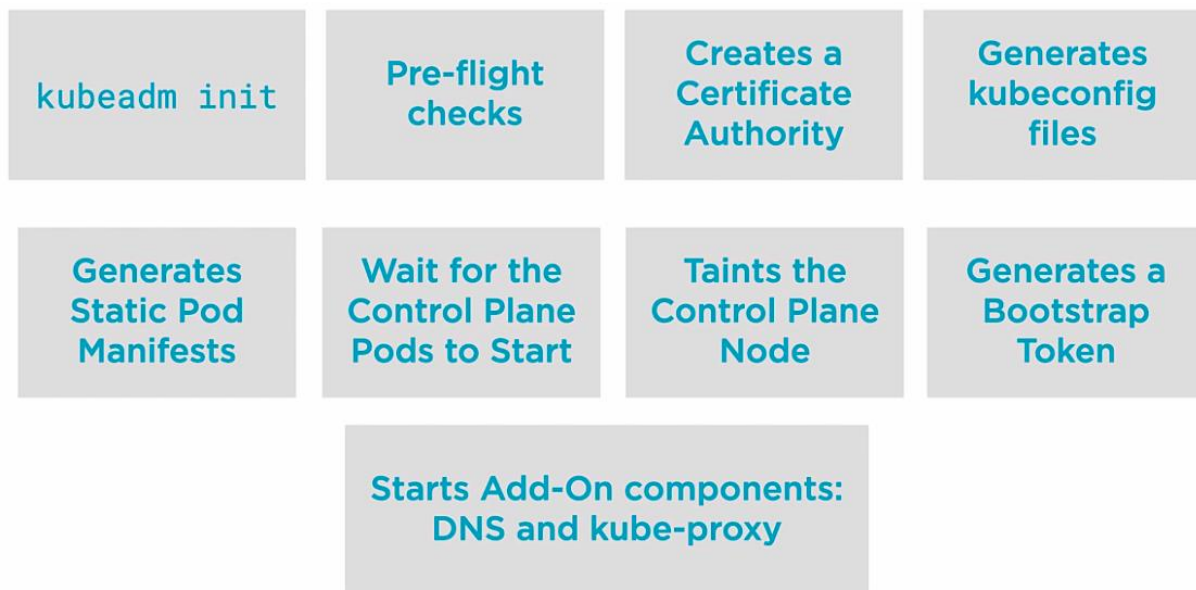
This will install apps as per below latest version available at the time of installation.

- docker.io (1.13.1-0ubuntu1~16.04.2).
- kubeadm (1.9.0-00).
- kubect1 (1.9.0-00).
- kubelet (1.9.0-00).
- kubernetes-cni (0.6.0-00).

Follow above steps on all VMs irrespective of their status as 'Nodes' or 'Master'.

## Bootstrapping Cluster with kubeadm

Below are the steps in the process of cluster bootstrapping.



```
$ kubeadm config images pull
```

```
[root@kube-master plusforum_in]# kubeadm config images pull
[config/images] Pulled k8s.gcr.io/kube-apiserver:v1.20.4
[config/images] Pulled k8s.gcr.io/kube-controller-manager:v1.20.4
[config/images] Pulled k8s.gcr.io/kube-scheduler:v1.20.4
[config/images] Pulled k8s.gcr.io/kube-proxy:v1.20.4
[config/images] Pulled k8s.gcr.io/pause:3.2
[config/images] Pulled k8s.gcr.io/etcd:3.4.13-0
[config/images] Pulled k8s.gcr.io/coredns:1.7.0
```

First get the kubeadm Config file. If we are required to modify this file we can do it here.

```
$ kubeadm config print init-defaults | tee ClusterConfiguration.yaml
```

To create a kube cluster Run,

```
$ kubeadm init
```

Or,

```
$ kubeadm init --apiserver-advertise-address 192.168.1.8 --ignore-
preflight-errors=all
```

IP address in above command is the ip address of Kubemaster server that you want to advertise.

This will download and initiate kuberadm container and other apps,

```
*****
```

While running the Kubeadm join command on the nodes, in case of nodes created using vagrant, the IP address published are not for corrected NIC. To resolve this problem set the correct IP Address for the Master, by using the command,

```
kubeadm init --apiserver-advertise-address=192.168.33.135
```

\*\*\*\*\*

Below message is displayed on kubemaster

```
"You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options
listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/
"
```

.... You will be prompted to follow below steps.

```
$ mkdir -p $HOME/.kube
```

```
$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
```

```
$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

But you won't be able to see the nodes ready and to make those ready we have to initiate all other nodes in the cluster.

**We need to install networking specific add-ons and available options are listed at,**

<https://kubernetes.io/docs/concepts/cluster-administration/addons/>

We use the command to start network router service (POD) using below command,

```
$ kubectl apply --filename https://git.io/weave-kube-1.6
```

Or,

```
$ kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version
| base64 | tr -d '\n')"
```

**Now on the VMs that are going to be the K8S nodes**, run below command

```
$ kubeadm join --token fc3846.48223acdabb2ae31 <master-ipaddress>:6443 --
discovery-token-ca-cert-hash
sha256:a905fc15814645e8a1cb3a18234f6ea5206263db820889d0b773cc1cd7751a0e
```

If the Kubeadm join command is not available, we can create a new token with below command,

```
$ sudo kubeadm token create --print-join-command
```

Or, below can retrieve existing token

```
$ kubeadm join --discovery-token-unsafe-skip-ca-verification --  
token=`kubeadm token list` 172.17.0.92:6443.
```

Also we can list all available and valid tokens using command,

```
$ kubeadm token list
```

# Using 'kubectl' command to interact with cluster components:

Below are some frequently used options while working with **kubectl**.

```
$ kubectl apply / create ... to create a resource
$ kubectl run ... to start a POD from an image
$ kubectl explain ... documentation of Kubernetes cluster resources
$ kubectl delete ... delete resource
$ kubectl get ... list resources
$ kubectl describe ... detailed resource information, valuable while troubleshooting
$ kubectl exec ... like docker exec command, to execute a command inside a container
$ kubectl logs ... view logs on a container
```

Along with these options we can also use some additional switches to get the output formatted.

**wide** – output additional information

**YAML** – YAML formatted API object

**JSON** – JSON formatted API object

**dry-run=<option>** – Option must be "none", "server", or "client". If client strategy, only print the object that would be sent, without sending it. If server strategy, submit server-side request without persisting the resource.

Example:

```
$ kubectl apply -f deployment.yaml -dry-run=server
$ kubectl apply -f deployment.yaml -dry-run=client ...validating
syntax in deployment.yaml file.
$ kubectl create deployment nginx -image=nginx -dry-run=client
-o yaml > deployment.yaml .... Create a manifest from an imperative
kubectl command
```

kubectl	[command]	[type]	[name]	[flags]
kubectl	get	pods	pod1	--output=yaml
kubectl	create	deployment	nginx	--image=nginx

In order to get the difference between a running Kubernetes object in kube cluster against the same being updated in a manifest file.

In below example, deployment object specified in deployment.yaml will be validated against the same deployment object deployed and running in Kubernetes cluster.

```
$ kubectl diff -f deployment.yaml
```

First, let's try to get information about the cluster

```
$ kubectl cluster-info
```

Kubernetes control plane is running at https://192.168.33.10:6443

CoreDNS is running at https://192.168.33.10:6443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
kube-master	Ready	master	21m	v1.9.0
kubenode-1	Ready	<none>	2m	v1.9.0
kubenode-2	Ready	<none>	39s	v1.9.0

```
$ kubectl get nodes -o wide
```

NAME	STATUS	ROLES	AGE	VERSION	INTERNAL-IP	EXTERNAL-IP	OS-IMAGE	KERNEL-VERSION	CONTAINER-RUNTIME
controlplane	Ready	control-plane,master	19h	v1.21.1	192.168.33.10	<none>	CentOS Linux 8	4.18.0-305.3.1.el8.x86_64	docker://20.10.7
kubenode1	NotReady	<none>	19h	v1.21.1	192.168.33.11	<none>	CentOS Linux 8	4.18.0-305.3.1.el8.x86_64	docker://20.10.7

Here we can see the additional nodes which have hostname as 'kubenode-1' and 'kubenode-2'.

Now that we are done with our Master and Node setup let's start with getting the PODs ready,

There are system PODs deployed into the cluster that we can see by running command,

TO get information on how many clusters that we can manage using our `kubectl` utility, we can use the command,

```
$ kubectl config get-context
```

Getting information on POD objects.

```
$ kubectl get pods --all-namespaces
```

Or,

```
$ kubectl get pods --namespace kube-system
```

We can check the status of PODs by adding `--watch` switch,

```
$ kubectl get pods --all-namespaces --watch
```

To get some details about how client and API Server interacts, we can use the verbosity option as status below,

```
$ kubectl get pods --watch -v 6
```

To kill the `watch` or `proxy` process and come out, we use the key combination `fg` and `ctrl+c`.

To get a detailed output of all resources in the cluster run below command,

```
$ kubectl get all --all-namespaces
```

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
kube-system	pod/coredns-558bd4d5db-6dglq	1/1	Running	1	20h
kube-system	pod/coredns-558bd4d5db-cwd5q	1/1	Running	1	20h
kube-system	pod/etcd-controlplane	1/1	Running	1	20h
kube-system	pod/kube-apiserver-controlplane	1/1	Running	1	20h
kube-system	pod/kube-controller-manager-controlplane	1/1	Running	1	20h
kube-system	pod/kube-proxy-k825c	1/1	Running	1	20h
kube-system	pod/kube-proxy-wmtrf	1/1	Running	0	20h
kube-system	pod/kube-scheduler-controlplane	1/1	Running	1	20h
kube-system	pod/weave-net-qchxl	2/2	Running	2	20h
kube-system	pod/weave-net-w95qs	1/2	CrashLoopBackOff	8	20h

NAMESPACE	NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
default	service/kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	20h
kube-system	service/kube-dns	ClusterIP	10.96.0.10	<none>	53/UDP,53/TCP,9153/TCP	20h

NAMESPACE	NAME	DESIRED	CURRENT	READY	UP-TO-DATE	AVAILABLE	NODE
SELECTOR	AGE						
kube-system	daemonset.apps/kube-proxy	2	2	1	2	1	
kubernetes.io/os=linux	20h						
kube-system	daemonset.apps/weave-net	2	2	1	2	1	<none>
	20h						

NAMESPACE	NAME	READY	UP-TO-DATE	AVAILABLE	AGE
kube-system	deployment.apps/coredns	2/2	2	2	20h

NAMESPACE	NAME	DESIRED	CURRENT	READY	AGE
kube-system	replicaset.apps/coredns-558bd4d5db	2	2	2	20h

To get a detailed information on a resource, we can use the `explain` option on a certain resource as below,

