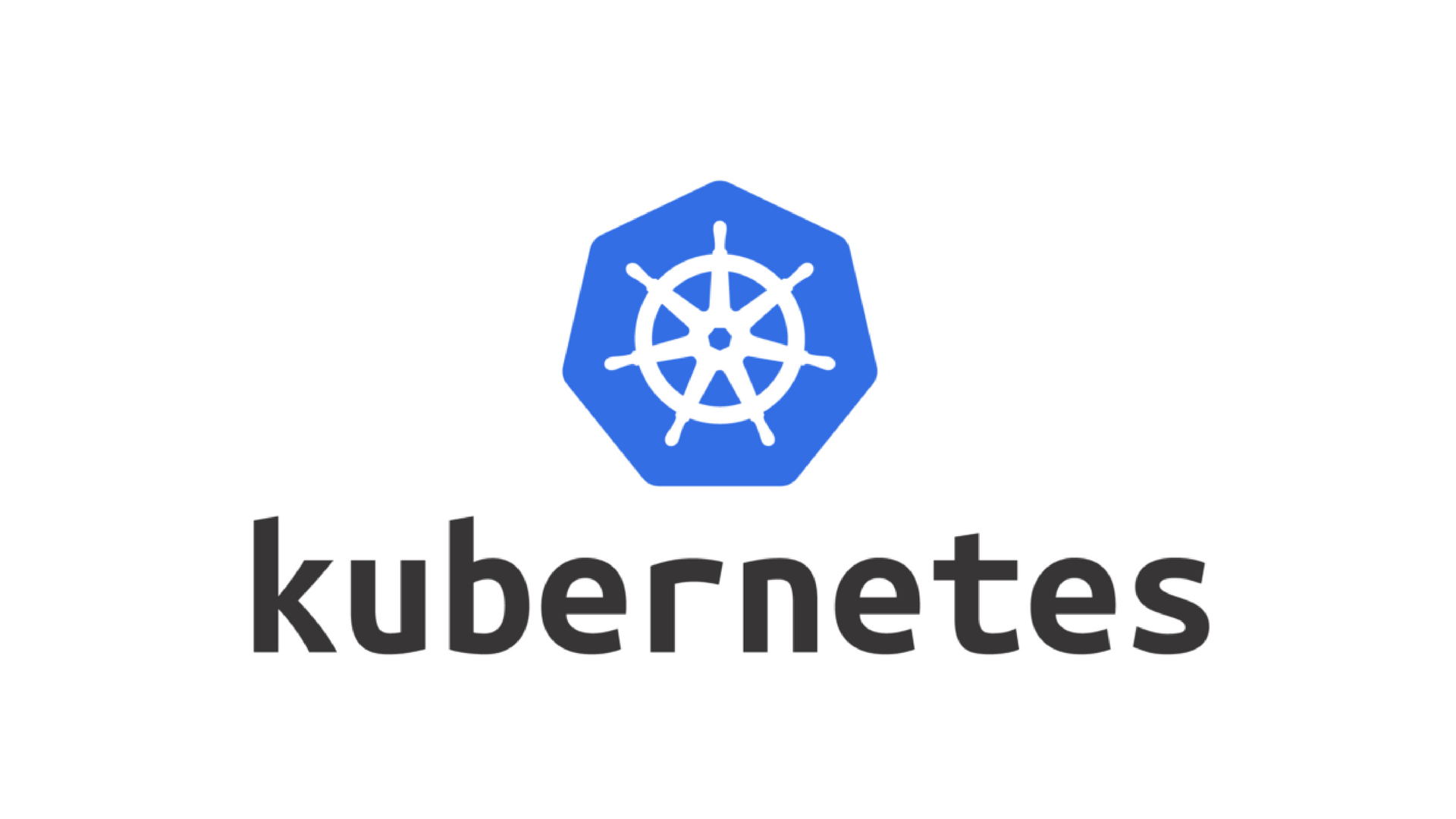
|  |
| --- |
| Kubernetes  RAJESH MUMMIDI 19/5/2023 |
|  |



1. Explain Kubernetes Architecture and components in detail?

A picture containing text, screenshot, number, font

Description automatically generated

**Kubernetes Architecture**

Kubernetes deployment is called a cluster. A Kubernetes cluster consists of at least one main (control) plane, and one or more worker machines, called [nodes](https://kubernetes.io/docs/concepts/architecture/nodes/). Both the control planes and node instances can be physical devices, virtual machines, or instances in the cloud.

