Managing Kubernetes deployments

A Kubernetes service can be used to easily expose an application deployed on a set of pods using a single endpoint. A service is both a REST object and an abstraction that defines:

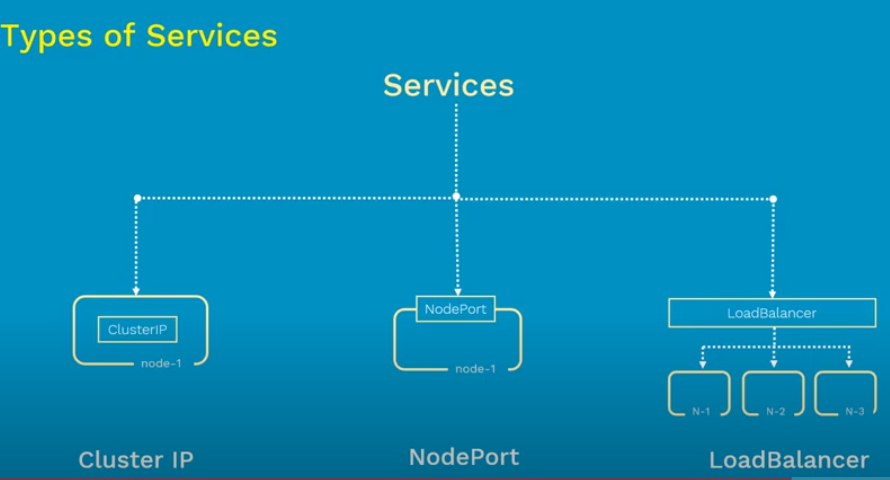
# A set of pods

# A policy to access them

Kubernetes services utilize selectors to target a set of pods:

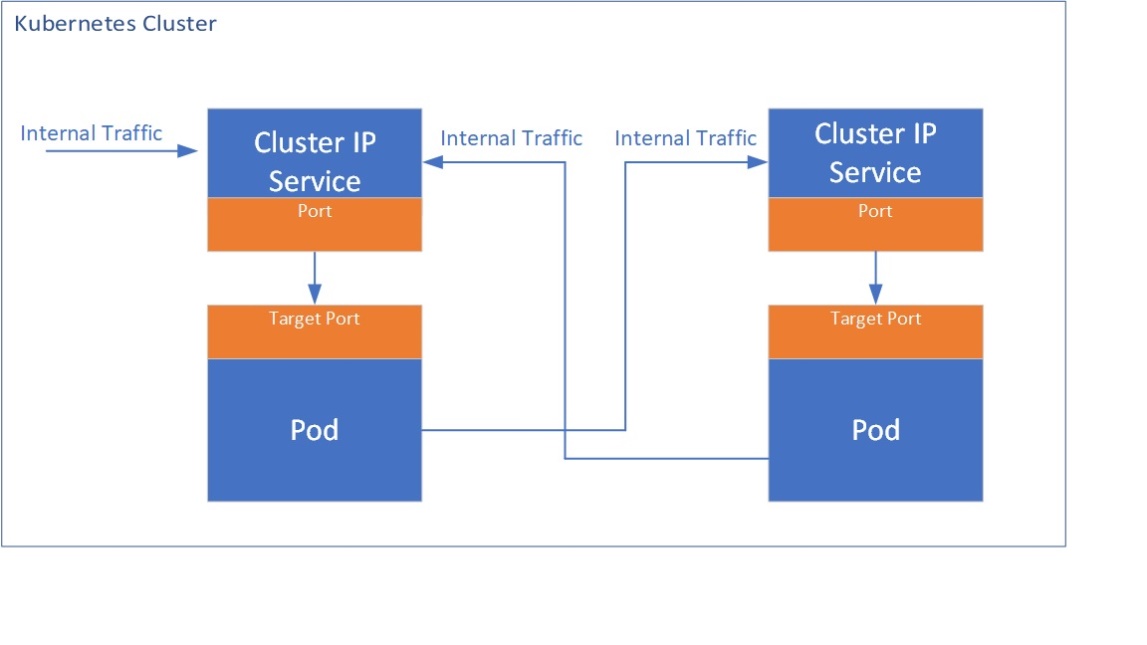
* **For native Kubernetes applications** (which use Kubernetes APIs for [service discovery](https://www.bmc.com/blogs/it-discovery/)), the endpoint API will be updated whenever there are changes to the pods in the service.
* **Non-native applications** can use virtual-IP-based bridge or load balancer implementation methods offered by Kubernetes to direct traffic to the backend pods.