```
$ kubectl explain pods | more
```

```
$ kubectl explain pod.spec | more
```

```
$ kubectl explain pod.spec.containers | more
```



We can also get list of all cluster resources by running command,

`$ kubectl api-resources` (Highlighted in RED are frequently used resources)

NAME	SHORTNAMES	APIVERSION	NAMESPACED	KIND
mutatingwebhookconfigurations		admissionregistration.k8s.io/v1	FALSE	MutatingWebhookConfiguration
validatingwebhookconfigurations		admissionregistration.k8s.io/v1	FALSE	ValidatingWebhookConfiguration
customresourcedefinitions	crd, crds	apiextensions.k8s.io/v1	FALSE	CustomResourceDefinition
apiservices		apiregistration.k8s.io/v1	FALSE	APIService
controllerrevisions		apps/v1	TRUE	ControllerRevision
<b>daemonsets</b>	<b>ds</b>	<b>apps/v1</b>	<b>TRUE</b>	<b>DaemonSet</b>
<b>deployments</b>	<b>deploy</b>	<b>apps/v1</b>	<b>TRUE</b>	<b>Deployment</b>
<b>replicasets</b>	<b>rs</b>	<b>apps/v1</b>	<b>TRUE</b>	<b>ReplicaSet</b>
<b>statefulsets</b>	<b>sts</b>	<b>apps/v1</b>	<b>TRUE</b>	<b>StatefulSet</b>
tokenreviews		authentication.k8s.io/v1	FALSE	TokenReview
localsubjectaccessreviews		authorization.k8s.io/v1	TRUE	LocalSubjectAccessReview
selfsubjectaccessreviews		authorization.k8s.io/v1	FALSE	SelfSubjectAccessReview
selfsubjectrulesreviews		authorization.k8s.io/v1	FALSE	SelfSubjectRulesReview
subjectaccessreviews		authorization.k8s.io/v1	FALSE	SubjectAccessReview
horizontalpodautoscalers	hpa	autoscaling/v1	TRUE	HorizontalPodAutoscaler
cronjobs	cj	batch/v1	TRUE	CronJob
jobs		batch/v1	TRUE	Job
certificatesigningrequests	csr	certificates.k8s.io/v1	FALSE	CertificateSigningRequest
leases		coordination.k8s.io/v1	TRUE	Lease
endpointslices		discovery.k8s.io/v1	TRUE	EndpointSlice
events	ev	events.k8s.io/v1	TRUE	Event
<b>ingresses</b>	<b>ing</b>	<b>extensions/v1beta1</b>	<b>TRUE</b>	<b>Ingress</b>
flowschemas		flowcontrol.apiserver.k8s.io/v1beta1	FALSE	FlowSchema
prioritylevelconfigurations		flowcontrol.apiserver.k8s.io/v1beta1	FALSE	PriorityLevelConfiguration
ingressclasses		networking.k8s.io/v1	FALSE	IngressClass
ingresses	ing	networking.k8s.io/v1	TRUE	Ingress
networkpolicies	netpol	networking.k8s.io/v1	TRUE	NetworkPolicy
runtimeclasses		node.k8s.io/v1	FALSE	RuntimeClass
podd disruptionbudgets	pdb	policy/v1	TRUE	PodDisruptionBudget
podsecuritypolicies	psp	policy/v1beta1	FALSE	PodSecurityPolicy
clusterrolebindings		rbac.authorization.k8s.io/v1	FALSE	ClusterRoleBinding
clusterroles		rbac.authorization.k8s.io/v1	FALSE	ClusterRole
rolebindings		rbac.authorization.k8s.io/v1	TRUE	RoleBinding

<b>roles</b>		<b>rbac.authorization.k8s.io/v1</b>	<b>TRUE</b>	<b>Role</b>
priorityclasses	pc	scheduling.k8s.io/v1	FALSE	PriorityClass
csidrivers		storage.k8s.io/v1	FALSE	CSIDriver
csinodes		storage.k8s.io/v1	FALSE	CSINode
<b>storageclasses</b>	<b>sc</b>	<b>storage.k8s.io/v1</b>	<b>FALSE</b>	<b>StorageClass</b>
volumeattachments		storage.k8s.io/v1	FALSE	VolumeAttachment
csistoragecapacities		storage.k8s.io/v1beta1	TRUE	CSIStorageCapacity
bindings		v1	TRUE	Binding
componentstatuses	cs	v1	FALSE	ComponentStatus
configmaps	cm	v1	TRUE	ConfigMap
endpoints	ep	v1	TRUE	Endpoints
events	ev	v1	TRUE	Event
limitranges	limits	v1	TRUE	LimitRange
<b>namespaces</b>	<b>ns</b>	<b>v1</b>	<b>FALSE</b>	<b>Namespace</b>
<b>nodes</b>	<b>no</b>	<b>v1</b>	<b>FALSE</b>	<b>Node</b>
<b>persistentvolumeclaims</b>	<b>pvc</b>	<b>v1</b>	<b>TRUE</b>	<b>PersistentVolumeClaim</b>
<b>persistentvolumes</b>	<b>pv</b>	<b>v1</b>	<b>FALSE</b>	<b>PersistentVolume</b>
<b>Pods</b>	<b>po</b>	<b>v1</b>	<b>TRUE</b>	<b>Pod</b>
podtemplates		v1	TRUE	PodTemplate
<b>replicationcontrollers</b>	<b>rc</b>	<b>v1</b>	<b>TRUE</b>	<b>ReplicationController</b>
resourcequotas	quota	v1	TRUE	ResourceQuota
<b>secrets</b>		<b>v1</b>	<b>TRUE</b>	<b>Secret</b>
serviceaccounts	sa	v1	TRUE	ServiceAccount
<b>services</b>	<b>svc</b>	<b>v1</b>	<b>TRUE</b>	<b>Service</b>

TO get resources from a specific API group,

```
$ kubectl api-groups --api-group=<groupName> ...
```

GroupName can be like **apps** or **batch** ... as can be seen in the above table of resources.

To get detailed documentation about a resource we can use explain option with a resource.

```
$ kubectl explain pod | more
```

Or,

```
$ kubectl explain pod -recursive
```

Similar to explain more detailed information about an existing object / resource in cluster can be retrieved with option **describe**.

```
$ kubectl describe node <nodename>
```

There are two ways to work with deploying objects in Kubernetes cluster,

- **Imperative**
  - We can deploy manage one resource / object at a time by defining requirements on command line, example,
  - `$ kubectl create deployment webapp --image=httpd:latest`
- **Declarative**
  - To manage multiple cluster resources / objects by defining multiple options as a code (DSC), we write a manifest file in YAML/JSON format.

## Using Imperative way:

Generating a manifest file using the dry-run option.

```
$ kubectl create deployment webapp \
--image=httpd:latest \
--dry-run=client -o yaml > deployment.yaml
```

As we run the above command for deployment object, we can use a similar one for creating service object and related manifest for the deployed deployment object.

```
$ kubectl expose deployment webapp \
--port=80 --target-port=8080 \
--dry-run=client -o yaml > service.yaml
```

With this command a YAML file (Manifest) for the load-balancer service will get created and the same can be deployed using `kubectl create -f service.yaml` command.

To deploy service imperatively (without Manifest) remove `--dry-run` option from command and run `kubectl expose` command to deploy service.

Once a deployment object is deployed imperatively, we can get a manifest created for the deployed object by using below command.

```
$ kubectl get deployment webapp -o deployment.yaml
```

Once objects are deployed in Kubernetes cluster, we can scale up or down the number of PODs by using scale option as shown below.

```
$ kubectl scale deployment webapp --replicas=15
```

## Starting with PODs practical (Declarative provisioning):

How to we define a API object and what goes into it, refer to the documentation..

<https://kubernetes.io/ocs/reference/kubernetes-api>

The POD is defined using a manifest file.

Here is sample manifest file.. pod.yml

```
apiVersion: v1
kind: Pod
metadata:
  name: static-web
  labels:
    role: myrole
    zone: prod
    version: v1
spec:
  containers:
  - name: web
    image: nginx
    ports:
    - name: web
      containerPort: 80
      protocol: TCP
```

Now the manifest is fed to the api-server using below command,

```
root@kubernetes-master:~# kubectl create -f pod.yml
pod "static-web" created
```

Check the status of POD using command,

```
root@kubernetes-master:~# kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
static-web	1/1	Running	0	14s

To check the POD status let's run the describe command.

```
root@kubernetes-master:~# kubectl describe pods
```