**Control Plane or Master Node**

* Also known as master node or head node.
* The control plane manages the worker nodes and the Pods in the cluster.
* In production environments, the control plane usually runs across multiple computers and a cluster usually runs multiple nodes, providing fault-tolerance and high availability.
* The control plane receives input from a CLI or UI via an API.
* It is not recommended to run user workloads on master mode.

**Node(s) or Worker node**

* Also known as worker node or compute node.
* A virtual or physical machine that contains the services necessary to run containerized applications.
* A Kubernetes cluster needs at least one worker node, but normally have many.
* The worker node(s) host the Pods that are the components of the application workload.
* Pods are scheduled and orchestrated to run on nodes.
* You can scale up and scale down cluster by adding and removing nodes.

**Kubernetes Control Plane Components**

The control plane’s components make global decisions about the cluster, as well as detecting and responding to cluster events. Kubernetes relies on several administrative services running on the control plane. These services manage aspects, such as cluster component communication, workload scheduling, and cluster state persistence.

**API Server (kube-apiserver)**

* API server exposes the Kubernetes API.
* Entry point for REST/kubectl — It is the front end for the Kubernetes control plane.
* It tracks the state of all cluster components and managing the interaction between them.
* It is designed to scale horizontally.
* It consumes YAML/JSON manifest files.
* It validates and processes the requests made via API.

**etcd (key-value store)**

* It is a consistent, distributed, and highly-available key value store.
* It is stateful, persistent storage that stores all of Kubernetes cluster data (cluster state and config).
* It is the source of truth for the cluster.
* It can be part of the control plane, or, it can be configured externally.

**Scheduler (kube-scheduler)**

* It schedules pods to worker nodes.
* It watches api-server for newly created Pods with no assigned node, and selects a healthy node for them to run on.
* If there are no suitable nodes, the pods are put in a pending state until such a healthy node appears.
* It watches API Server for new work tasks.

Factors taken into account for scheduling decisions include:

* Individual and collective resource requirements.
* Hardware/software/policy constraints.
* Affinity and anti-affinity specifications.
* Data locality.
* Inter-workload interference.
* Deadlines and taints.

**Controller Manager (kube-controller-manager)**

* It watches the desired state of the objects it manages and watches their current state through the API server.
* It takes corrective steps to make sure that the current state is the same as the desired state.
* It is controller of controllers.
* It runs controller processes. Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process.

Some types of controllers are:

* Node controller: Responsible for noticing and responding when nodes go down.
* Job controller: Watches for Job objects that represent one-off tasks, then creates Pods to run those tasks to completion.
* Endpoints controller: Populates the Endpoints object (that is, joins Services & Pods).
* Service Account & Token controllers: Create default accounts and API access tokens for new namespaces.

**Cloud Controller Manager**

* The cloud controller manager integrates with the underlying cloud technologies in your cluster when the cluster is running in a cloud environment.
* The cloud-controller-manager only runs controllers that are specific to your cloud provider.
* Cloud controller lets you link your cluster into cloud provider’s API, and separates out the components that interact with that cloud platform from components that only interact with your cluster.

The following controllers can have cloud provider dependencies:

* Node controller: For checking the cloud provider to determine if a node has been deleted in the cloud after it stops responding.
* Route controller: For setting up routes in the underlying cloud infrastructure.
* Service controller: For creating, updating and deleting cloud provider load balancers.

**Kubernetes Node Components**

Node components run on every node, maintaining running pods and providing the Kubernetes runtime environment.

**kubelet**

* It is an agent that runs on each node in the cluster.
* It acts as a conduit between the API server and the node.
* It makes sure that containers are running in a Pod and they are healthy.
* It instantiates and executes Pods.
* It watches API Server for work tasks.
* It gets instructions from master and reports back to Masters.

**kube-proxy**

* It is networking component that plays vital role in networking.
* It manages IP translation and routing.
* It is a network proxy that runs on each node in cluster.
* It maintains network rules on nodes. These network rules allow network communication to Pods from inside or outside of cluster.
* It ensure each Pod gets unique IP address.
* It makes possible that all containers in a pod share a single IP.
* It facilitating Kubernetes networking services and load-balancing across all pods in a service.
* It deals with individual host sub-netting and ensure that the services are available to external parties.

**Container runtime**

* The container runtime is the software that is responsible for running containers (in Pods).
* To run the containers, each worker node has a container runtime engine.
* It pulls images from a container image registry and starts and stops containers.

