Kubernetes

What is Kubernetes?

Kubernetes is an open source container orchestration platform developed by Google for managing microservices or containerized applications across a distributed cluster of nodes. Kubernetes is highly resilient and supports zero downtime, rollback, scaling, and self-healing of containers. The main objective of Kubernetes is to hide the complexity of managing a fleet of containers.

Features of Kubernetes

Following are some of the important features of Kubernetes.

- Continues development, integration and deployment
- Containerized infrastructure
- Application-centric management
- Auto-scalable infrastructure
- Environment consistency across development testing and production
- Loosely coupled infrastructure, where each component can act as a separate unit
- Higher density of resource utilization
- Predictable infrastructure which is going to be created

One of the key components of Kubernetes is, it can run application on clusters of physical and virtual machine infrastructure. It also has the capability to run applications on cloud. **It helps in moving from host-centric infrastructure to container-centric infrastructure.**

Main Components of the Kubernetes Master Server

etcd cluster – a distributed key value storage that stores Kubernetes cluster data
kube-apiserver – the central management entity that receives all REST requests for modifications to cluster elements
kube-controller-manager – runs controller processes like replication controller (sets number of replicas in a pod) and endpoints controller (populates services, pods and other objects)
cloud-controller-manager – responsible for managing controller processes with dependencies on the underlying cloud provider

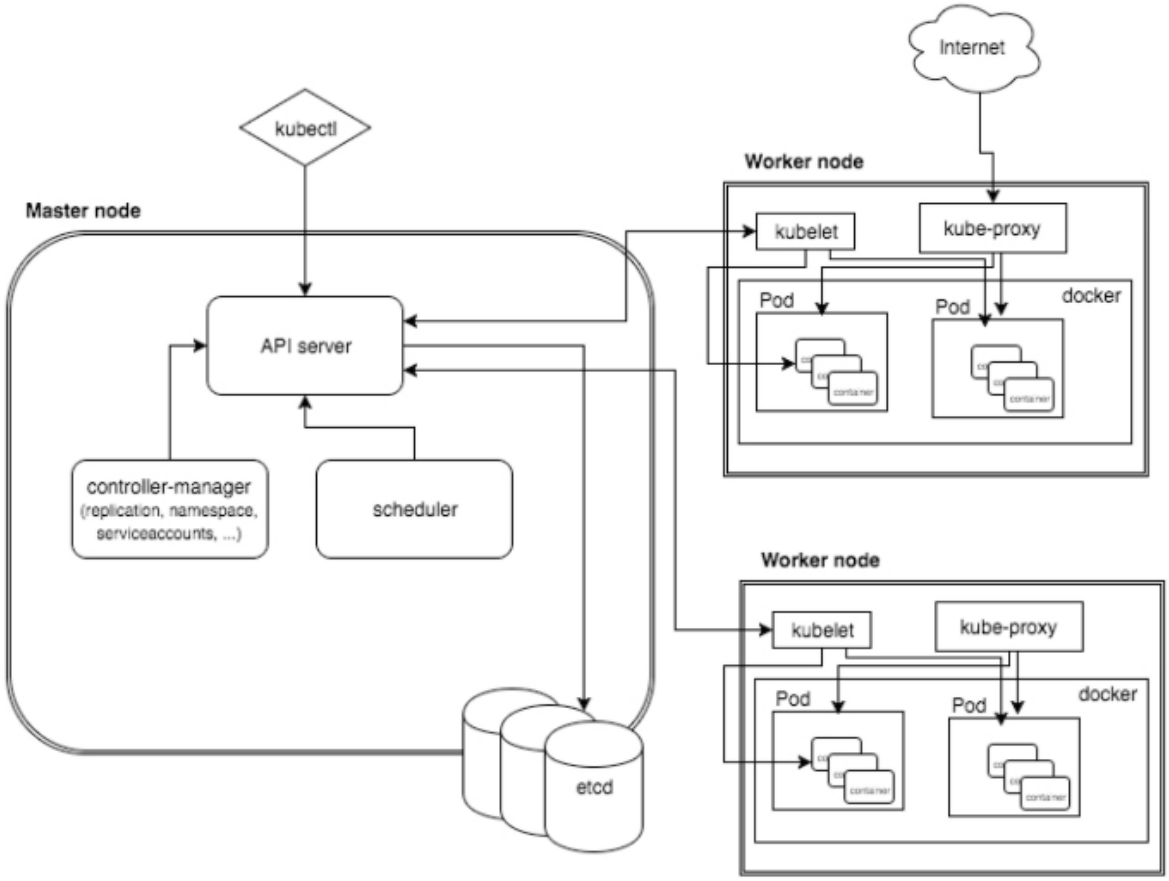
kube-scheduler – helps schedule the pods (a co-located group of containers inside which our application processes are running) on the cluster nodes based on resource utilization

Main components of the Kubernetes Node (Worker) Server

kubelet – the main service on a node, taking in new or modified pod specifications from kube-apiserver, and ensuring that pods and containers are healthy and running
kube-proxy – runs on each worker node to deal with individual host subnetting and expose services

kubectl is a command line tool that interacts with kube-apiserver and send commands to the master node. Each command is converted into an API call.