Name:	static-web
Namespace:	default
Node:	kubenode-2/10.142.0.4
Start Time:	Tue, 02 Jan 2018 08:18:51 +0000
Labels:	role=myrole version=v1 zone=prod
Annotations:	<none>
Status:	Running
IP:	10.36.0.1
Containers:	
web:	

```

    Container ID:
docker://8cabde7b6892ba3db454f0d73622d05659698f0ed484364c5d990b22e970fe7
3
    Image:      nginx
    Image ID:   docker-
pullable://nginx@sha256:cf8d5726fc897486a4f628d3b93483e3f391a76ea4897de0
500ef1f9abcd69a1
    Port:      80/TCP
    State:     Running
    Started:   Tue, 02 Jan 2018 08:18:59 +0000
    Ready:     True
    Restart Count: 0
    Environment: <none>
    Mounts:
    /var/run/secrets/kubernetes.io/serviceaccount from default-token-
qtvzc (ro)
Conditions:
  Type           Status
  Initialized    True
  Ready          True
  PodScheduled   True
Volumes:
  default-token-qtvzc:
    Type:      Secret (a volume populated by a Secret)
    SecretName: default-token-qtvzc
    Optional:  false
QoS Class:     BestEffort
Node-Selectors: <none>
Tolerations:   node.kubernetes.io/not-ready:NoExecute for 300s
                node.kubernetes.io/unreachable:NoExecute for 300s
Events:
  Type           Reason              Age             From              Message
  ----           -
  Normal        Scheduled            1m             default-scheduler Successfully
assigned static-web to kubenode-2
  Normal        SuccessfulMountVolume 1m             kubelet, kubenode-2
MountVolume.SetUp succeeded for volume "default-token-qtvzc"
  Normal        Pulling              1m             kubelet, kubenode-2 pulling
image "nginx"
  Normal        Pulled                1m             kubelet, kubenode-2 Successfully
pulled image "nginx"
  Normal        Created              1m             kubelet, kubenode-2 Created
container
  Normal        Started              1m             kubelet, kubenode-2 Started
container

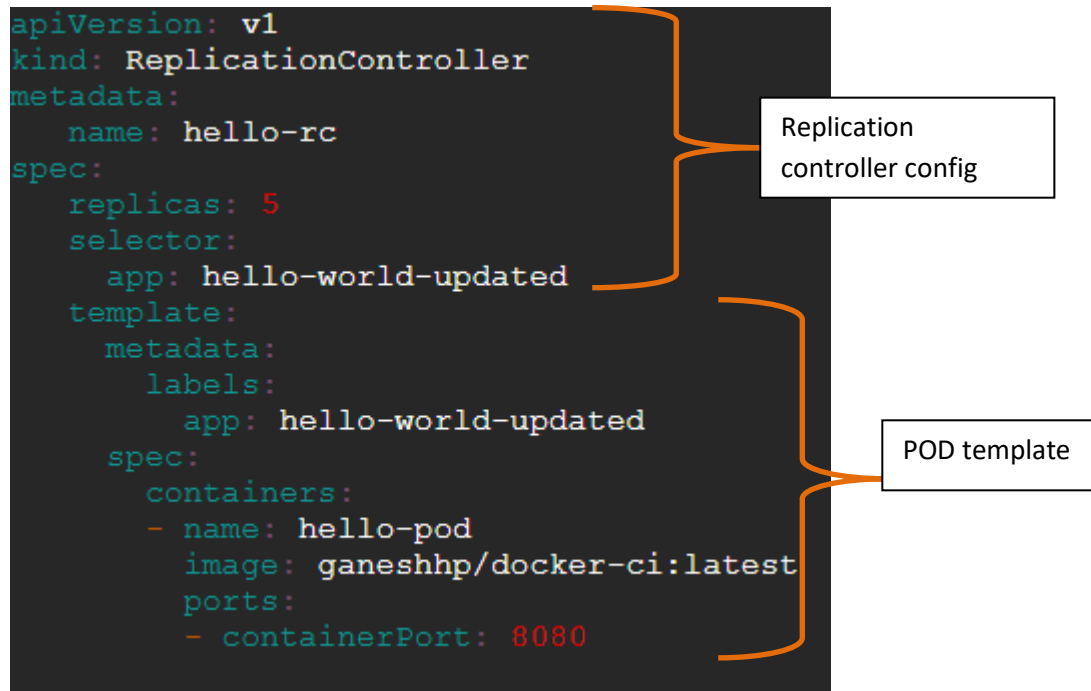
```

To delete a POD, we run the command,

```
$ kubectl delete pods <pod-name>
```

```
root@k8s-master:~# kubectl delete pods static-web
pod "static-web" deleted
```

Now let's look at **Replication Controller**, which is the right ways of applying Desired State Configuration.



So here the replication controller manifest is the actual 'Desired State Configuration' (DSC), where we state how many PODs we want to create and also supply the POD configuration that's what we have seen in earlier POD manifest.

Now try running the file one more time again using the same command,

```
$ kubectl create -f rc.yml
```

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
static-web-replica-bh8rk	1/1	Running	0	23s
static-web-replica-m9ck4	1/1	Running	0	23s
static-web-replica-pkd8h	1/1	Running	0	23s
static-web-replica-qwflh	1/1	Running	0	23s
static-web-replica-t4sjs	1/1	Running	0	23s

```
$ kubectl describe pods static-web-replica-bh8rk
```

```
Name:          static-web-replica-bh8rk
Namespace:     default
Node:          kubenode-1/10.142.0.3
Start Time:    Tue, 02 Jan 2018 09:06:02 +0000
Labels:        app=static-web
Annotations:    <none>
Status:        Running
IP:            10.44.0.1
Controlled By: ReplicationController/static-web-replica
Containers:
  web:
    Container ID: docker://781f3009341dbc5df4beb5b54b815d39063dc2a16737cf2b7e6d71a1b06b3c27
    Image:         nginx
```

```

    Image           ID:                                     docker-
pullable://nginx@sha256:cf8d5726fc897486a4f628d3b93483e3f391a76ea4897de0
500ef1f9abcd69a1
    Port:           80/TCP
    State:          Running
        Started:    Tue, 02 Jan 2018 09:06:11 +0000
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
        /var/run/secrets/kubernetes.io/serviceaccount from default-token-
qtvzc (ro)
Conditions:
  Type           Status
  Initialized    True
  Ready          True
  PodScheduled   True
Volumes:
  default-token-qtvzc:
    Type:          Secret (a volume populated by a Secret)
    SecretName:    default-token-qtvzc
    Optional:      false
QoS Class:       BestEffort
Node-Selectors:  <none>
Tolerations:     node.kubernetes.io/not-ready:NoExecute for 300s
                  node.kubernetes.io/unreachable:NoExecute for 300s
Events:
  Type           Reason            Age    From                      Message
  ----           -
  Normal        Scheduled          1m     default-scheduler        Successfully
assigned static-web-replica-bh8rk to kubenode-1
  Normal        SuccessfulMountVolume 1m     kubelet, kubenode-1      MountVolume.SetUp succeeded for volume "default-token-qtvzc"
  Normal        Pulling            1m     kubelet, kubenode-1      pulling
image "nginx"
  Normal        Pulled             1m     kubelet, kubenode-1      Successfully
pulled image "nginx"
  Normal        Created            59s    kubelet, kubenode-1      Created
container
  Normal        Started            59s    kubelet, kubenode-1      Started
container

```

Now if we make any changes to the rc.yml file we can use command, to apply the changes to the cluster.

```
$ kubectl apply -f rc.yml
```

If we want to run this command with dry-run, we can use it with dry-run=server option

```
$ kubectl apply -f deployment.yaml --dry-run=server
```

Here check the age of container the earlier containers will keep on running and additional gets added.

```

root@kubernetes-master:/home/ghpalnitkar/Kube-project# kubectl apply -f rc.yml
Warning: kubectl apply should be used on resource created by either kubectl create --save-config or kubectl apply
replicationcontroller "static-web-replica" configured
root@kubernetes-master:/home/ghpalnitkar/Kube-project# kubectl get pods
NAME                                READY    STATUS    RESTARTS   AGE
static-web-replica-2nxqp            1/1      Running   0           27s
static-web-replica-bh8rk            1/1      Running   0           21m
static-web-replica-jx7pk            1/1      Running   0           27s
static-web-replica-ksgsg            1/1      Running   0           27s
static-web-replica-m9ck4            1/1      Running   0           21m
static-web-replica-pkd8h            1/1      Running   0           21m
static-web-replica-qwflh            1/1      Running   0           21m
static-web-replica-t4sjs            1/1      Running   0           21m
static-web-replica-v5nzm            1/1      Running   0           27s
static-web-replica-x6dx7            1/1      Running   0           27s
root@kubernetes-master:/home/ghpalnitkar/Kube-project#

```

Similarly if we reduce the number of container, kubernetes will make sure to destroy surplus containers and keep the one running as mentioned in the rc.yml.

```

root@kubernetes-master:/home/ghpalnitkar/Kube-project# kubectl apply -f rc.yml
replicationcontroller "static-web-replica" configured
root@kubernetes-master:/home/ghpalnitkar/Kube-project# kubectl get pods
NAME                                READY    STATUS    RESTARTS   AGE
static-web-replica-bh8rk            1/1      Running   0           25m
static-web-replica-pkd8h            1/1      Running   0           25m
static-web-replica-qwflh            1/1      Running   0           25m
static-web-replica-t4sjs            1/1      Running   0           25m

```

## K8S Services:

We can create a service object imperatively, by running a command as shown below.

```
$ kubectl expose rc static-web-replica --name=web-service --target-port=80 --type=NodePort
```

Here we are creating a service type object 'web-service' to and exposing the replication cluster port 80 of each pod to the service on port 80.

The type of the service that is used is NodePort service.

But running the service using command line is not always the option. We use a YAML file to define the service and tag the service to a replication controller which in turn runs PODs.

Below is such YAML file for Service.



```

apiVersion: v1
kind: Service
metadata:
  name: hello-service
  labels:
    app: hello-world-updated
spec:
  type: NodePort
  ports:
  - port: 8080
    nodePort: 30000
    protocol: TCP
  selector:
    app: hello-world-updated

```

\$ kubectl describe service

\$ kubectl describe ep <same-as-service-name>

## Kubernetes Deployment:

Deployments manage Replica sets and Replica sets manage PODs.