Kubernetes supports several container runtimes:

1. [Docker](https://docs.docker.com/engine/)
2. [containerd](https://containerd.io/docs/)
3. [CRI-O](https://cri-o.io/#what-is-cri-o)
4. Any implementation of the Kubernetes CRI (Container Runtime Interface).

2)Types of cluster creation ? kubeadm, kops(terraform), Mini kube, kube spray (ansible)?

A Kubernetes cluster is a set of nodes that run containerized applications. Containerizing applications packages an app with its dependences and some necessary services. They are more lightweight and flexible than virtual machines. In this way, Kubernetes clusters allow for applications to be more easily developed, moved and managed.

Kubernetes clusters allow containers to run across multiple machines and environments: virtual, physical, cloud-based, and on-premises. Kubernetes containers are not restricted to a specific operating system, unlike virtual machines. Instead, they are able to share operating systems and run anywhere.

Kubernetes clusters are comprised of one master node and a number of worker nodes. These nodes can either be physical computers or virtual machines, depending on the cluster.

The master node controls the state of the cluster; for example, which applications are running and their corresponding container images. The master node is the origin for all task assignments. It coordinates processes such as:

* Scheduling and scaling applications
* Maintaining a cluster’s state
* Implementing updates

The worker nodes are the components that run these applications. Worker nodes perform tasks assigned by the master node. They can either be virtual machines or physical computers, all operating as part of one system.

There must be a minimum of one master node and one worker node for a Kubernetes cluster to be operational. For production and staging, the cluster is distributed across multiple worker nodes. For testing, the components can all run on the same physical or virtual node.

Here are the types of cluster creation methods and the steps to create a cluster using Kubeadm, Kops, Minikube, and Kubespray:

1. Kubeadm:
   * Install Docker: Install Docker on the nodes that will be part of the cluster.
   * Install Kubeadm, Kubelet, and Kubectl: Install the necessary Kubernetes components on all nodes.
   * Initialize the Control Plane: On the master node, run the **kubeadm init** command to initialize the control plane.
   * Join Worker Nodes: On each worker node, run the **kubeadm join** command with the appropriate parameters to join the cluster.
   * Configure Kubectl: Copy the kubeconfig file from the master node to your local machine and set the **KUBECONFIG** environment variable.
   * Deploy a Pod Network: Install a network plugin like Calico, Flannel, or Weave to enable pod networking.
   * Verify Cluster: Run **kubectl get nodes** to verify that all nodes are in the "Ready" state.
2. Kops:
   * Install Kops: Install Kops on your local machine.
   * Create an S3 Bucket: Create an S3 bucket to store the cluster configuration.
   * Create Cluster Configuration: Use Kops to create the cluster configuration file.
   * Create Cluster: Run **kops create cluster** command with the cluster configuration file to create the cluster.
   * Update DNS: Update the DNS records to point to the newly created cluster.
   * Validate Cluster: Run **kops validate cluster** to verify that the cluster is running correctly.
3. Minikube:
   * Install Minikube: Install Minikube on your local machine.
   * Start Minikube: Run the **minikube start** command to start the local cluster.
   * Verify Cluster: Run **kubectl get nodes** to verify that the Minikube cluster is running.
   * Interact with Cluster: Use **kubectl** commands to deploy and manage applications on the Minikube cluster.
4. Kubespray:
   * Install Ansible: Install Ansible on your local machine.
   * Clone Kubespray: Clone the Kubespray repository to your local machine.
   * Configure Inventory: Edit the inventory file to define the nodes and their roles in the cluster.
   * Customize Configuration: Modify the configuration files according to your requirements.
   * Run Playbooks: Run the Ansible playbooks provided by Kubespray to deploy the cluster.
   * Verify Cluster: Use **kubectl** commands to verify the cluster and its components.

Top of Form

3) What is Pods? Check pod.yaml will have?

A pod is the smallest execution unit in Kubernetes. A pod encapsulates one or more applications. Pods are ephemeral by nature, if a pod (or the node it executes on) fails, Kubernetes can automatically create a new replica of that pod to continue operations. Pods include one or more [containers](https://www.vmware.com/topics/glossary/content/containers.html) (such as Docker containers).

Pods also provide environmental dependencies, including persistent storage volumes (storage that is permanent and available to all pods in the cluster) and configuration data needed to run the container(s) within the pod.