## Types of Kubernetes services



1. **ClusterIP:**

It exposes the service within the Kubernetes cluster. This Service is only reachable from within the cluster. It can not be accessed from outside the cluster.



apiVersion: v1

kind: Service

metadata:

name: clusterip-demo-service

spec:

selector:

app: myapp

type: ClusterIp

ports:

- protocol: TCP

port: 80

targetPort: 8085

**NodePort:**  
It will expose the service on a static port on the deployed node. This service can be accessed from outside the cluster using the NodeIP:Nodeport.

apiVersion: v1

kind: Service

metadata:

name: nodeport-demo-service

spec:

selector:

app: myapp

type: NodePort

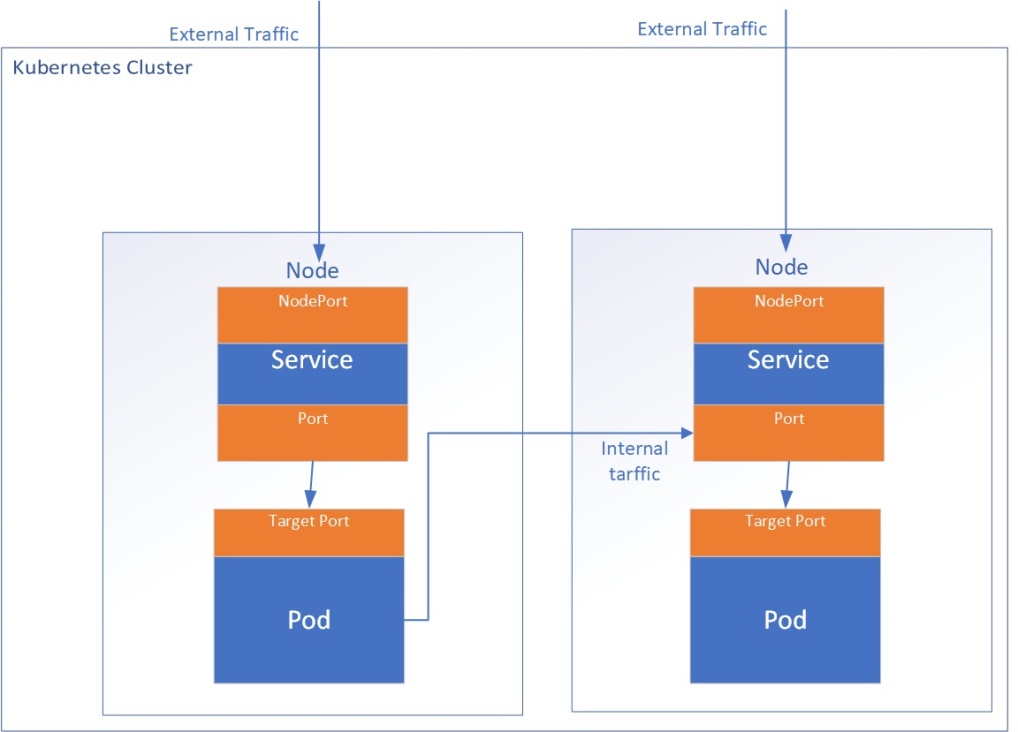
ports:

- protocol: TCP

port: 80

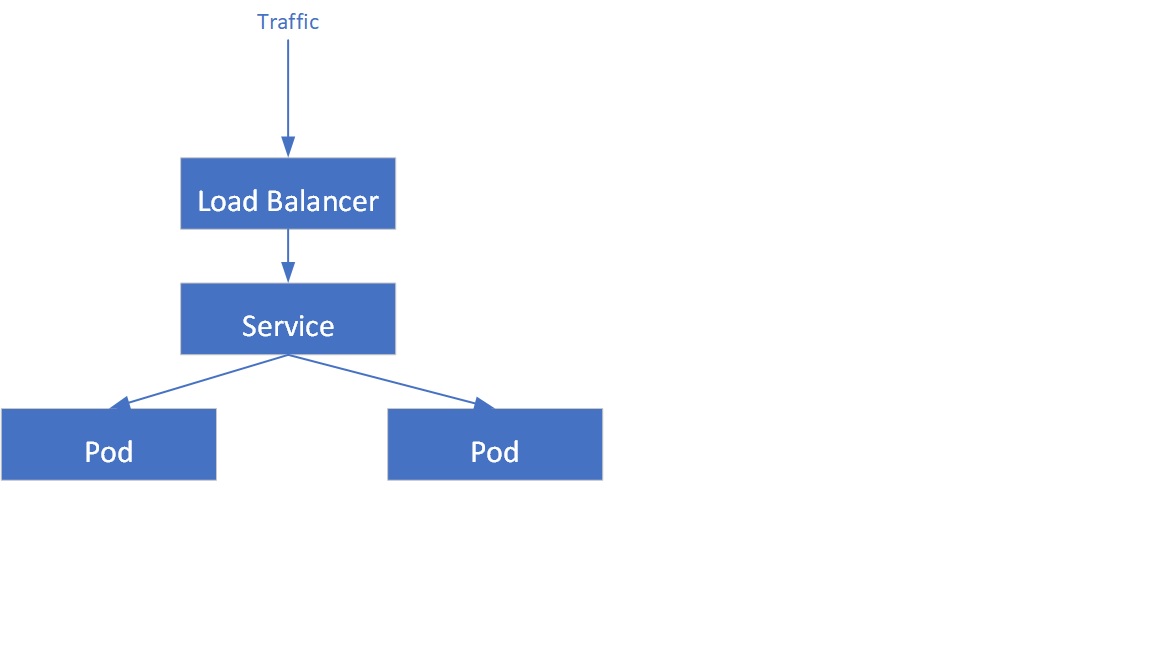
targetPort: 8085

nodePort: 30070



**Load Balancer:**

Exposes the Service externally using a cloud provider's load balancer, this creates a Public IP on the Cloud provider



apiVersion: v1

kind: Service

metadata:

name: loadbalancer-demo-service

spec:

selector:

app: myapp

ports:

- protocol: TCP

port: 80

targetPort: 8085

type: LoadBalancer

**ExternalName:**It  maps the Service to the contents of the externalName field by returning a CNAME record

### Defining a service with a selector

In the following example, we have defined a simple service exposing port 80 in the service while targeting port 8080 in the Pods with the selector label “app=webserver-nginx”

apiVersion: v1

kind: Service

metadata:

name: nginx-service

spec:

selector:

app: webserver-nginx

ports:

- protocol: TCP

port: 80

targetPort: 8080

In this example, we have exposed both ports 80 and 443 to target ports 8080 and 8090 to route HTTP and HTTPS traffic to underlying pods using the selector “app=webserver-nginx-multiport”

apiVersion: v1

kind: Service

metadata:

name: nginx-service

spec:

selector:

app: webserver-nginx-multiport

ports:

- name: http

protocol: TCP

port: 80

targetPort: 8080

- name: https

protocol: TCP

port: 443

targetPort: 8090

## Creating a Kubernetes service

apache-deployment.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: apache-deployment

labels:

app: webserver

spec:

replicas: 4

selector:

matchLabels:

app: webserver

template:

metadata:

labels:

app: webserver

spec:

containers:

- name: apache

image: bitnami/apache:latest

ports:

- containerPort: 80

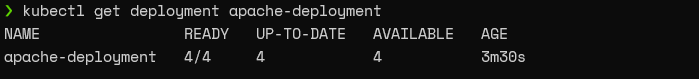
Create the deployment.

kubectl apply -f apache-deployment.yaml

apache deployment

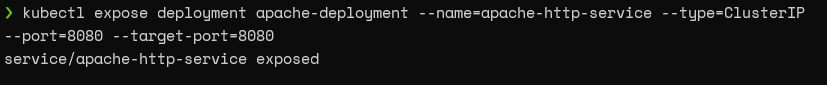
Check the status of the deployment.

kubectl get deployment apache-deployment



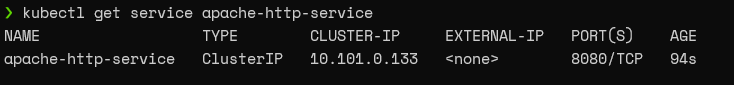
apache-deployment” with the ClusterIP service type. We will point the service to port 8080 as the “bitnami/apache” image uses port 8080 as the HTTP port.

kubectl expose deployment apache-deployment --name=apache-http-service --type=ClusterIP --port=8080 --target-port=8080



Check whether the service is created correctly.

kubectl get service apache-http-service



## Discovering Kubernetes services

You can use two methods to discover a service—DNS or Environment Variable.

### DNS

This is the preferred method for service discovery. Here, the DNS server is added to the Kubernetes cluster that watches the Kubernetes API and creates DNS records for each new service. When the DNS is enabled, cluster-wide all Pods will be able to perform name resolution of services.

### Environment Variables

In this method, kubelet adds environment variables to Pods for each active service. When using this method, the desired service must be created before creating the Pods which will use that service. Otherwise, the Pods would not get the necessary environment variables for the desired service.

## Headless Services

When load-balancing and single service IP are not required, users can create a headless service by explicitly specifying “none” for the cluster IP field (.spec.clusterIP). These headless services do not have associated Cluster IPs, and no load balancing or proxying is provided for them by the platform. The Kube-proxy also ignores these services.

DNS configuration of these services depends on the selectors defined in the services.

* **Headless service with selectors.**The endpoint controller will create the endpoint records in the API, modifying the DNS record to return an A record that points to the necessary Pods.
* **Headless service without selectors.**The endpoint controller will not create any endpoint records without having the selectors configured.

Commands

kubectl apply -f apache-deployment.yaml

kubectl get deployment apache-deployment

kubectl expose deployment apache-deployment --name=apache-http-service --type=ClusterIP --port=8080 --target-port=8080

kubectl get service apache-http-service