### Deployments (Updates and Rollbacks)

Replica Sets... (Scalability, reliability and desired state)

PODs...[Containers]

**Replication controllers** are replaced by **Replica sets** in deployment object.

```

apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: hello-deploy
spec:
  replicas: 5
  minReadySeconds: 2
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  template:
    metadata:
      labels:
        app: hello-world-updated
    spec:
      containers:
      - name: hello-pod
        image: ganeshhp/maven-petclinic-project:latest
        ports:
        - containerPort: 8080

```

In order to allow PODs to be created on kubernetes master, use below command,

```
$ kubectl taint nodes --all node-role.kubernetes.io/master-
```

## **Horizontal POD Autoscaler (HPA):**

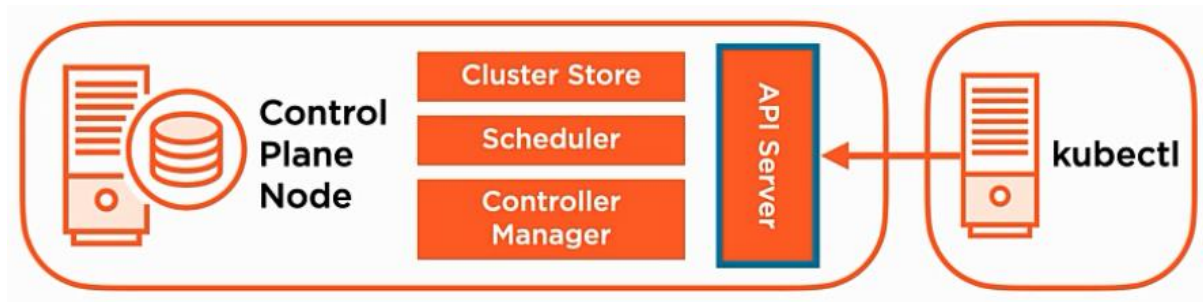
**To AutoScale existing deployment object in cluster.**

```
$ kubectl autoscale deployment.apps/wordpress-deploy --min=2 --max=10
```

```
$ kubectl autoscale deployment.apps/wordpress-deploy --max=10 --cpu-percent=80
```

```
$ kubectl autoscale rc.apache=rc/hello-apache --max=10 --cpu-percent=80
```

## Working with Kubernetes API server and POD objects:



### About the API objects:

API Objects are the persistent entities in the cluster.

The API server represents the state of the objects in the cluster and the cluster store which is usually the 'etcd' service helps in storing the state data for the API objects in the cluster.

The objects in cluster are organized by,

- 1) Kind – POD, Service, Deployment
- 2) Group:
  - a. Core – the objects like PODs
  - b. Apps - objects like deployment that helps in deploying the Application into cluster
  - c. Storage – Objects which are used specifically for storing data from PODs
  - d. Version – API version, version schema that each API objects are related to.
- 3) Some the API objects - Pods, Deployments, Service, Persistent volumes and many more.
- 4) For detailed documentation refer to below link,

<https://kubernetes.io/docs/reference/kubernetes-api>

## Few handy commands...

- 1) Using Command line to create PODS and other Kubernetes objects...

```
$ kubectl run hello-world --replicas=5 --labels="run=load-balancer" --image=gcr.io/google-samples/node-hello:1.0 --port=8080
```

- 2) Setting up environment variables in Container running inside cluster...

```
apiVersion: v1
kind: Pod
metadata:
  name: print-greeting
spec:
  containers:
  - name: env-print-demo
    image: bash
    env:
    - name: GREETING
      value: "Warm greetings to"
    - name: HONORIFIC
      value: "The Most Honorable"
    - name: NAME
      value: "Kubernetes"
    command: ["echo"]
    args: ["$(GREETING) $(HONORIFIC) $(NAME)"]
```

- 3) Creating service object from command line to expose existing deployment.  

```
$ kubectl expose deployment hello-world --type=LoadBalancer --name=my-service
```
- 4) Get information of PODs / namespaces running on a specific node in Kubernetes cluster, use below command.  

```
$ kubectl get pods --all-namespaces --field-selector spec.nodeName=k8s2 -o wide
```
- 5) In case if a nodes is to be taken out of the kubernetes cluster, use below steps,  

```
$ kubectl cordon k8s1
```

Once the Node is disabled for scheduling, the PODs are needed to be shifted from the cordoned node.

```
$ kubectl drain k8s1 --delete-local-data --ignore-daemonsets
```

Now after the nodes os corrected and ready to join the cluster we can 'uncordon' the node and run the deployment manifest file again, so that all PODs are rescheduled on the added Node.

```
$ kubectl uncordon k8s1
```

```
$ kubectl apply -f deployment.yml
```

To get detailed information on running object, use `-o wide` switch, such as,

```
$ kubectl get all -o wide
```

Incase of error faced while initing Kubeadm on VM running on VMware or VirtualBox:

- `kubeadm reset`
- `echo 'Environment="KUBELET_EXTRA_ARGS=--fail-swap-on=false"' >> /etc/systemd/system/kubelet.service.d/10-kubeadm.conf`
- `systemctl daemon reload`
- `systemctl daemon-reload`
- `systemctl restart kubelet`
- `swapoff -a`
- `kubeadm init`

if required run the below command with ignoring error.

- `kubeadm init --ignore-preflight-errors=all`

For many steps here you will want to see what a `Pod` running in the cluster. The simplest way to do this is to run an interactive busybox `Pod`:

```
$ kubectl run -it --rm --restart=Never busybox --image=busybox sh
```

If you don't see a command prompt, try pressing enter.

```
/ #
```

If you already have a running `Pod` that you prefer to use, you can run a command inside the POD using:

```
$ kubectl exec <POD-NAME> -c <CONTAINER-NAME> -- <COMMAND>
```

Create object 'secret'.

```
$ kubectl create secret generic mysql-pass --from-literal=password=my_password
```

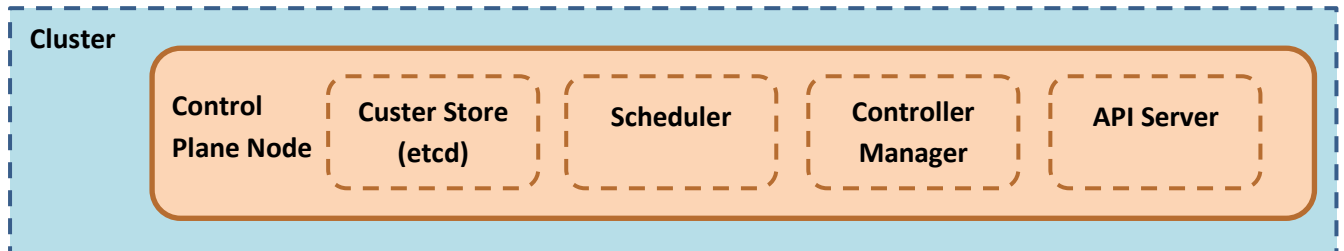
'Secret' being an object can be explored using describe command.

# Managing Kubernetes Controllers and Deployments

Kubernetes Principles:

- 1) Desired state declarative configuration:
- 2) Controllers Control Loops
- 3) The API server

Below mentioned are the components running on a control plane Node.

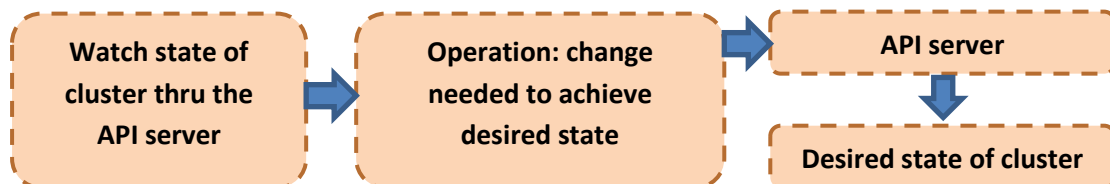


The controller manager makes sure that the desired state of the cluster is maintained using the API server.

- 1) Kube-controller-manager: responsible for maintaining the desired state in the cluster. There's always only one Kube-controller-manager per cluster. Control loops that run inside the manager help in deploying the deployment, service, replica sets resources in the cluster.
- 2) Cloud-controller-manager: this manager helps the cloud service providers to manage the resources in cloud specific kubernetes environment and manage in the cloud centric environment.

## What are controller operations?

- 1) Watch state: the controller watches the cluster state through API server and changes the current state of the cluster into the desired state. The change is also implemented thru the API server.



Two main controller types that help to achieve desired state for workload PODs.

### POD controllers:

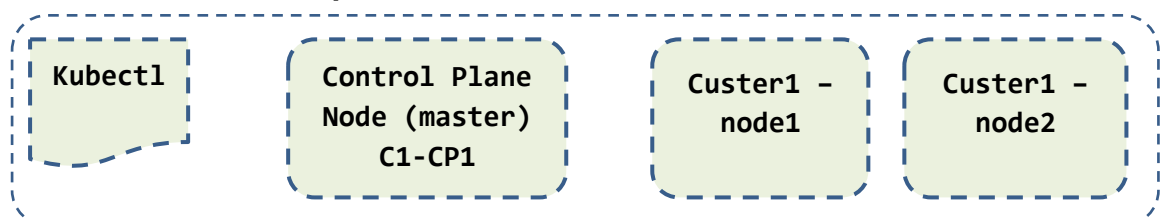
- 1) **ReplicaSet**: maintains the desired number of PODs using POD template in a cluster.

- 2) **Deployment:** manage different versions of POD templates and easily scale and deploy new version. This is core controller process that runs in cluster.
- 3) **DaemonSet:** All or some nodes run a POD that is used in cluster management. Example is the **kube-proxy service, monitoring and log collection agents** services that runs inside PODs that are not workload related POD.
- 4) **StatefulSet:** StatefulSet is the workload API object used to manage stateful applications.  
Manages the deployment and scaling of a set of Pods, and provides guarantees about the ordering and uniqueness of these Pods. Like a Deployment, a StatefulSet manages Pods that are based on an identical container spec. Unlike a Deployment, a StatefulSet maintains a sticky identity for each of their Pods. These pods are created from the same spec, but are not interchangeable: each has a persistent identifier that it maintains across any rescheduling.
- 5) **Job:** A Job creates one or more Pods and will continue to retry execution of the Pods until a specified number of them successfully terminate. As pods successfully complete, the Job tracks the successful completions. When a specified number of successful completion is reached, the task (ie, Job) is complete. Deleting a Job will clean up the Pods it created. Suspending a Job will delete its active Pods until the Job is resumed again.
- 6) **CronJob:** A CronJob creates Jobs on a repeating schedule.  
One CronJob object is like one line of a crontab (cron table) file. It runs a job periodically on a given schedule, written in Cron format.

#### Controller to manage resources other than POD.

- 1) **Node:** A controller to manage the state of each node in the cluster. Is the Node up or down and the state of resources on the node.
- 2) **Service:** creation and deletion of Load Balancers in the cluster infrastructure.
- 3) **Endpoint controller:** managed the service controller to connect to the PODs based on the labels.
- 4) Many more controllers like **Namespace, Route controllers, PersistentVolume** controller are available.