PODs Kubernetes Manifest file ( Yaml File)

**apiVersion**: v1

**kind**: Pod

**metadata**:

**name**: nginx

**spec**:

**containers**:

- **name**: nginx

**image**: nginx:1.14.2

**ports**:

- **containerPort**: 80

4) What are the objects you have used so far ?

Kubernetes Objects are persistent entities in the cluster. These objects are used to represent the state of the cluster.

The following are some of the Kubernetes Objects:

* pods
* Namespaces
* Replication Controller (Manages Pods)
* Deployment Controller (Manages Pods)
* Stateful Sets
* Daemon Sets
* Services
* Config Maps
* Volumes

5) What is a Kubernetes Namespace?

Namespaces are a way to organize clusters into virtual sub-clusters — they can be helpful when different teams or projects share a Kubernetes cluster. Any number of namespaces are supported within a cluster, each logically separated from others but with the ability to communicate with each other. Namespaces cannot be nested within each other.

Any resource that exists within [Kubernetes](https://tanzu.vmware.com/kubernetes-vs-docker" \t "_blank) exists either in the default namespace or a namespace that is created by the cluster operator. Only nodes and persistent storage volumes exist outside of the namespace; these low-level resources are always visible to every namespace in the cluster.

Default” namespace in Kubernetes:

Kubernetes comes with three namespaces out-of-the-box. They are:

1. **default:** As its name implies, this is the namespace that is referenced by default for every Kubernetes command, and where every Kubernetes resource is located by default. Until new namespaces are created, the entire cluster resides in ‘default’.
2. **kube-system:** Used for [Kubernetes components](https://www.vmware.com/topics/glossary/content/components-kubernetes.html) and should be avoided.
3. **kube-public:** Used for public resources. Not recommended for use by users.

There are many use cases for Kubernetes namespaces, including:

* Allowing teams or projects to exist in their own virtual clusters without fear of impacting each other’s work.
* Enhancing role-based access controls (RBAC) by limiting users and processes to certain namespaces.
* Enabling the dividing of a cluster’s resources between multiple teams and users via resource quotas.
* Providing an easy method of separating development, testing, and deployment of containerized applications enabling the entire lifecycle to take place on the same cluster.

Command to create namespace 🡪 **kubectl create namespace <namespace-name>**

6) What is Kubernetes Deployment?

A Kubernetes Deployment tells [Kubernetes](https://tanzu.vmware.com/kubernetes-vs-docker) how to create or modify instances of the pods that hold a [containerized application](https://tanzu.vmware.com/containers). Deployments can help to efficiently scale the number of replica pods, enable the rollout of updated code in a controlled manner, or roll back to an earlier deployment version if necessary. Kubernetes deployments are completed using kubectl.

Deployment is an object that manages the deployment and scaling of a set of pods. It provides a declarative way to define and maintain the desired state of applications running in the cluster.

Deployment set Manifest File (Yaml File)

**apiVersion**: apps/v1

**kind**: Deployment

**metadata**:

**name**: nginx-deployment

**labels**:

**app**: nginx

**spec**:

**replicas**: 3

**selector**:

**matchLabels**:

**app**: nginx

**template**:

**metadata**:

**labels**:

**app**: nginx

**spec**:

**containers**:

- **name**: nginx

**image**: nginx:1.14.2

**ports**:

- **containerPort**: 80

7) what is Replica set?

A Replica Set is defined with fields, including a selector that specifies how to identify Pods it can acquire, a number of replicas indicating how many Pods it should be maintained, and a pod template specifying the data of new Pods it should create to meet the number of replicas criteria. A Replica Set then fulfills its purpose by creating and deleting Pods as needed to reach the desired number. When a Replica Set needs to create new Pods, it uses its Pod template.

ReplicaSet is an object that ensures a specified number of identical pods are running at all times. It is a lower-level construct than a Deployment but is often used by Deployments internally to manage the pod replicas.

Replica set Manifest File (Yaml File)

**apiVersion**: apps/v1

**kind**: ReplicaSet

**metadata**:

**name**: frontend

**labels**:

**app**: guestbook

**tier**: frontend

**spec**:

*# modify replicas according to your case*

**replicas**: 3

**selector**:

**matchLabels**:

**tier**: frontend

**template**:

**metadata**:

**labels**:

**tier**: frontend

**spec**:

**containers**:

- **name**: php-redis

**image**: gcr.io/google\_samples/gb-frontend:v3

8) what is Deamon set?