Kubernetes Architecture



Master Components

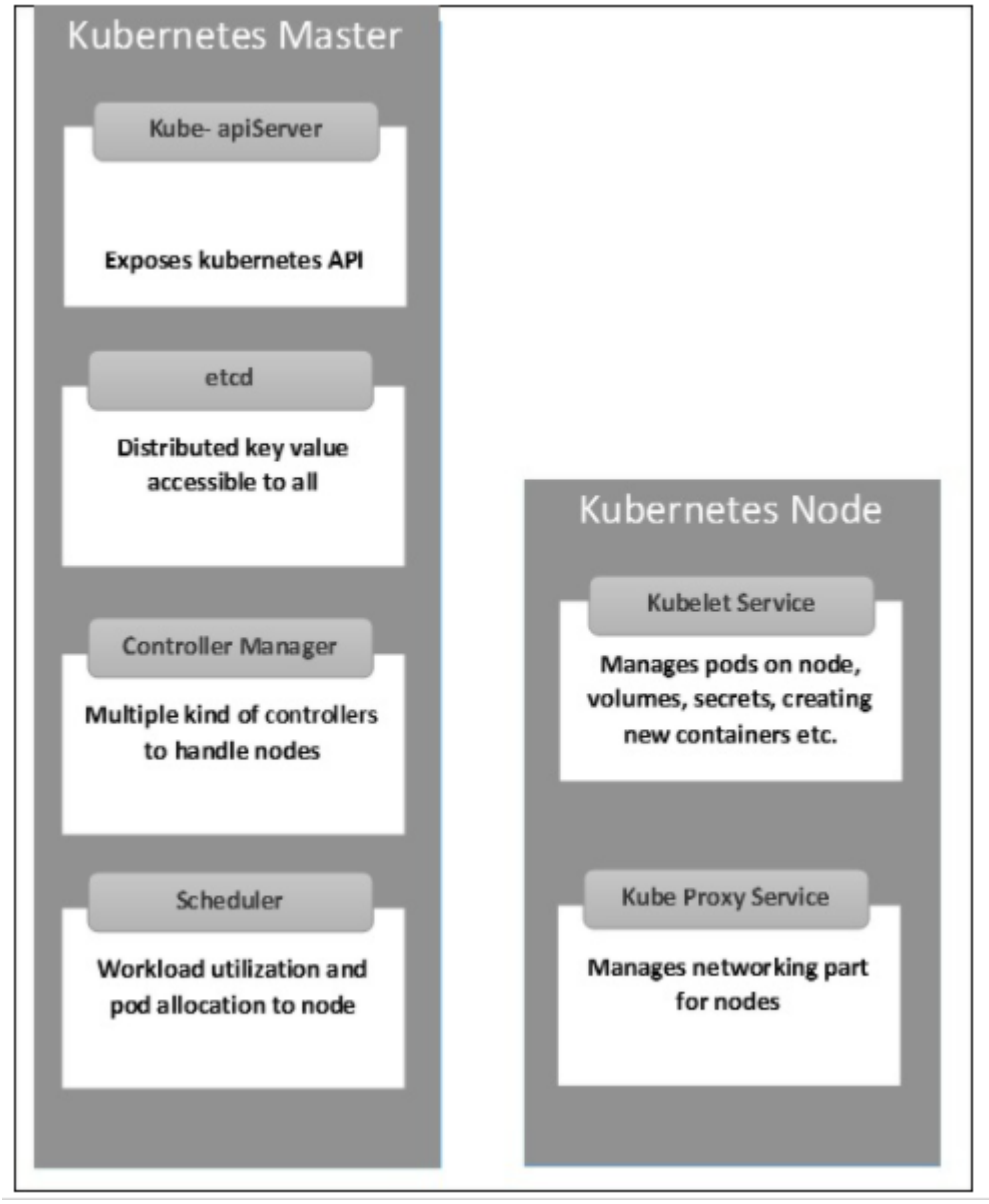
Below are the main components found on the master node:

- **etcd cluster** – a simple, distributed key value storage which is used to store the Kubernetes cluster data (such as number of pods, their state, namespace, etc), API objects and service discovery details. It is only accessible from the API server for security reasons. etcd enables notifications to the cluster about configuration changes with the help of watchers. Notifications are API requests on each etcd cluster node to trigger the update of information in the node's storage.
- **kube-apiserver** – Kubernetes API server is the central management entity that receives all REST requests for modifications (to pods, services, replication sets/controllers and others), serving as frontend to the cluster. Also, this is the only component that communicates with the etcd cluster, making sure data is stored in etcd and is in agreement with the service details of the deployed pods.
- **kube-controller-manager** – runs a number of distinct controller processes in the background (for example, replication controller controls number of replicas in a pod, endpoints controller populates endpoint objects like services and pods, and others) to regulate the shared state of the cluster and perform routine tasks. When a change in a service configuration occurs (for example, replacing the image from which the pods are running, or changing parameters in the configuration yaml file), the controller spots the change and starts working towards the new desired state.
- **cloud-controller-manager** – is responsible for managing controller processes with dependencies on the underlying cloud provider (if applicable). For example, when a controller needs to check if a node was terminated or set up routes, load balancers or volumes in the cloud infrastructure all that is handled by the cloud-controller-manager.
- **kube-scheduler** – helps schedule the pods (a co-located group of containers inside which our application processes are running) on the various nodes based on resource utilization. It reads the service's operational requirements and schedules it on the best fit node. For example, if the application needs 1GB of memory and 2 CPU cores, then the pods for that application will be scheduled on a node with at least those resources. The scheduler runs each time there is a need to schedule pods. The scheduler must know the total resources available as well as resources allocated to existing workloads on each node.

Node (worker) components

Below are the main components found on a (worker) node:

- **kubelet** – the main service on a node, regularly taking in new or modified pod specifications (primarily through the kube-apiserver) and ensuring that pods and their containers are healthy and running in the desired state. This component also reports to the master on the health of the host where it is running.
- **kube-proxy** – a proxy service that runs on each worker node to deal with individual host subnetting and expose services to the external world. It performs request forwarding to the correct pods/containers across the various isolated networks in a cluster.



Kubernetes Concepts

Making use of Kubernetes requires understanding the different abstractions it uses to represent the state of the system, such as services, pods, volumes, namespaces, and deployments.

- **Pod** – generally refers to one or more containers that should be controlled as a single application. A pod encapsulates application containers, storage resources, a unique network ID and other configuration on how to run the containers.
- **Service** – pods are volatile, that is Kubernetes does not guarantee a given physical pod will be kept alive (for instance, the replication controller might kill and start a new set of pods). Instead,

a service represents a logical set of pods and acts as a gateway, allowing (client) pods to send requests to the service without needing to keep track of which physical pods actually make up the service.

- **Volume** – similar to a container volume in Docker, but a Kubernetes volume applies to a whole pod and is mounted on all containers in the pod. Kubernetes guarantees data is preserved across container restarts. The volume will be removed only when the pod gets destroyed. Also, a pod can have multiple volumes (possibly of different types) associated.
- **Namespace** – a virtual cluster (a single physical cluster can run multiple virtual ones) intended for environments with many users spread across multiple teams or projects, for isolation of concerns. Resources inside a namespace must be unique and cannot access resources in a different namespace. Also, a namespace can be allocated a resource quota to avoid consuming more than its share of the physical cluster's overall resources.
- **Deployment** – describes the desired state of a pod or a replica set, in a yaml file. The deployment controller then gradually updates the environment (for example, creating or deleting replicas) until the current state matches the desired state specified in the deployment file. For example, if the yaml file defines 2 replicas for a pod but only one is currently running, an extra one will get created. Note that replicas managed via a deployment should not be manipulated directly, only via new deployments.

Kubernetes – Images

Kubernetes (Docker) images are the key building blocks of Containerized Infrastructure. As of now, we are only supporting Kubernetes to support Docker images. Each container in a pod has its Docker image running inside it.

When we are configuring a pod, the image property in the configuration file has the same syntax as the Docker command does. The configuration file has a field to define the image name, which we are planning to pull from the registry.