**To start the Practical part, start the Kubernetes cluster:**



Start with examining the system PODs in cluster, run below command,

```
$ kubectl get --namespace kube-system all
```

Now this will give list of all resources running under kube-system namespace.

Let's now get details about the 'coredns' deployment.

```
$ kubectl get --namespace kube-system deployment coredns
```

Now this will give details about 2 coredns PODs running under this deployment.

Now let's look at some of the 'DaemonSet' running in the cluster.

```
$ kubectl get -namespace kube-system daemonset
```

Demonset will run PODs that run services needed for the cluster to run on all nodes in the cluster.

For example the **kube-proxy resource** will run on all nodes in the cluster.

### **The Deployment Controller:**

- Managing Application state with Deployments
  - 1) Creating PODS in cluster
  - 2) Updating application and PODs
  - 3) Scaling up/down number of PODs in the cluster.

**Declaratively:** Preferred way to create deployment resource.

- Writing a deployment Spec in Code (YAML).
  - Selector: defines which POD to be deployed using deployment object.
  - Replicas: number of replicas of POD to be deployed
  - POD Template: technical details like container image, Volume to be mapped to the POD to be deployed.

**Imperatively:** Define and deploy deployment resource at command line. This may not be the preferred way but a quick way to get the application running in cluster. Example:

```
$ kubectl create deployment hello-world --  
image=ganeshhp/helloworld:latest
```

Using above command, we can create the deployment object with one replica. As we have not passed the replicas parameter with value.

To scale the deployment replicas, we can use the command,



```
$ kubectl scale deployment hello-world --replicas=2
```

```
$ kubectl get deployment ... to list deployment resources in the cluster
```

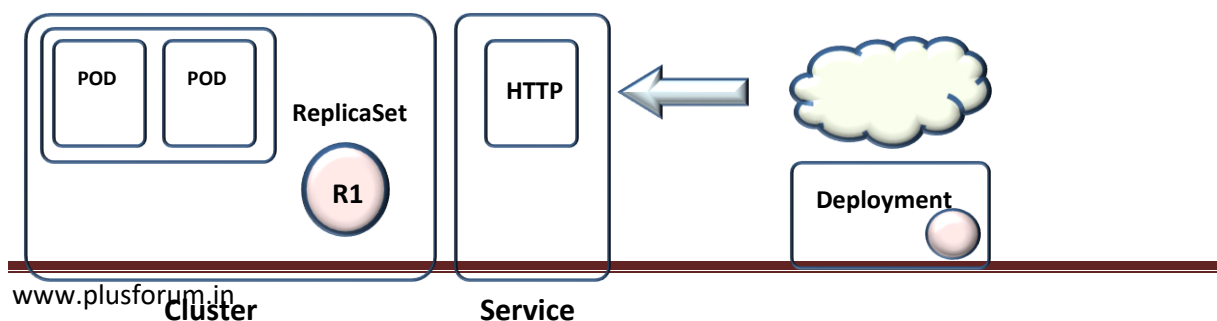
```
$ kubectl delete deployment hello-world ... in case we want to remove /  
delete the resource, we use delete command in kubectl options.
```

**Declaratively:** write a manifest file as shown below.

```
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: hello-world  
spec:  
  replicas: 4  
  selector:  
    matchLabels:  
      app: hello-world  
  template:  
    metadata:  
      labels:  
        app: hello-world  
    spec:  
      containers:  
        - name: hello-world  
          image: ganeshhp/helloworldweb:latest  
          ports:  
            - containerPort: 8080
```

-----

```
apiVersion: v1  
kind: Service  
metadata:  
  name: hello-world  
  labels:  
    app: hello-world  
spec:  
  selector:  
    app: hello-world  
  ports:  
    - port: 80  
      protocol: TCP  
      targetPort: 8080
```



Now, to deploy the deployment resource declaratively as written in manifest file, use the command,

```
$ kubectl create -f deployment.yml ... here we can use create or apply command in kubectl.
```

As the manifest file has deployment and service resource definitions, we will get both resources deployed with create action.

```
$ kubectl get deployments .. To see deployment resources status
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
hello-world	1/1	1	1	3d18h

```
$ kubectl get replicaset
```

Command to see ReplicaSet resource created by the deployment object. ReplicaSet is responsible for keeping the number of PODs created and available.

NAME	DESIRED	CURRENT	READY	AGE
hello-world-6f8d5c6d89	1	1	1	3d18h

```
$ kubectl get all ..
```

This will show all available resources deployed in default namespace.

If we look at the description of a POD that's created using Deployment / ReplicaSet, we can see the information about resource that is controlling the POD.

```
$ kubectl describe pod --namespace=default | head -n=15
```

```
Name:          hello-world-6f8d5c6d89-rdb2c
Namespace:     default
Priority:       0
Node:          minikube/192.168.99.100
Start Time:    Sun, 19 Dec 2021 00:01:07 +0530
Labels:        app=hello-world
               pod-template-hash=6f8d5c6d89
Annotations:   <none>
Status:        Running
IP:            172.17.0.3
IPs:           172.17.0.3
```

```
Controlled By:  ReplicaSet/hello-world-6f8d5c6d89
```

Containers:

## Understanding Replicasets:

### What does the ReplicaSet provides:

A ReplicaSet's purpose is to maintain a stable set of replica Pods running at any given time. As such, it is often used to guarantee the availability of a specified number of identical Pods.

- 1) Deploys a defined number of PODs to maintain the desired state of PODs in the cluster.
- 2) **Consists of Selector, Number of Replicas, and POD templates**
- 3) As a practice we do not create ReplicaSet directly, but instead is created by the deployment resource while we define number of replicas in the Deployment resource definition. It is the underlying building block of deployment object.
- 4) As part of ReplicaSet definition, we can use different type of selectors. As shown below. *matchLabels* and *matchExpressions*

```
apiVersion: apps/v1
kind: ReplicaSet
...
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hello-world-pod
  template:
    metadata:
      labels:
        app: hello-world-pod
    spec:
      containers:
        ...
```

```
apiVersion: apps/v1
kind: ReplicaSet
...
spec:
  replicas: 1
  selector:
    matchExpressions:
      - key: app
        operator: In
        values:
          - hello-world-pod-me
  template:
    metadata:
```

How ReplicaSet handled failure.

While in the cluster if there's a POD failure, the ReplicaSet send a request to the APIserver to terminate the failed POD and create a new POD in that place.

In case if the Node failure happens, this is treated in a different way. The Node failure can be a transient failure or a permanent failure. Transient failure can be seen as a brief unavailability of the node due to network or hardware issue. If the node is not reachable then the Containers running on that node are assigned with error status. If the nodes comes up again in some time and if the POD / Containers are not running, then the **POD restart policy** is applied using the kube-controller manager and the PODs are restarted as the PODs are still scheduled on that Node.

Now if the Node is down for a longer time, intentionally taken down then the Kube-Controller manager has a pod-eviction-timeout setting that is set to 5 minutes default value. After 5 minutes the PODs on the failed node are unscheduled and recreated on available Node in the cluster to match the desired state of POD replicas.

## Demo for ReplicaSet:

We have a YAML file (as stated in this document on page 41) that has the Deployment and Service object defined. Let's deploy the objects with create command.

```
$ kubectl create -f deployment.yaml
```

```
$ kubectl get replicaset
```

NAME	DESIRED	CURRENT	READY	AGE
hello-world-5646fcc96b	5	5	5	14s

```
$ kubectl describe replicaset hello-world
```

```
Name:          hello-world-5646fcc96b
Namespace:     default
Selector:      app=hello-world,pod-template-hash=5646fcc96b
Labels:        app=hello-world
               pod-template-hash=5646fcc96b
Annotations:   deployment.kubernetes.io/desired-replicas: 5
               deployment.kubernetes.io/max-replicas: 7
               deployment.kubernetes.io/revision: 1
Controlled By: Deployment/hello-world
Replicas:      5 current / 5 desired
Pods Status:   5 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  app=hello-world
           pod-template-hash=5646fcc96b
```

Now after checking the details about the replicaset object, let's delete the Deployment object which in turn will remove the ReplicaSet object as well.

Now, after deleting the Deployment object let's **create a new one**, but this time with `matchExpressions` instead of `matchLabels`.

```
$ kubectl create -f deployment-me.yaml
```

```
-----
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-world
spec:
  replicas: 4
  selector:
    matchExpressions:
      - key: app
        operator: In
        values:
          - hello-world-pod-me
  template:
    metadata:
      labels:
```

```

        app: hello-world
spec:
  containers:
  - name: hello-world
    image: ganeshhp/helloworldweb:latest
    ports:
    - containerPort: 8080

```

-----

Let get details of the deployment object.

```
$ kubectl describe deployment/hello-world
```

```

Name:          hello-world-f597dc95
Namespace:     default
Selector:      app in (hello-world-pod-me),pod-template-hash=f597dc95
Labels:        app=hello-world-pod-me
               pod-template-hash=f597dc95
Annotations:   deployment.kubernetes.io/desired-replicas: 5
               deployment.kubernetes.io/max-replicas: 7
               deployment.kubernetes.io/revision: 1
Controlled By: Deployment/hello-world
Replicas:      5 current / 5 desired
Pods Status:   5 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  app=hello-world-pod-me
          pod-template-hash=f597dc95
  Containers:
    hello-world:
      Image:      gcr.io/google-samples/hello-app:1.0
      Port:       8080/TCP
      Host Port:  0/TCP
      Environment: <none>

```

Now to see if the replicaset creates a POD in place of a failed / deleted POD, try deleting a POD from the running list of PODs. Also try changing Label of a running pod by using below command and note how replicaset creates a new PODs in the place of non matching POD.

```
$ kubectl label pod hello-world-[tab]-[tab] app=DEBUG -
overwrite
```

Now check if the labels are changed, by running below command,

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

Now as soon as a POD has a POD with different label, not matching the label to the replicaset, then a new POD will get created to match required number of PODs matching the labels.

Now let's change the Label back again to original state for the POD, we will see that to match the number of PODs in cluster matching the Label, one of the POD will get deleted if required.

## Node Failure Operation:

Now what if a node fails in the cluster.. check that with forcefull stopping a Node and observing changes in the cluster.

For this we would need atleast two worker nodes in the cluster.

1) Shutdown one of the node of the worker node in the cluster.