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

Some typical uses of a DaemonSet are:

* running a cluster storage daemon on every node
* running a logs collection daemon on every node
* running a node monitoring daemon on every node

**apiVersion: apps/v1**

**kind: DaemonSet**

**metadata:**

**name: mynode-exporter**

**labels:**

**app: mynode-exporter**

**spec:**

**selector:**

**matchLabels:**

**app: mynode-exporter**

**template:**

**metadata:**

**labels:**

**app: mynode-exporter**

**spec:**

**containers:**

**- name: mynode-exporter**

**image: prom/node-exporter:v0.18.1**

**ports:**

**- containerPort: 9100**

**hostPort: 9100**

**protocol: TCP**

9) what is Conflict map ?

Config Map is an API object used to store non-sensitive configuration data that can be consumed by pods, containers, or other resources in the cluster. It provides a way to decouple configuration from the application code, allowing for easier management and modification of configuration settings without redeploying the application. Config Map can store key-value pairs or even entire configuration files.

10) what is secrets ?

A Secret is an object that contains a small amount of sensitive data such as a password, a token, or a key. Such information might otherwise be put in a [Pod](https://kubernetes.io/docs/concepts/workloads/pods/) specification or in a [container image](https://kubernetes.io/docs/reference/glossary/?all=true#term-image). Using a Secret means that you don't need to include confidential data in your application code.

Because Secrets can be created independently of the Pods that use them, there is less risk of the Secret (and its data) being exposed during the workflow of creating, viewing, and editing Pods. Kubernetes, and applications that run in your cluster, can also take additional precautions with Secrets, such as avoiding writing secret data to nonvolatile storage.

11) what are the services in Kubernetes and its types?

There are four types of Kubernetes services — ClusterIP, NodePort, LoadBalancer and ExternalName. The type property in the Service's spec determines how the service is exposed to the network.

A screenshot of a diagram

Description automatically generated with low confidence

ClusterIP: Exposes the Service on an internal cluster IP address for communication within the cluster.

NodePort: Exposes the Service on a static port on each node, making it accessible from outside the cluster.

LoadBalancer: Provisions a cloud load balancer with an external IP address to distribute traffic to the Service.

ExternalName: Redirects requests to an external DNS name, useful for integrating with external services.

Headless: Allows direct access to individual pods without load balancing, each pod gets its own DNS record

12) what is Nodeport (port range) ?

NodePort is a type of Service that exposes the Service on a static port on each node in the cluster. The port range for NodePort is between 30000 and 32767.

13) what is cluster IP?

* ClusterIP is the default and most common service type.
* Kubernetes will assign a cluster-internal IP address to ClusterIP service. This makes the service only reachable within the cluster.
* You cannot make requests to service (pods) from outside the cluster.
* You can optionally set cluster IP in the service definition file.

**Use Cases**

* Inter service communication within the cluster. For example, communication between the front-end and back-end components of your app.

**apiVersion: apps/v1**

**kind: Deployment**

**metadata:**

**name: mginx-deployment**

**spec:**

**selector:**

**matchLabels:**

**app: nginx-app**

**replicas: 3**

**template:**

**metadata:**

**labels:**

**app: nginx-app**

**spec:**

**containers:**

**- name: nginx**

**image: nginx:1.13.12**

**ports:**

**- containerPort: 80**

**---**

**apiVersion: v1**

**kind: Service**

**metadata:**

**name: clusterip-svc**

**spec:**

**selector:**

**app: nginx-app**

**type: ClusterIP**

**ports:**

**- port: 80**

**targetPort: 80**

14) what is NodePort ?

* NodePort service is an extension of ClusterIP service. A ClusterIP Service, to which the NodePort Service routes, is automatically created.
* It exposes the service outside of the cluster by adding a cluster-wide port on top of ClusterIP.
* NodePort exposes the service on each Node’s IP at a static port (the NodePort). Each node proxies that port into your Service. So, external traffic has access to fixed port on each Node. It means any request to your cluster on that port gets forwarded to the service.
* You can contact the NodePort Service, from outside the cluster, by requesting <NodeIP>:<NodePort>.
* Node port must be in the range of 30000–32767. Manually allocating a port to the service is optional. If it is undefined, Kubernetes will automatically assign one.
* If you are going to choose node port explicitly, ensure that the port was not already used by another service.