Following is the common configuration structure which will pull image from Docker registry and deploy in to Kubernetes container.

```
apiVersion: v1
kind: pod
metadata:
   name: Tesing_for_Image_pull ----------> 1
   spec:
      containers:
         - name: neo4j-server ------------------------> 2
           image: <Name of the Docker image>----------> 3
           imagePullPolicy: Always ------------->4
           command: ["echo", "SUCCESS"] ------------------->
```

In the above code, we have defined –

- **name: Tesing_for_Image_pull** – This name is given to identify and check what is the name of the container that would get created after pulling the images from Docker registry.
- **name: neo4j-server** – This is the name given to the container that we are trying to create. Like we have given neo4j-server.
- **image: <Name of the Docker image>** – This is the name of the image which we are trying to pull from the Docker or internal registry of images. We need to define a complete registry path along with the image name that we are trying to pull.
- **imagePullPolicy** – Always - This image pull policy defines that whenever we run this file to create the container, it will pull the same name again.
- **command: ["echo", "SUCCESS"]** – With this, when we create the container and if everything goes fine, it will display a message when we will access the container
- In order to pull the image and create a container, we will run the following command.

```
$ kubectl create –f Tesing_for_Image_pull
```

- Once we fetch the log, we will get the output as successful.

```
$ kubectl log Tesing_for_Image_pull
```

- The above command will produce an output of success or we will get an output as failure.

Labels

Labels are key-value pairs which are attached to pods, replication controller and services. They are used as identifying attributes for objects such as pods and replication controller. They can be added to an object at creation time and can be added or modified at the run time.

Selectors

Labels do not provide uniqueness. In general, we can say many objects can carry the same labels. Labels selector are core grouping primitive in Kubernetes. They are used by the users to select a set of objects.

Kubernetes API currently supports two type of selectors –

- Equality-based selectors
- Set-based selectors

Equality-based Selectors

They allow filtering by key and value. Matching objects should satisfy all the specified labels.

Set-based Selectors

Set-based selectors allow filtering of keys according to a set of values.

```
apiVersion: v1
kind: Service
metadata:
   name: sp-neo4j-standalone
spec:
   ports:
      - port: 7474
   name: neo4j
   type: NodePort
   selector:
      app: salesplatform ----------> 1
      component: neo4j ------------> 2
```

In the above code, we are using the label selector as **app: salesplatform** and component as **component: neo4j**.

Once we run the file using the **kubectl** command, it will create a service with the name **sp-neo4j-standalone** which will communicate on port 7474. The ype is **NodePort** with the new label selector as **app: salesplatform** and **component: neo4j**.

Kubernetes - Namespace

Namespace provides an additional qualification to a resource name. This is helpful when multiple teams are using the same cluster and there is a potential of name collision. It can be as a virtual wall between multiple clusters.

Functionality of Namespace

Following are some of the important functionalities of a Namespace in Kubernetes −

- Namespaces help pod-to-pod communication using the same namespace.
- Namespaces are virtual clusters that can sit on top of the same physical cluster.
- They provide logical separation between the teams and their environments.

Create a Namespace

The following command is used to create a namespace.

```
apiVersion: v1
kind: Namespace
metadata
   name: elk
```

Control the Namespace

The following command is used to control the namespace.

```
$ kubectl create –f namespace.yml ---------> 1
$ kubectl get namespace ------------------> 2
$ kubectl get namespace <Namespace name> -------->3
$ kubectl describe namespace <Namespace name> ---->4
$ kubectl delete namespace <Namespace name>
```

In the above code,

- We are using the command to create a namespace.
- This will list all the available namespace.
- This will get a particular namespace whose name is specified in the command.
- This will describe the complete details about the service.
- This will delete a particular namespace present in the cluster.

Using Namespace in Service - Example

Following is an example of a sample file for using namespace in service

```
apiVersion: v1
kind: Service
metadata:
   name: elasticsearch
   namespace: elk
   labels:
      component: elasticsearch
spec:
   type: LoadBalancer
   selector:
      component: elasticsearch
```

```
   ports:
   - name: http
      port: 9200
      protocol: TCP
   - name: transport
      port: 9300
      protocol: TCP
```

In the above code, we are using the same namespace under service metadata with the name of **elk**.

Kubernetes - Node

A node is a working machine in Kubernetes cluster which is also known as a minion. They are working units which can be physical, VM, or a cloud instance.

Each node has all the required configuration required to run a pod on it such as the proxy service and kubelet service along with the Docker, which is used to run the Docker containers on the pod created on the node.

They are not created by Kubernetes but they are created externally either by the cloud service provider or the Kubernetes cluster manager on physical or VM machines.

The key component of Kubernetes to handle multiple nodes is the controller manager, which runs multiple kind of controllers to manage nodes. To manage nodes, Kubernetes creates an object of kind node which will validate that the object which is created is a valid node.

Service with Selector

```
apiVersion: v1
kind: node
metadata:
   name: < ip address of the node>
   labels:
      name: <lable name>
```

Node Controller

They are the collection of services which run in the Kubernetes master and continuously monitor the node in the cluster on the basis of metadata.name. If all the required services are running, then the node is validated and a newly created pod will be assigned to that node by the controller. If it is not valid, then the master will not assign any pod to it and will wait until it becomes valid.

Kubernetes master registers the node automatically, if **–register-node** flag is true.

```
–register-node = true
```

However, if the cluster administrator wants to manage it manually then it could be done by turning the flat of –

```
–register-node = false
```

Kubernetes - API

Kubernetes API serves as a foundation for declarative configuration schema for the system. **Kubectl** command-line tool can be used to create, update, delete, and get API object. Kubernetes API acts a communicator among different components of Kubernetes.

Adding API to Kubernetes

Adding a new API to Kubernetes will add new features to Kubernetes, which will increase the functionality of Kubernetes. However, alongside it will also increase the cost and maintainability of the system. In order to create a balance between the cost and complexity, there are a few sets defined for it.

The API which is getting added should be useful to more than 50% of the users. There is no other way to implement the functionality in Kubernetes. Exceptional circumstances are discussed in the community meeting of Kubernetes, and then API is added.

API Changes

In order to increase the capability of Kubernetes, changes are continuously introduced to the system. It is done by Kubernetes team to add the functionality to Kubernetes without removing or impacting the existing functionality of the system.

To demonstrate the general process, here is an (hypothetical) example –

- A user POSTs a Pod object to **/api/v7beta1/...**
- The JSON is unmarshalled into a **v7beta1.Pod** structure
- Default values are applied to the **v7beta1.Pod**
- The **v7beta1.Pod** is converted to an **api.Pod** structure
- The **api.Pod** is validated, and any errors are returned to the user
- The **api.Pod** is converted to a v6.Pod (because v6 is the latest stable version)

- The **v6.Pod** is marshalled into JSON and written to **etcd**

The implication of this process is that API changes must be done carefully and backward compatible.