```
$ shutdown -h now
```

While the node is shutting down, we can watch the status of cluster using,

```
$ kubectl get nodes --watch
```

This will report the node status as 'NotReady'.

NAME	STATUS	ROLE	AGE	VERSION
node1	NotReady	<none>	31d	v1.13.1

Now while the node is shutdown, the Controller still treats this as a **transient error** and waits for 5 minutes (default) time till the node comes back again. During this time if the node doesn't come back the POD status will report below.

NAME	READY	STATUS	RESTARTS	AGE
hello-world-f597dc95-fcs8z	0/1	Error_	0	6m55s

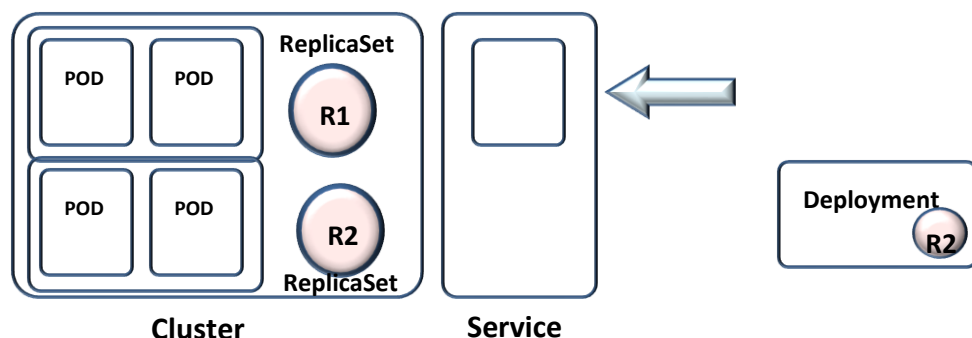
After some time when the node recovers and is reachable within the timeline of 5 minutes, we will see the POD getting back again after the controller attempts a restart.

NAME	READY	STATUS	RESTARTS	AGE
hello-world-f597dc95-fcs8z	1/1	Running	1	7m21s

Now let's see if the nodes is permanently shutdown and doesn't come back in the 5 minutes ([pod-eviction-timeout](#)) timeline window. The POD will get deleted from the unavailable Node and a new POD will get deployed on the available POD to match the required replica numbers.

## Deploying and Maintaining Applications with Deployment object.

- 1) Updating [Deployments](#)
- 2) Controlling Rollouts
- 3) Scaling Applications



The Deployment object helps in updating the application without compromising the application availability. When the specification of the POD templates changes, the update is triggered and causes a new ReplicaSet to be created and this makes sure to delete and remove a POD from earlier replicaset and create new set of PODs with new replicaset.

### Updating deployment object:

Now, to update already deployed deployment object, we can use different ways, first we can use **set image** command.

```
$ kubectl set image deployment hello-world hello-world=web-app:2.0
```

Object type and object name      Image name

We also add a **--record** flag to this command to add annotation, which can later be used for comparing earlier version of deployment to newer version and then help in rollback of the deployment.

```
$ kubectl set image deployment hello-world hello-world=new-app:2.0 --record
```

Similarly we can use edit command to edit the object that is defined in a manifest file.

```
$ kubectl edit deployment hello-world
```

With this command the api-server is presented with request to get details of the object and open that in an editor.

And finally, we have the **kubectl apply -f** command to apply changes to existing object using changes mentioned in manifest file. We can export definition of an object into a YAML file and then edit the YAML file and send it back to the API-Server.

```
$ kubectl apply -f hello-world-deployment.yaml --record
```

### Checking deployment status

While we are updating a deployment object, we can check the status of update by using **rollout** and **describe** command option.

**rollout** Available Commands:

history	View rollout history
pause	Mark the provided resource as paused
restart	Restart a resource
resume	Resume a paused resource
status	Show the status of the rollout
undo	Undo a previous rollout

```
$ kubectl rollout status deployment hello-world
```

```
$ kubectl describe deployment hello-world
```

Deployment status.

- 1) **Complete** – all update work in finished
- 2) **Progressing** – update in progress
- 3) **Failed** – update could not be completed.

When we are deploying a new version of deployment (image version) using declarative way, when we deploy the YAML file, it creates a new ReplicaSet and stop all PODs started by older ReplicaSet. Both ReplicaSets will continue to exists till we remove it manually. We can use this for undoing new deployment if needed. Use below command,

```
$ kubectl get replicaset
```

NAME	DESIRED	CURRENT	READY	AGE
hello-world-54875c5d5c	10	10	10	6m41s
hello-world-5646fcc96b	0	0	0	9m21s

Now try getting information on the new replicaset

```
$ kubectl describe replicaset hello-world-54875c5d5c
```

Also try getting description for the earlier replicaset. This should show POD status as deleted.

## Using Deployments to Change State:

Control rollouts of a new versoin of your applications.

- Update strategy
- Pause rollout to update and make correction toan object
- Rollback to an earlier version
- Restart deployments

### Update Strategy:

#### RollingUpdate (default)

- A new ReplicaSet starts scaling up and,
- old ReplicaSet start scaling down.

#### Recreate

Terminates all PODs in the current ReplicaSet and Set priot to scaling up new Replicas Set. When application doesn't support running different version concurrently.



## RollingUpdate strategy:

**maxUnavailable** ... ensures only a certain number of PODs are made unavailable. (Default is 25%)

**maxSurge**... ensures only a certain number of PODs are new created above the number of desired state. (Default us 25%)

## Controlling Deployment Rollout

With using **rolling update strategy** and **POD readiness probe** defined, we can control the deployment of POD in a deployment object.

```
apiVersion: apps/v1
kind: Deployment
---
spec:
  replicas: 20
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable:
      maxSurge: 5
  ---
  template:
    ---
    spec:
      containers:
        ---
        readinessProbe
          httpGet:
            path: /index.html
            port: 8080
          initialDelaySeconds: 10
          perdiosSeconds: 10
```

## Pausing and Resuming a Deployment:

- 1) While the deployment is paused, changes are not rolled out.
- 2) We then can batch multiple changes together and then resume the deployment
- 3) Until the deployment is paused, the current state of deployment is maintained
- 4) When we resume the deployment and if we have added new changes to the deployment, then a new Replicaset is created and deployed with resumed deployment.  
Imperatively this can be done by running command,

```
$ kubectl rollout pause deployment my-deployment
```

```
$ kubectl rollout resume deployment my-deployment
```

## Rolling back a Deployment

- 1) In case there's application failure after rolling out a new version and in order to recover from failure, we have only option to revert back the changes pushed to the environment, then we can use the **rollback** option to rollback the deployment. This is possible as Kubernetes maintains a history of all deployment.
- 2) In Kubernetes deployment, every deployment object rolled-out has a annotation. This is maintained as the history.
- 3) By default the revision history is maintained for 10 deployment revisions. This value can be changed in deployment configuration. The revision history number to be set to 0 as well, this will help to clean up all deployment revision info as soon the deployment is created.
- 4) To perform a rollback, we need to have information about revision history and this can be queried by running command,

```
$ kubectl rollout history deployment my-deployment
```

Once we get the information, we want to know details about which application is rolled out in a particular version, then we can get detailed info with below command.

```
$ kubectl rollout history deployment my-deployment --revision=1
```

Now to perform the rollback of the deployment to earlier revision we use command,

```
$ kubectl rollout undo deployment my-deployment
```

To perform rollback of deployment to a particular version, we use below command,

```
$ kubectl rollout undo deployment my-deployment --to-revision=1
```

In this case Kubernetes **scale up** the old replicaset and **scale-down** the new replicaset for the revision selected

### Restarting a deployment:

In case we want to restart the PODs in a deployment object, may be due to some environment change etc., we can use the **restart** option in the command as show below.

```
$ kubectl rollout restart deployment my-deployment
```

In this case the PODs are not actually restarted, but the restart option causes a new replicaset to create with same POD definition, and scale down old replica set.

## Demonstration:

To demonstrate the deployment rollback, we can use a deployment manifest to rollout a particular version. Then we make changes to the manifest with a broken container image, causing the container to fail. To recover from this failed rollout we use rollback strategy as shown above.

Create a file `deployment-probe.yaml`

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-world
spec:
  replicas: 10
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 10%
      maxSurge: 2
  revisionHistoryLimit: 20
  selector:
    matchLabels:
      app: hello-world
  template:
    metadata:
      labels:
        app: hello-world
    spec:
      containers:
        - name: hello-world
          image: httpd:latest
          ports:
            - containerPort: 8080
          readinessProbe:
            httpGet:
              path: /index.html
              port: 8080
            initialDelaySeconds: 10
            periodSeconds: 10
```

```
$ kubectl apply -f deployment-probe.yaml --record
```

Using record we can keep the revision history along with annotation and thus using the revision history we can rollback the deployment rolled out.

### Scaling Deployments:

- Manually... by specifying number of replicas to scale to.

```
$ kubectl scale deployment hello-deploy --replicas=10
```

Or,

```
$ kubectl apply -f deployment.yaml
```

- Horizontal Pod Autoscaler

```
$ kubectl autoscale deployment/hello-deploy --max=5 --min=2  
--cpu-percent=80
```

### Demonstration:

```
$ kubectl create deployment hello-world --  
image=ganeshhp/hello-world:latest
```

```
$ kubectl get deployment hello-world
```

```
$ kubectl scale deployment hello-world --replicas=10
```

```
$ kubectl get deployment hello-world
```

Now, the same change can be achieved declaratively as well, by updating the manifest (yaml) file. In the spec section of deployment object, we have replicas set to 10, we change the number to 15. If we run this on environment by using,

```
$ kubectl apply -f deployment-replicas-15.yaml
```

After running the above command, we check the status by running `kubectl get deployments` command.

Also check the status by running, `describe` command on the deployment.