**Use Cases**

* When you want to enable external connectivity to your service.
* Using a NodePort gives you the freedom to set up your own load balancing solution, to configure environments that are not fully supported by Kubernetes, or even to expose one or more nodes’ IPs directly.
* Prefer to place a load balancer above your nodes to avoid node failure.

**apiVersion: v1**

**kind: Service**

**metadata:**

**name: my-frontend-service**

**spec:**

**type: NodePort**

**selector:**

**app: web**

**ports:**

**- name: http**

**protocol: TCP**

**port: 80**

**targetPort: 8080**

**nodePort: 30000**

15) what is Load balancer in Kubernetes ?

* LoadBalancer service is an extension of NodePort service. NodePort and ClusterIP Services, to which the external load balancer routes, are automatically created.
* It integrates NodePort with cloud-based load balancers.
* It exposes the Service externally using a cloud provider’s load balancer.
* Each cloud provider (AWS, Azure, GCP, etc) has its own native load balancer implementation. The cloud provider will create a load balancer, which then automatically routes requests to your Kubernetes Service.
* Traffic from the external load balancer is directed at the backend Pods. The cloud provider decides how it is load balanced.
* The actual creation of the load balancer happens asynchronously.
* Every time you want to expose a service to the outside world, you have to create a new LoadBalancer and get an IP address.

**Use Cases**

* When you are using a cloud provider to host your Kubernetes cluster.

**apiVersion: v1**

**kind: Service**

**metadata:**

**name: my-frontend-service**

**spec:**

**type: LoadBalancer**

**clusterIP: 10.0.171.123**

**loadBalancerIP: 123.123.123.123**

**selector:**

**app: web**

**ports:**

**- name: http**

**protocol: TCP**

**port: 80**

**targetPort: 8080**

16) Kubernetes most used Commands ?

Cluster Creation:

* **kops create cluster --name <cluster-name> --zones <availability-zones>**: Creates a Kubernetes cluster using **kops**. Specify the cluster name and availability zones where the cluster should be deployed.

Cluster Deletion:

* **kops delete cluster --name <cluster-name> --yes**: Deletes a Kubernetes cluster created with **kops**. Specify the cluster name and use the **--yes** flag to confirm the deletion.

Resource Creation or Update:

* **kubectl apply -f <manifest-file>**: Applies or updates Kubernetes resources defined in a manifest file. Use the **-f** flag followed by the file path to the manifest file.

Resource Deletion:

* **kubectl delete -f <manifest-file>**: Deletes Kubernetes resources defined in a manifest file. Use the **-f** flag followed by the file path to the manifest file.

Cluster Info:

* **kubectl cluster-info**: Displays information about the Kubernetes cluster, including the API server endpoint, cluster version, and more.

Cluster Nodes:

* **kubectl get nodes**: Lists all the nodes in the Kubernetes cluster, along with their status and other details.

Pods:

* **kubectl get pods**: Lists all the pods running in the cluster, along with their status, IP address, and other information.
* **kubectl describe pod <pod-name>**: Provides detailed information about a specific pod, including its current state, events, and container statuses.

Services:

* **kubectl get services**: Lists all the services in the cluster, including their types, cluster IP, and external endpoints.
* **kubectl describe service <service-name>**: Provides detailed information about a specific service, including its type, endpoints, and associated pods.
* Deployments:
* **kubectl create deployment <deployment-name> --image=<image>**: Creates a new deployment using the specified image.
* **kubectl get deployments**: Lists all the deployments in the cluster.
* **kubectl describe deployment <deployment-name>**: Provides detailed information about a specific deployment.
* ReplicaSets:
* **kubectl get replicasets**: Lists all the ReplicaSets in the cluster.
* **kubectl describe replicaset <replicaset-name>**: Provides detailed information about a specific ReplicaSet.
* Pod Logs:
* **kubectl logs <pod-name>**: Displays the logs of a specific pod.
* **kubectl logs -f <pod-name>**: Streams the logs of a specific pod in real-time.
* Pod Exec:
* **kubectl exec -it <pod-name> -- <command>**: Executes a command inside a specific pod.
* Namespace:
* **kubectl create namespace <namespace-name>**: Creates a new namespace.
* **kubectl get namespaces**: Lists all the namespaces in the cluster.
* **kubectl describe namespace <namespace-name>**: Provides detailed information about a specific namespace.
* Secrets:
* **kubectl create secret <secret-type> <secret-name> --from-literal=<key>=<value>**: Creates a new secret.
* **kubectl get secrets**: Lists all the secrets in the cluster.