API Versioning

To make it easier to support multiple structures, Kubernetes supports multiple API versions each at different API path such as **/api/v1** or **/apsi/extensions/v1beta1** Versioning standards at Kubernetes are defined in multiple standards.

Stable Level

- The version name is **vX** where **X** is an integer.
- Stable versions of features will appear in the released software for many subsequent versions.

Kubernetes – Service

A service can be defined as a logical set of pods. It can be defined as an abstraction on the top of the pod which provides a single IP address and DNS name by which pods can be accessed. With Service, it is very easy to manage load balancing configuration. It helps pods to scale very easily.

A service is a REST object in Kubernetes whose definition can be posted to Kubernetes apiServer on the Kubernetes master to create a new instance.

Service without Selector

```
apiVersion: v1
kind: Service
metadata:
   name: Tutorial_point_service
spec:
   ports:
   - port: 8080
   targetPort: 31999
```

The above configuration will create a service with the name Tutorial_point_service.

Service Config File with Selector

```
apiVersion: v1
kind: Service
metadata:
  name: Tutorial_point_service
spec:
  selector:
    application: "My Application" ------------------> (Selector)
  ports:
  - port: 8080
  targetPort: 31999
```

In this example, we have a selector; so in order to transfer traffic, we need to create an endpoint manually.

```
apiVersion: v1
kind: Endpoints
metadata:
  name: Tutorial_point_service
subnets:
  address:
    "ip": "192.168.168.40" ------------------> (Selector)
  ports:
    - port: 8080
```

In the above code, we have created an endpoint which will route the traffic to the endpoint defined as “192.168.168.40:8080”

Types of Services

ClusterIP – This helps in restricting the service within the cluster. It exposes the service within the defined Kubernetes cluster.

```
spec:
   type: NodePort
   ports:
   - port: 8080
      nodePort: 31999
      name: NodeportService
```

NodePort – It will expose the service on a static port on the deployed node. A **ClusterIP** service, to which **NodePort** service will route, is automatically created. The service can be accessed from outside the cluster using the **NodeIP:nodePort**.

```
spec:
   ports:
   - port: 8080
      nodePort: 31999
      name: NodeportService
      clusterIP: 10.20.30.40
```

Load Balancer – It uses cloud providers' load balancer. NodePort and ClusterIP services are created automatically to which the external load balancer will route.

A full service **yaml** file with service type as Node Port. Try to create one yourself.

```
apiVersion: v1
kind: Service
metadata:
   name: appname
   labels:
      k8s-app: appname
spec:
   type: NodePort
   ports:
   - port: 8080
      nodePort: 31999
      name: omninginx
   selector:
      k8s-app: appname
      component: nginx
      env: env_name
```

Kubernetes - Pod

A pod is a collection of containers and its storage inside a node of a Kubernetes cluster. It is possible to create a pod with multiple containers inside it.

For example, keeping a database container and data container in the same pod.

Types of Pod

There are two types of Pods –

- Single container pod
- Multi container pod

Single Container Pod

They can be simply created with the kubectl run command, where you have a defined image on the Docker registry which we will pull while creating a pod.

```
$ kubectl run <name of pod> --image=<name of the image from registry>
```

Example – We will create a pod with a tomcat image which is available on the Docker hub.

```
$ kubectl run tomcat --image = tomcat:8.0
```

This can also be done by creating the **yaml** file and then running the **kubectl create** command.

```
apiVersion: v1
kind: Pod
metadata:
   name: Tomcat
spec:
   containers:
   - name: Tomcat
     image: tomcat: 8.0
     ports:
      containerPort: 7500
     imagePullPolicy: Always
```

Once the above **yaml** file is created, we will save the file with the name of **tomcat.yml** and run the create command to run the document.

```
$ kubectl create –f tomcat.yml
```

It will create a pod with the name of tomcat. We can use the describe command along with **kubectl** to describe the pod.

Multi Container Pod

Multi container pods are created using **yaml mail** with the definition of the containers.

```
apiVersion: v1
kind: Pod
metadata:
   name: Tomcat
spec:
   containers:
   - name: Tomcat
    image: tomcat: 8.0
    ports:
containerPort: 7500
   imagePullPolicy: Always
   -name: Database
   Image: mongoDB
   Ports:
containerPort: 7501
   imagePullPolicy: Always
```

In the above code, we have created one pod with two containers inside it, one for tomcat and the other for MongoDB.

Kubernetes - Replication Controller

Replication Controller is one of the key features of Kubernetes, which is responsible for managing the pod lifecycle. It is responsible for making sure that the specified number of pod replicas are running at any point of time. It is used in time when one wants to make sure that the specified number of pod or at least one pod is running. It has the capability to bring up or down the specified no of pod.

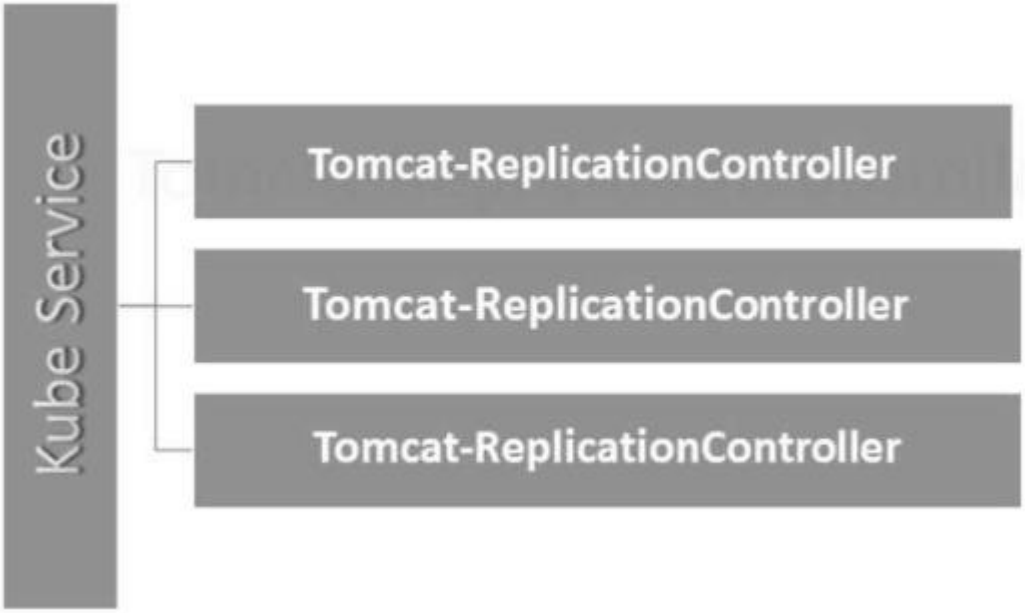
It is a best practice to use the replication controller to manage the pod life cycle rather than creating a pod again and again.

```
apiVersion: v1
kind: ReplicationController ------------------------> 1
metadata:
   name: Tomcat-ReplicationController ----------------------
--> 2
spec:
```

```
replicas: 3 ----------------------> 3
template:
   metadata:
      name: Tomcat-ReplicationController
   labels:
      app: App
      component: neo4j
   spec:
      containers:
      - name: Tomcat- ----------------------> 4
      image: tomcat: 8.0
      ports:
         - containerPort: 7474 ----------------------> 5
```