### Deployment Tips – how to control:

- 1) Control the rollout with an update strategy appropriate for your application.
- 2) Use Readiness Probes for your application.
- 3) Use the `--record` option to leave a trail of work for others.

## Some additional Controller types in Kubernetes:

Deployment and Maintaining application using

- 1) **DaemonSets**
- 2) **Jobs and CronJobs controllers.**
- 3) **StatefulSets**

What's a **DaemonSet**: A DaemonSet is an api in kubernetes that ensures a single POD always runs on a node or subset of Nodes. These are like kubernetes managed system daemons which is like **systemd** or **initd** in Linux.

A **Job** is a Kubernetes API that ensure to run a POD or a task once inside the cluster.

To schedule a **Job** in Kubernetes cluster we have **CronJobs** as api object in kubernetes to run the job at a specified schedule.

**StatefulSets** is a api object in kubernetes that provides required infrastructure to run a statefull application. **StatefulSets** are valuable for applications that require one or more of the following.

- Stable, unique network identifiers.
- Stable, persistent storage.
- Ordered graceful deployment and scaling.
- Ordered, automated rolling updates.

### 1) **Daemonsets:**

This API ensures that all or some nodes run a POD, effectively an init daemon inside your cluster.

Examples,

- [Kube-proxy](#) for network service
- Log collectors
- Metric servers
- Resource monitoring agents
- Storage daemons

Defining a daemonset:

[DaemonSet.yaml](#)

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: hello-world-ds
spec:
  selector:
    matchLabels:
      app: hello-world-app
```

```

template:
  metadata:
    labels:
      app: hello-world-app
  spec:
    containers:
      - name: hello-world
        image: gcr.io/google-sample/hello-app:1.0

```

Defining a DaemonSet with a **node selector** so that the daemonset POD can run on a specific Node or set of nodes.

#### DaemonSet-node.yaml

```

apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: hello-world-ds
spec:
  selector:
    matchLabels:
      app: hello-world-app
  template:
    metadata:
      labels:
        app: hello-world-app
    spec:
      nodeSelector:
        node: hello-world-ns
      containers:
        - name: hello-world
          image: gcr.io/google-sample/hello-app:1.0

```

Here the label needs to be assigned to the Nodes on which we want the daemonset POD to be running and same is used in `nodeSelector` statement.

### **DaemonSet Update strategy:**

Default update strategy is **RollingUpdate** and default maxUnavailable integer is **1**.

**OnDelete** update strategy will cause the Daemonset to be updated to new template only on deletion of the earlier version. If we want to upgrade the Daemonset to a new version, update the manifest YAML with desired changes and Daemonset will push the changes to the cluster allowing the desired state configuration to be maintained. In this there is no option to pause the upgradation of daemonset like the one in deployment api object.

## Demo:

In this demo we will create a **DaemonSet** with one **POD** on each **Node** and then we will create one POD on selected Node.

First, let's get the list of available Nodes in the cluster.

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
docker-desktop	Ready	control-plane,master	10d	v1.22.5

Now we will also check the default **DaemonSet**, i.e. the **kube-proxy** that runs in the cluster. This can be checked by running command,

```
$ kubectl get daemonsets --namespace kube-system kube-proxy
```

NAME	DESIRED	CURRENT	READY	UP-TO-DATE	AVAILABLE	NODE SELECTOR	AGE
kube-proxy	1	1	1	1	1	kubernetes.io/os=linux	10d

Here, we are querying the `kube-system` namespace for the `kube-proxy` daemonset.

Now let's deploy a Daemonset using yaml file.

```
$ kubectl create -f DaemonSet.yaml
```

Verify daemonset status

```
$ kubectl get daemonsets
```

Verify where the daemonset PODs are running.

```
$ kubectl get daemonsets -o wide
```

```
$ kubectl get pods -o wide
```

Now, to get information on Callout, labels, desired/current state Nodes scheduled, Pod status, template, events.

```
$ kubectl describe daemonset hello-world-ds
```

Now let's try to change the Label for a POD and observe if the Daemonset creates a new POD with matching Label.

```
$ kubectl label pods <podname> app=not-hello-world-ds --  
overwrite
```

After we run this command, you will notice one new POD been created in place of the changed label POD., so now there are more than one PODs running on the node.

Now, lets try creating **DaemonSet** on only a subset of nodes and not on all worker nodes.

First remove all Daemonsets from earlier run. And also remove the POD that was taken out of Daemonset.

```
$ kubectl delete -f DaemonSet.yaml
```

### Creating DaemonSet on subset of Nodes:

```
$ kubectl apply -f DaemonSet-node.yaml
```

The file is updated with a NodeSelector statement. After running this DaemonSet you will notice that the POD is not scheduled yet. This means we need to Label the Nodes as well to match in the **nodeselector**.

In this case we will label the node docker-desktop with the matching Label as **app:hello-world-ns**

```
$ kubectl label node docker-desktop app:hello-world-ns
```

Now, let's chake the status of the daemonset.

```
$ kubectl get daemonsets
```

NAME	DESIRED	CURRENT	READY	UP-TO-DATE	AVAILABLE	NODE SELECTOR	AGE
hello-world-ds	1	1	1	1	1	node=hello-world-ns	60s

Here, we see the daemonset is running and available.

Now if we label the node back again with label not machting to the daemonset label, let's see what happens.

```
$ kubectl label node docker-desktop node-
```

The POD will get deleted as the matching node is not found in the cluster. The same can be seen with the daemonset describe command.

```
$ kubectl describe daemonsets hello-world-ds
```

Events:					
Type	Reason	Age	From	Message	
Normal	SuccessfulCreate	67s	daemonset-controller	Created pod: hello-world-ds-qfd5r	
Normal	SuccessfulDelete	5s	daemonset-controller	Deleted pod: hello-world-ds-qfd5r	



Now, let's see how we can **update a daemonset** using update strategy. In this case the yaml file has been updated with a new image info and deploying a new daemonset is done using running the yaml file.

```
$ kubectl apply -f DaemonSet-update.yaml
```

```
$ kubectl describe daemonsets hello-world.
```

In this file we haven't mentioned the update strategy. The default **daemonset** update strategy is **rollingupdate**. We can see that by running a command, check the events to see the details.

```
$ kubectl get Daemonsets hellow-world-ds -o yaml | more
```

After we complete all activities related to demonstrating Daemonset, clear the daemonsets using

```
$ kubectl delete -f daemonset-ns.yaml
```

## 2) JOBS:

Controller objects so far introduced ([ReplicaSet](#), [Deployment](#), [DaemonSet](#)) allow to start and run the PODS in the cluster continuously and stay in that state to maintain the Desired State Configuration (DSC).

But there are requirements when we do want to run a task just once or periodically, but not continuously like running a POD in cluster continuously, for this we have the [Job](#) or [CronJob](#) objects.

- **Jobs** create one or more Pods,
- Runs a program in a container to completion.
- Ensures that a specified number of Pods completes successfully.
- Workload examples.
  - o Ad-hoc tasks
  - o Batch job to be executed inside the cluster
  - o Data oriented tasks like moving data from systemA to SystemB.

The Job controller object helps to run the POD to completion. In case of a interrupted execution the POD, like in case of a node failure, then the JOB controller has the task to reschedule the POD to a different node. Or if the application running inside the POD returns a Non-Zero exit code, then the Job controller kicks in the the POD **restartPolicy**. The default restartpolicy is **Always**, we should change it to 'OnFailure'.

If the Job completes successfully, the status is set to 'Completed'.

Let's write a yaml file, `job.yaml`

```
apiVersion: batch/v1
kind: Job
metadata:
  name: hello-world-job
spec:
  template:
    spec:
      containers:
      - name: ubuntu
        image: ubuntu
        command:
        - "/bin/bash"
        - "-c"
        - "/bin/echo Hello From Pod $(hostname) at $(date)"
      restartPolicy: Never
```

Controlling the JOB execution:

- 1) backOffLimit – number of job retries before it's marked as 'failed'.
- 2) activeDeadlineSeconds: max execution time for the job
- 3) parallelism – max number of running PODs in a job at a point of time.
- 4) Completion - number of PODs that need to finish successfully.

#### Demonstration:

Let's first set the '`restartPolicy` to `OnFailure`'

```
$ kubectl apply -f job.yaml
```

Follow the job status with a `watch`

```
$ kubectl get job --watch
```

NAME	COMPLETIONS	DURATION	AGE
hello-world-job	0/1	9s	9s
hello-world-job	1/1	22s	22s

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
hello-world-job--1-rj9tw	0/1	Completed	0	3m28s

This indicates, the Pods started by the task completed successfully.  
Try to run the describe command on the pod created by the Job.

```
$ kubectl describe job hello-world-job
```

Events:				
Type	Reason	Age	From	Message
----	-----	----	----	-----
Normal	SuccessfulCreate	61s	job-controller	Created pod: hello-world-job--1-rj9tw
Normal	Completed	39s	job-controller	Job completed

```
$ kubectl logs hello-world-job--1-rj9tw
```

```
PS C:\Users\Lenovo> kubectl logs hello-world-job--1-rj9tw
Hello from POD hello-world-job--1-rj9tw at Sun Feb 27 14:29:05 UTC 2022
```

So from this demonstration we are able to see that the job helped us to get a POD started with a container running specific commands and after the container completed execution of the commands, the container stops in other words the Pod completes its lifecycle.

As we have set the restartPolicy value to 'Never', the POD doesn't restart on completion. If we set this to [OnFailure](#), we can expect the POD to restart automatically on failed POD in the JOB.

A Job can also be executed with parallel option allowing multiple PODs to run in parallel.

### paralleljob.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: hello-world-job
spec:
  completions: 50 # allows us to run 50 number of PODs in all as part of the JOB
  parallelism: 10 # allows to run 10 PODs in parallel
  template:
    spec:
      containers:
      - name: ubuntu
        image: ubuntu
        command:
        - "/bin/bash"
        - "-c"
        - "/bin/echo Hello From Pod $(hostname) at $(date)"
      restartPolicy: Never
```

```
$ kubectl apply -f paralleljob.yaml # this will start creating 10 PODs as part of the job.
```

### CronJobs:

- CronJob will run a Job on a given time based schedule
  - o Conceptually CronJob is similar to Linux / Unix Cron utility.

- It used the standard cron format.

Example workloads –

- 1) Periodic workload or scheduled tasks
- 2) CronJob resource is created when the object is submitted to the API server
- 3) When it's time, a Job is created via the Job template from the CronJob Object.

Controlling CronJob Execution:

- Schedule – a cron formatted schedule
- Suspend – suspends the cron job
- startingDeadlineSeconds – the Job that hasn't started on this amount of time is marked as failed.
- ConcurrencyPolicy – handled concurrent execution of Job, allowed values are 'Allow', 'Forbid' and 'Replace'

### cronjob.yaml

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: hello-world-cron
spec:
  schedule: "*/1 * * * *" # here the job will get scheduled every minute
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: ubuntu
              image: ubuntu
              command:
                - "/bin/bash"
                - "-c"
                - "/bin/echo Hello From Pod $(hostname) at $(date)"
          restartPolicy: Never
```

Demonstration:

```
$ kubectl apply -f cronjob.yaml
```

Here we will be to see job getting scheduled in turn starting POD every minute and thus PODs starting → completing → creating new

**StatefulSet:**

This api object helps to maintain a statefull application to run inside a POD, i.e. we will be able to run a statefull application with help persistent storage, persistent Naming, or Headless Service.

Service / application like databse service need persistent access to the datafile.

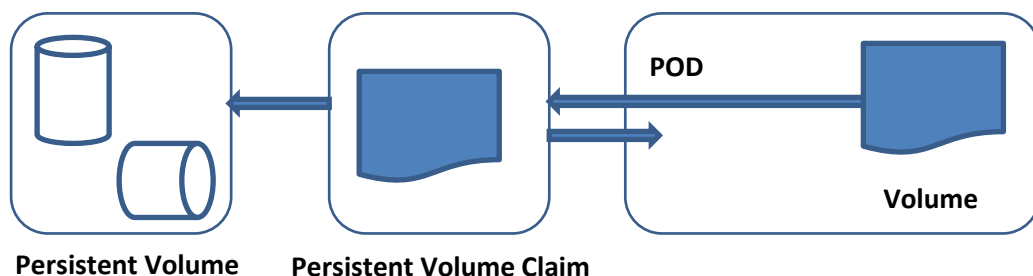
# Configuring and Managing Kubernetes Storage and Scheduling

## Persistent storage and containers:

The containers are ephemeral in nature. Container writable layer is destroyed when a container is deleted. When a POD is deleted, its container(s) is deleted from the node, so if there is any data written / stored inside the container in a POD, it will get lost with deleting the POD.

## Storage API objects in Kubernetes

- 1) **Volume** : This is the actual storage, like AWS elastic storage, or a folder. This is defined inside the POD spec.
- 2) **Persistent Volume (PV)**: This is the actual storage available inside the cluster for the use inside a POD.
- 3) **Persistent Volume Claim (PVC)**: Persistent Volume claim is the actual request made by the user for using the PersistentVolume inside a POD. Without having a PVC, one can not use the PV inside a POD.
- 4) **Storage Class**: this can be seen group of storage for different storages (PVs) available inside kubernetes cluster.



## Volume:

- a. Persistent storage deployed as part of the POD specification in the POD manifest along with technical specification of the storage, like if it is NFS, we will have to provide NFS server DNS name and path.
- b. The volume declaration also have to provided with access information for the storage. This can pose few challenges.
  - a. The code (manifest) for the POD can not be made portable as the technical details for the volume have to be defined inside the POD spec, making it not easy for portability between diff environments.
  - b. Volumes also have the same lifecycle like the POD. So if a POD is removed / deleted, then the volume also gets deleted.
  - c. To over come these challenges, we use the PVC that is easily portable as well.

## Persistent Volume:

Persistent volume is defined by the cluster administrators and is mapped to the actual storage, like NFS, Cloud storage, storage disk etc.. So it is the responsibility of the administrator to create or delete the storage **API object**. In the PV API object definitions we will have to provide technical specification of the storage, like NFS server DNS, Path etc.

As this is an independent object, if the POD is deleted the PV still lives on, it is persistent in nature.

The actual storage to be mapped on the node is managed by the Kubelet service.

Check below link for more detailed information on PVs.

<https://kubernetes.io/docs/concepts/storage/persistent-volumes/>

Types of Persistent volumes.

These can be broadly into, Networked storage, Block and Cloud storage. And some sample types of PVs are as listed below, but there are a lot more to this list.

- **Networked: NFS, azurefile**
- **Block: Fibre Channel, iSCSI**
- **Cloud: awsElasticBlockStore, azuredisk, gcePersistentDisk**

Check below link for more details.

[Persistent Volumes | Kubernetes](#)

## Persistent Volume Claims

The Persistent Volume Claim is an API object which is basically a request for storage by a user to be used inside a POD. The PVC is defined with some technical specification like, Size, Access Mode (readWriteOnce, readWriteMany, readOnlyMany), Storage Class.

Using PVC adds portability of the application configuration inside a POD.

When we deploy the PVC object, the cluster will map a PVC to an existing PV in static provisioning of volume depending upon the size, access mode and class requested.

### Access Modes in PV:

The access mode defines how a persistent volume will be accessed by multiple Nodes or a Node. Here we say node because a Persistent Volume is mapped to a Node or Nodes with deployment of the PV API object.

- 1) ReadWriteOnce (RWO)
- 2) ReadWriteMany (RWX)
- 3) ReadOnlyMany (ROX)
- 4) ReadWriteOncePod (RWOP) . if we want only one POD in the cluster to allow accessing the PV.

In case if the actual storage has a different access characteristics like if the NFS has read only characteristic then this setting will supersede the API object access mode setting.



## Static and Dynamic Provisioning of Persistent Volumes:

### Workflow:

Create a **PersistentVolume** → Create a **PersistentVolumeClaim** → define **Volume** in **POD Spec**.

Defining all the objects manually is what the static provisioning is. In case of Dynamic provisioning of PV, A PersistentVolume is not deployed manually but is created automatically at the time of mapping a PVC in a POD definition, if the PV is not already present that matches the requested specification of a PV then the dynamic provisioning is used.

### Storage Lifecycle in Kubernetes cluster:

Binding 	Using 	Reclaim
<p>In the Binding process once a PVC is Created, the control loop find a PV that Matches PVC to PV, depending on size, access mode and class requested.</p> <p>If a PV is not present that matches to the specification, then the request goes into pending in state and it binds to the PV whenever a PV is available.</p>	<p>Once the binding is done the PVC and PV mapped till PVC's lifetime.</p> <p>Using the PVC is done by using the PVC in a POD's specification and that stays till the lifetime of the POD.</p>	<p>As there is a one-to-one mapping of the PVC and PV, if the PVC gets deleted, the PV can be deleted (<u>often incase of dynamic provisioning of PV</u>) or retained.</p> <p>Once a PVC is deleted the PV can be reused / Reclaims as specified in the PV reclaim policy.</p> <p><b>Reclaim policy</b> options can be <b>Delete</b> (Default), <b>Retain</b> or <b>Recycle</b>.</p>

### Defining a Persistent Volume:

#### nfs-pv.yaml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-nfs-data
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Retain
  nfs:
```



```
path: "/export/volumes/pod"
server: 172.17.0.2
```

## Defining Persistent Volume Claim

In a PVC definition there multiple specifications under which a PV gets mapped. accessMode, resources, storageClassName. Selector.

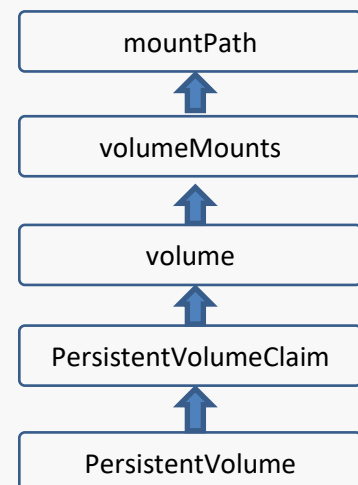
### nfs-pvc.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-nfs-data
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 5Gi
```

## Using the Persistent Volume in PODs:

### nfs-https-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  volumes:
    - name: website
      persistentVolumeClaim:
        claimName: pvc-nfs-data
  containers:
    - name: myfrontend
      image: httpd:latest
      volumeMounts:
        - name: website
          mountPath: "/var/www/html"
```



**LAB:** environment (Ubuntu 18.04, VirtualBox VMs, 2vCPUs, 2GB RAM, 50 GB. SWAP disabled)

### Demo:

- 1) Storage Server overview
- 2) Static provisioning Persistent Volume
- 3) Storage Lifecycle and reclaims Policy.

Setting up the NFS server:

```
$ sudo apt-get install nfs-kernel-server
```

```
$ sudo mkdir /export/volumes
```

```
$ sudo mkdir /exports/volumes/pod
```

Now to configure the NFS server, we have the config file at `/etc/exports`.

Let's run below command,

```
$ sudo bash -c 'echo "/export/volumes  
*(rw,no_root_squash,no_subtree_check)" > /etc/exports'
```

```
$ cat /etc/exports
```

```
$ sudo systemctl restart nts-kernel-server.service
```

Now, In order the NFS export to be accessed on the cluster nodes we will also have to install the `nfs-common` (client package).

```
$ sudo apt-get install nfs-common
```

Now, check if we can mount the NFS server share on the node.

```
$ sudo mount -t nfs4 <nfs-sever-dns>:/export/volumes /mnt/
```

```
$ mount | grep nfs
```

```
$ sudo umount /mnt
```

Now let's start with deploying the YAML files starting with deploying the PV.

```
$ kubectl apply -f nfs-pv.yaml
```

```
$ kubectl get PersistentVolume pv-nfs-data
```

To get more details,

```
$ kubectl describe PersistentVolume pv-nfs-data
```

Now that we have the PV created, let's deploy the PVC,

```
$ kubectl apply -f nfs-pvc.yaml
```

```
$ kubectl get PersistentVolumeClaim pvc-nfs-data
```

Check the status of PV which show the PV is bound

To get more details.

```
$ kubectl describe PersistentVolumeClaim pvc-nfs-data
```