Setup Details

- **Kind: ReplicationController** → In the above code, we have defined the kind as replication controller which tells the **kubectl** that the **yaml** file is going to be used for creating the replication controller.
- **name: Tomcat-ReplicationController** → This helps in identifying the name with which the replication controller will be created. If we run the kubctl, get **rc < Tomcat-ReplicationController >** it will show the replication controller details.
- **replicas: 3** → This helps the replication controller to understand that it needs to maintain three replicas of a pod at any point of time in the pod lifecycle.
- **name: Tomcat** → In the spec section, we have defined the name as tomcat which will tell the replication controller that the container present inside the pods is tomcat.
- **containerPort: 7474** → It helps in making sure that all the nodes in the cluster where the pod is running the container inside the pod will be exposed on the same port 7474.



Here, the Kubernetes service is working as a load balancer for three tomcat replicas.

Kubernetes - Replica Sets

Replica Set ensures how many replica of pod should be running. It can be considered as a replacement of replication controller. The key difference between the replica set and the replication controller is, the replication controller only supports equality-based selector whereas the replica set supports set-based selector.

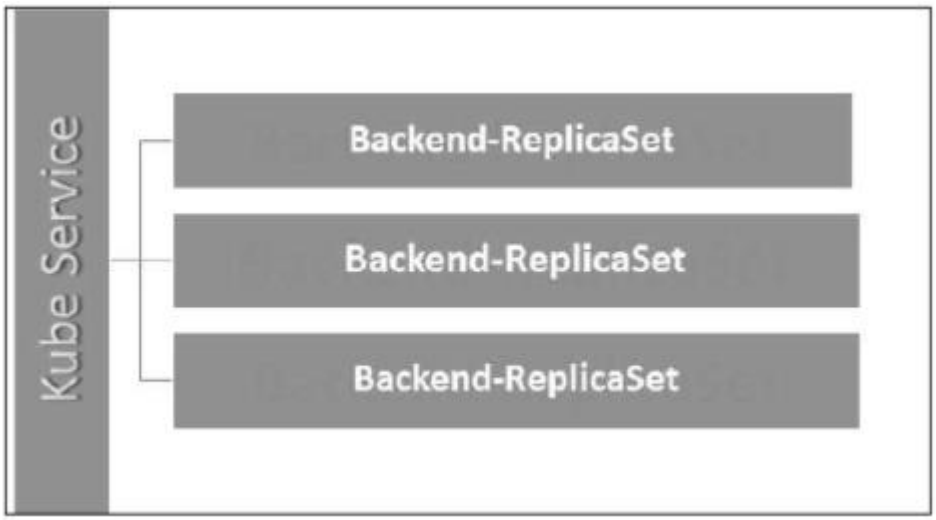
```
apiVersion: extensions/v1beta1 ---------------------->1
kind: ReplicaSet ------------------------> 2
metadata:
   name: Tomcat-ReplicaSet
spec:
   replicas: 3
   selector:
      matchLables:
         tier: Backend ------------------> 3
      matchExpression:
{ key: tier, operation: In, values: [Backend]} -------------->
4
template:
   metadata:
      labels:
         app: Tomcat-ReplicaSet
         tier: Backend
      labels:
```

```
        app: App
        component: neo4j
  spec:
     containers:
     - name: Tomcat
     image: tomcat: 8.0
     ports:
     - containerPort: 7474
```

Setup Details

- **apiVersion: extensions/v1beta1** → In the above code, the API version is the advanced beta version of Kubernetes which supports the concept of replica set.
- **kind: ReplicaSet** → We have defined the kind as the replica set which helps kubectl to understand that the file is used to create a replica set.
- **tier: Backend** → We have defined the label tier as backend which creates a matching selector.
- **{key: tier, operation: In, values: [Backend]}** → This will help **matchExpression** to understand the matching condition we have defined and in the operation which is used by **matchlabel** to find details.

Run the above file using **kubectl** and create the backend replica set with the provided definition in the **yaml** file.



Kubernetes - Deployments

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

They provide many updated features of **matchLabels** and **selectors**. We have got a new controller in the Kubernetes master called the deployment controller which makes it happen. It has the capability to change the deployment midway.

Changing the Deployment

Updating – The user can update the ongoing deployment before it is completed. In this, the existing deployment will be settled and new deployment will be created.

Deleting – The user can pause/cancel the deployment by deleting it before it is completed. Recreating the same deployment will resume it.

Rollback – We can roll back the deployment or the deployment in progress. The user can create or update the deployment by using **DeploymentSpec.PodTemplateSpec = oldRC.PodTemplateSpec.**

Deployment Strategies

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

Recreate – This feature will kill all the existing RC and then bring up the new ones. This results in quick deployment however it will result in downtime when the old pods are down and the new pods have not come up.

Rolling Update – This feature gradually brings down the old RC and brings up the new one. This results in slow deployment, however there is no deployment. At all times, few old pods and few new pods are available in this process.

The configuration file of Deployment looks like this.

```
apiVersion: extensions/v1beta1 ---------------------->1
kind: Deployment ------------------------> 2
metadata:
   name: Tomcat-ReplicaSet
spec:
   replicas: 3
```

```
   template:
      metadata:
         lables:
            app: Tomcat-ReplicaSet
            tier: Backend
   spec:
      containers:
         - name: Tomcatimage:
            tomcat: 8.0
            ports:
               - containerPort: 7474
```

In the above code, the only thing which is different from the replica set is we have defined the kind as deployment.

Create Deployment

```
$ kubectl create -f Deployment.yaml --record
deployment "Deployment" created Successfully.
```

Fetch the Deployment

```
$ kubectl get deployments
NAME           DESIRED     CURRENT     UP-TO-DATE     AVILABLE
AGE
Deployment        3           3           3              3
20s
```

Check the Status of Deployment

```
$ kubectl rollout status deployment/Deployment
```

Updating the Deployment

```
$ kubectl set image deployment/Deployment tomcat=tomcat:6.0
```

Check Rollback revision using replica sets

```
$kubectl get rs
```

Rollback back History

```
$kubectl rollout history deployment/Deployment
```

Rolling Back to Previous Deployment

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

Secrets

Secrets can be defined as Kubernetes objects used to store sensitive data such as user name and passwords with encryption.
a Secret is an object that contains a small amount of sensitive data such as login usernames and passwords, tokens, keys, etc.
The primary purpose of Secrets is to reduce the risk of exposing sensitive data while deploying applications on Kubernetes.

How to Encypt password using base64:

```
echo -n 'nginxadmin' | base64
echo -n 'nginx123' | base64
```

Eg: Sample Secret Yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
type: Opaque
data:
  username: bmdpbnhhZG1pbg==
  password: bmdpbngxMjM=
```

```
kubectl create -f secret. Yaml

kubectl get secrets

kubectl describe secret my-secret

kubectl edit secret my-secret
```

How to use secrets inside Pod or deployment:

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
```

```
  - name: nginx-container
    image: nginx:latest
    env:
      - name: USER
        valueFrom:
          secretKeyRef:
            name: my-secret
            key: username
      - name: PASSWORD
        valueFrom:
          secretKeyRef:
            name: my-secret
            key: password
```

```
kubectl create -f pod.yml

kubectl logs -f mypod

 kubectl exec -it mypod bash

Test the environment variable using the echo command:

echo $USER $PASSWORD
```

How to decode the base64 encoded passwords:

```
echo 'bmdpbnhhZG1pbg==' | base64 --decode
echo 'bmdpbngxMjM=' | base64 --decode
```

Volumes

hostPath volume type is a durable volume type that mounts a directory from the host Node's filesystem into a Pod.
The file in the volume remains intact even if the Pod crashes, is terminated or is deleted.
It is important that the directory and the Pod are created or scheduled on the same Node.

Syntax for Host volumes:

```
volumes:
- name: vol_name         # The name of the volume
  hostPath:
```

```
  path: /data          # directory location on host
```

A volumeMounts, on the other hand, entails mounting of the declared volume into a container in the same Pod.

Syntax for Pod Volumes:

```
spec:
 containers:
 - name: my-app
   image: nginx
   volumeMounts:
   - name: vol_name
     mountPath: /app/config
```

How to use volumes in Pod and Deployment:

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp
spec:
  containers:
  - name: my-app
    image: nginx
    ports:
    - containerPort: 8080
    volumeMounts:
    - name: my-volume
      mountPath: /app
  volumes:
  - name: my-volume
    hostPath:
      path: /mnt/vpath
```

Kubernetes Installation

Step 1: Install and Configure Docker on Master and Worker Nodes.

Prerequisite:
Operating system: Amazon AMI
Instance Type: t2. medium (2 core CPU and 4GB RAM)

Step a: Install docker package using yum command

```
yum install docker -y
```

Step b: Start docker Service and check status

```
systemctl start docker
```

```
systemctl status docker
```

```
systemctl enable docker
```

Step 2: Add Kubernetes repo to both Master and Worker Nodes.

```
cat <<EOF | sudo tee /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-\$basearch
enabled=1
gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
exclude=kubelet kubeadm kubectl
EOF
```

Step 3: Turn of the swap space and check if selinux in disabled on master and worker

```
swapoff --all

sestatus
```

Step 4: Install kubelet, kubeadm , kubectl & start and enable service on both master and worker

```
yum install -y kubelet kubeadm kubectl --disableexcludes=kubernetes


systemctl enable kubelet && systemctl start kubelet
```

Step 5: Initialize kubeadm only on Master Node

```
kubeadm init
```

Now Run the command below only on master node, same commands will be display above output from kubeadm init command. Preserve the output in Notepad for join more worker nodes in future.

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

Step 6: Run Kubeadm Join command from output of kubeadm init only on Worker Nodes

```
<kubeadm join command copies from master node>
```

Step 7: Set the path using export command.

```
Master Node:

export KUBECONFIG=/etc/kubernetes/admin.conf

Worker Node:

export KUBECONFIG=/etc/kubernetes/kubelet.conf
```

Step 8: Run kubectl get nodes to check system state which is not ready.

```
kubectl get nodes
```

```
[root@ip-172-31-4-94 bitnami]# kubectl get nodes
NAME                                          STATUS     ROLES                  AGE     VERSION
ip-172-31-2-221.us-west-2.compute.internal    NotReady   control-plane,master   4m53s   v1.21.0
ip-172-31-4-94.us-west-2.compute.internal     NotReady   <none>                 47s     v1.21.0
```

Step 9: Configure the networking using flannel yamls for bringing states of nodes to ready on Master Node

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

kubectl apply -f calico.yaml
```

Step10: Check the kubectl commands to see if all services are running.

```
kubectl get nodes

kubectl get pods --all-namespaces
```

==========XXXXXXX==========

Realtime Scenario for Application Deployment, Upgrade and Rollback:

Application deployment using deployment and Service ymls:

Step 1: Create Deployment yaml to workspace.

vi deployment.yml

```
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  name: my-app-deploy
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-app
        image: docker.io/goudsagar/my-app:0.0.1
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 8080
```

Step 2: Run the below command for running the yaml file.

Kubectl apply –f deployment.yml

```
[root@ip-172-31-37-238 bitnami]# kubectl apply -f deployment.yml
deployment.apps/my-app-deploy created
```

Step 3: Check deployment status

kubectl get deployment
kubectl get deployment –o wide

```
[root@ip-172-31-37-238 bitnami]# kubectl get deployment
NAME            READY   UP-TO-DATE   AVAILABLE   AGE
my-app-deploy   1/1     1            1           8s
```

```
[root@ip-172-31-37-238 bitnami]# kubectl get deployment -o wide
NAME            READY   UP-TO-DATE   AVAILABLE   AGE   CONTAINERS   IMAGES                              SELECTOR
my-app-deploy   1/1     1            1           50m   my-app       docker.io/goudsagar/my-app:0.0.1    app=my-app
[root@ip-172-31-37-238 bitnami]#
```

Step 4: Create service yml file for accessing and exposing the application URL.

vi service.yml

```
apiVersion: v1
kind: Service
metadata:
  name: my-app-service
spec:
  ports:
    - name: http
      protocol: TCP
      port: 80
      targetPort: 8080
  externalIPs:
  - 3.143.5.188
  selector:
    app: my-app
  type: LoadBalancer
```

```
kubectl apply –f service.yml
```

```
[root@ip-172-31-37-238 bitnami]# kubectl apply -f service.yml
service/my-app-service created
```

Step 5: Check the service status

```
kubectl get svc
kubectl get svc –o wide
```

```
[root@ip-172-31-37-238 bitnami]# kubectl get svc
NAME             TYPE           CLUSTER-IP      EXTERNAL-IP     PORT(S)          AGE
kubernetes       ClusterIP      10.96.0.1       <none>          443/TCP          38m
my-app-service   LoadBalancer   10.99.168.184   54.244.152.59   80:31501/TCP     8s
```

```
[root@ip-172-31-37-238 bitnami]# kubectl get svc -o wide
NAME             TYPE           CLUSTER-IP      EXTERNAL-IP     PORT(S)          AGE   SELECTOR
kubernetes       ClusterIP      10.96.0.1       <none>          443/TCP          87m   <none>
my-app-service   LoadBalancer   10.99.168.184   54.244.152.59   80:31501/TCP     49m   app=my-app
[root@ip-172-31-37-238 bitnami]#
```

Step 6: Check the pods status

```
kubectl get pods
kubectl get pods –o wide
```

```
[root@ip-172-31-37-238 bitnami]# kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
my-app-deploy-56c567bf4c-m7wl2      1/1     Running   0          2m55s
```

```
[root@ip-172-31-37-238 bitnami]# kubectl get pods -o wide
NAME                                READY   STATUS    RESTARTS   AGE   IP              NODE                                              NOMINATED NODE   READINESS GATES
my-app-deploy-56c567bf4c-sjfnv      1/1     Running   0          25m   192.168.51.138  ip-172-31-43-213.us-west-2.compute.internal       <none>           <none>
[root@ip-172-31-37-238 bitnami]#
```

Step 7: Checking deployment/pod/service logs

```
kubectl logs –f pod/deployment/service
```

```
[root@ip-172-31-37-238 bitnami]# kubectl logs -f my-app-deploy-56c567bf4c-tcznp
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.4.3)
2021-05-09 18:09:53.322  INFO 1 --- [           main] c.e.restservice.RestServiceApplication   : Starting RestServiceApplication v0.0.1-SNAPSHOT using Java 1.8.0_212 on my-app-deploy-56c567bf
4c-tcznp with PID 1 (/my-app.jar started by root in /)
2021-05-09 18:09:53.336  INFO 1 --- [           main] c.e.restservice.RestServiceApplication   : No active profile set, falling back to default profiles: default
2021-05-09 18:09:56.036  INFO 1 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
2021-05-09 18:09:56.085  INFO 1 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2021-05-09 18:09:56.086  INFO 1 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.43]
2021-05-09 18:09:56.235  INFO 1 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2021-05-09 18:09:56.235  INFO 1 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 2723 ms
2021-05-09 18:09:57.444  INFO 1 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor  : Initializing ExecutorService 'applicationTaskExecutor'
2021-05-09 18:09:57.941  INFO 1 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
2021-05-09 18:09:57.984  INFO 1 --- [           main] c.e.restservice.RestServiceApplication   : Started RestServiceApplication in 6.265 seconds (JVM running for 8.136)
2021-05-09 18:10:38.680  INFO 1 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet 'dispatcherServlet'
2021-05-09 18:10:38.680  INFO 1 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
```

Step 8: Checking the details of deployments

```
kubectl describe deployments
kubectl describe pod pod_name
```

```
[root@ip-172-31-37-238 bitnami]# kubectl describe deployments
Name:                   my-app-deploy
Namespace:              default
CreationTimestamp:      Sun, 09 May 2021 17:10:37 +0000
Labels:                 <none>
Annotations:            deployment.kubernetes.io/revision: 5
Selector:               app=my-app
Replicas:               3 desired | 3 updated | 3 total | 3 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=my-app
  Containers:
   my-app:
    Image:        docker.io/goudsagar/my-app:0.0.1
    Port:         8080/TCP
    Host Port:    0/TCP
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Progressing    True    NewReplicaSetAvailable
  Available      True    MinimumReplicasAvailable
OldReplicaSets:  <none>
NewReplicaSet:   my-app-deploy-56c567bf4c (3/3 replicas created)
```

```
Events:
  Type    Reason             Age                From                   Message
  ----    ------             ----               ----                   -------
  Normal  ScalingReplicaSet  47m                deployment-controller  Scaled up replica set my-app-deploy-d9b657b8 to 2
  Normal  ScalingReplicaSet  47m                deployment-controller  Scaled up replica set my-app-deploy-d9b657b8 to 3
  Normal  ScalingReplicaSet  45m                deployment-controller  Scaled up replica set my-app-deploy-56c567bf4c to 2
  Normal  ScalingReplicaSet  45m                deployment-controller  Scaled down replica set my-app-deploy-d9b657b8 to 2
  Normal  ScalingReplicaSet  45m                deployment-controller  Scaled down replica set my-app-deploy-d9b657b8 to 1
  Normal  ScalingReplicaSet  20m                deployment-controller  Scaled down replica set my-app-deploy-56c567bf4c to 1
  Normal  ScalingReplicaSet  13m (x2 over 57m)   deployment-controller  Scaled up replica set my-app-deploy-d9b657b8 to 1
  Normal  ScalingReplicaSet  13m (x2 over 57m)   deployment-controller  Scaled down replica set my-app-deploy-56c567bf4c to 0
  Normal  ScalingReplicaSet  10m (x3 over 69m)   deployment-controller  Scaled up replica set my-app-deploy-56c567bf4c to 1
  Normal  ScalingReplicaSet  10m (x2 over 45m)   deployment-controller  Scaled down replica set my-app-deploy-d9b657b8 to 0
  Normal  ScalingReplicaSet  6m54s (x2 over 45m)  deployment-controller  Scaled up replica set my-app-deploy-56c567bf4c to 3
[root@ip-172-31-37-238 bitnami]# ||
```

```
[root@ip-172-31-37-238 bitnami]# kubectl describe pod my-app-deploy-56c567bf4c-tcznp
Name:         my-app-deploy-56c567bf4c-tcznp
Namespace:    default
Priority:     0
Node:         ip-172-31-43-213.us-west-2.compute.internal/172.31.43.213
Start Time:   Sun, 09 May 2021 18:09:48 +0000
Labels:       app=my-app
              pod-template-hash=56c567bf4c
Annotations:  cni.projectcalico.org/podIP: 192.168.51.142/32
              cni.projectcalico.org/podIPs: 192.168.51.142/32
Status:       Running
IP:           192.168.51.142
IPs:
  IP:           192.168.51.142
Controlled By:  ReplicaSet/my-app-deploy-56c567bf4c
Containers:
  my-app:
    Container ID:   docker://58252c8a04915636eeabcb2694d9b453cbaabd0a8feb2a3e1236b36b7b016117
    Image:          docker.io/goudsagar/my-app:0.0.1
    Image ID:       docker-pullable://docker.io/goudsagar/my-app@sha256:ace3d8c6f1c1b061128aba35e20bde519d4f6131e8d766671a8af4ed7e4b72b5
    Port:           8080/TCP
    Host Port:      0/TCP
    State:          Running
      Started:      Sun, 09 May 2021 18:09:49 +0000
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-wkqgz (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  kube-api-access-wkqgz:
    Type:                    Projected (a volume that contains injected data from multiple sources)
```

```
    Type:                    Projected (a volume that contains injected data from multiple sources)
    TokenExpirationSeconds:  3607
    ConfigMapName:           kube-root-ca.crt
    ConfigMapOptional:       <nil>
    DownwardAPI:             true
QoS Class:                   BestEffort
Node-Selectors:              <none>
Tolerations:                 node.kubernetes.io/not-ready:NoExecute op=Exists for 300s
                             node.kubernetes.io/unreachable:NoExecute op=Exists for 300s
Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  13m   default-scheduler  Successfully assigned default/my-app-deploy-56c567bf4c-tcznp to ip-172-31-43-213.us-west-2.compute.internal
  Normal  Pulled     13m   kubelet            Container image "docker.io/goudsagar/my-app:0.0.1" already present on machine
  Normal  Created    13m   kubelet            Created container my-app
  Normal  Started    13m   kubelet            Started container my-app
[root@ip-172-31-37-238 bitnami]#
```

Step 9: Checking service status and describe service

kubectl get svc
kubectl describe svc service-name

```
[root@ip-172-31-37-238 bitnami]# kubectl get svc
NAME              TYPE           CLUSTER-IP      EXTERNAL-IP     PORT(S)          AGE
kubernetes        ClusterIP      10.96.0.1       <none>          443/TCP          111m
my-app-service    LoadBalancer   10.99.168.184   54.244.152.59   80:31501/TCP     73m
[root@ip-172-31-37-238 bitnami]# kubectl describe svc my-app-service
Name:                     my-app-service
Namespace:                default
Labels:                   <none>
Annotations:              <none>
Selector:                 app=my-app
Type:                     LoadBalancer
IP Family Policy:         SingleStack
IP Families:              IPv4
IP:                       10.99.168.184
IPs:                      10.99.168.184
External IPs:             54.244.152.59
Port:                     http  80/TCP
TargetPort:               8080/TCP
NodePort:                 http  31501/TCP
Endpoints:                192.168.51.142:8080,192.168.51.143:8080,192.168.51.144:8080
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
[root@ip-172-31-37-238 bitnami]#
```

Step 10: Access URL in Browser

Rolling Strategy in Kubernetes

Step1: Run "kubectl set image" command for deployment the latest/new feature image replacing the old one.

```
kubectl set image deployment/my-app-deploy my-app=docker.io/goudsagar/webapp-application-tomcat:latest
```

```
[root@ip-172-31-37-238 bitnami]# kubectl set image deployment/my-app-deploy my-app=docker.io/goudsagar/webapp-application-tomcat:latest
deployment.apps/my-app-deploy image updated
```

Step 2: Check the rollout status

```
kubectl rollout status deployment/my-app-deploy
```

```
[root@ip-172-31-37-238 bitnami]# kubectl rollout status deployment/my-app-deploy
deployment "my-app-deploy" successfully rolled out
```

Step 3: Check the rollout history

```
[root@ip-172-31-37-238 bitnami]# kubectl rollout history deployment/my-app-deploy
deployment.apps/my-app-deploy
REVISION  CHANGE-CAUSE
2         <none>
3         <none>

[root@ip-172-31-37-238 bitnami]#
```

Step 4: Check the replica sets

```
[root@ip-172-31-37-238 bitnami]# kubectl get rs
NAME                       DESIRED   CURRENT   READY   AGE
my-app-deploy-56c567bf4c   1         1         1       52m
my-app-deploy-d9b657b8     0         0         0       40m
[root@ip-172-31-37-238 bitnami]#
```

Step 5: Check the Application from Browser

Step 6: Undo the previous deployment

kubectl rollout undo deployment/my-app-deploy –to-revision=3

```
[root@ip-172-31-37-238 bitnami]# kubectl rollout undo deployment/my-app-deploy --to-revision=3
deployment.apps/my-app-deploy rolled back
[root@ip-172-31-37-238 bitnami]#
```

Step 7: check deployment, service and pod status

```
[root@ip-172-31-37-238 bitnami]# kubectl get deployment
NAME            READY   UP-TO-DATE   AVAILABLE   AGE
my-app-deploy   1/1     1            1           62m
[root@ip-172-31-37-238 bitnami]# kubectl get pods
NAME                             READY   STATUS    RESTARTS   AGE
my-app-deploy-56c567bf4c-tcznp   1/1     Running   0          2m56s
[root@ip-172-31-37-238 bitnami]# kubectl get svc
NAME             TYPE           CLUSTER-IP      EXTERNAL-IP     PORT(S)        AGE
kubernetes       ClusterIP      10.96.0.1       <none>          443/TCP        100m
my-app-service   LoadBalancer   10.99.168.184   54.244.152.59   80:31501/TCP   61m
[root@ip-172-31-37-238 bitnami]#
```

Step 8: Check the application from browser

Replica Sets:

Scaling up or down replica sets

```
kubectl scale –replicas=3 deployment/my-app-deploy
```

```
[root@ip-172-31-37-238 bitnami]# kubectl scale --replicas=3 deployment/my-app-deploy
deployment.apps/my-app-deploy scaled
[root@ip-172-31-37-238 bitnami]#
```

Check status of deployment/pod/service/replica sets using below commands

```
kubectl get deployment

kubectl get pods

kubectl get svc

kubectl get rs
```

```
[root@ip-172-31-37-238 bitnami]# kubectl get deployment
NAME            READY   UP-TO-DATE   AVAILABLE   AGE
my-app-deploy   3/3     3            3           62m
[root@ip-172-31-37-238 bitnami]# kubectl get pods
NAME                              READY   STATUS    RESTARTS   AGE
my-app-deploy-56c567bf4c-tcznp    1/1     Running   0          3m36s
my-app-deploy-56c567bf4c-x7v2t    1/1     Running   0          11s
my-app-deploy-56c567bf4c-zzqpf    1/1     Running   0          11s
[root@ip-172-31-37-238 bitnami]# kubectl get svc
NAME             TYPE           CLUSTER-IP      EXTERNAL-IP     PORT(S)        AGE
kubernetes       ClusterIP      10.96.0.1       <none>          443/TCP        100m
my-app-service   LoadBalancer   10.99.168.184   54.244.152.59   80:31501/TCP   62m
[root@ip-172-31-37-238 bitnami]#
[root@ip-172-31-37-238 bitnami]# kubectl get rs
NAME                       DESIRED   CURRENT   READY   AGE
my-app-deploy-56c567bf4c   3         3         3       62m
my-app-deploy-d9b657b8     0         0         0       51m
[root@ip-172-31-37-238 bitnami]#
```