**Kubernetes Services**

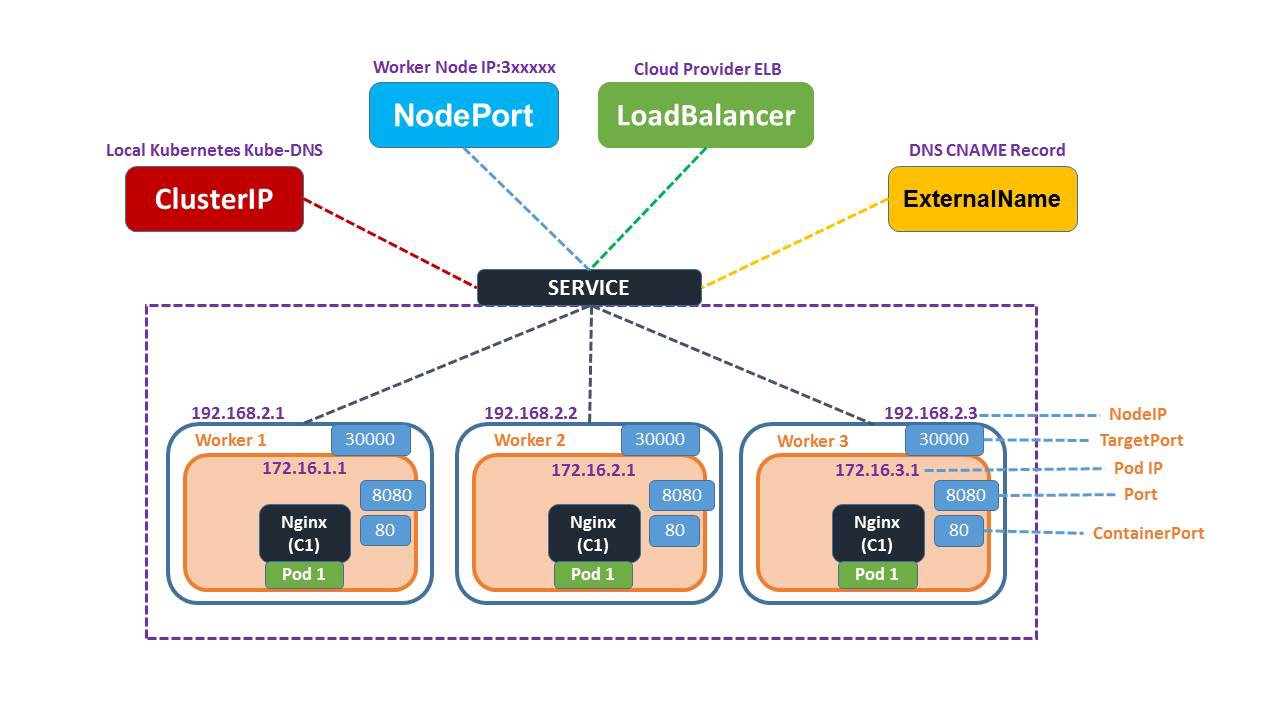
* So far we have learnt how to create Pods, Deployments and other controllers which help in deploying the containerized applications.
* We also have learnt each pods gets IP address.
* How can we connect to the applications running in the pods from external and also with in cluster.
* To help with this we have Kubernetes services. Services allows us to make logical set of pods discoverable and accessible for other pods running inside the cluster or to the external world.
* Kubernetes p[ods](https://kubernetes.io/docs/concepts/workloads/pods/) are ephemeral in nature. [Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/) object(s) can create and destroy pods dynamically. Each pod does have its own IP address, hence in a deployment, the set of pods running change all the time, so do the IP address for the pods.
* This leads to a problem: if some set of pods (call them “backends”) provides functionality to other pods (call them “frontends”) inside your cluster, how do the frontends find out and keep track of which IP address to connect to, so that the frontend can use the backend part of the workload?
* Let’s try to understand the problem by considering an example consisting of a set of different pods as following:
* *A set of pods that serve****frontend****requests  
  A set of pods that serve****backend****requests  
  A set of pods that server****data base****requests*

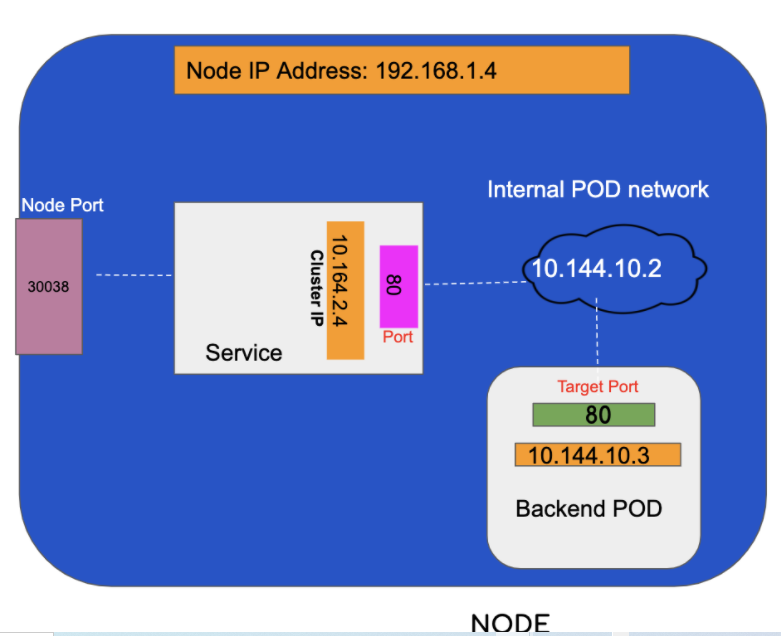
If we deployed a pod that is hosting an application. We cannot directly talk to the pod using its IP address.

This is where the service object in K8s helps; it’s like the glue that connects different objects in K8s (similar to what routers do in networking, i.e., connect different networks). The service is like a virtual server and has its own IP address within the K8s cluster it resides in.

Kubernetes ServiceTypes allow you to specify what kind of service you want. The default is ClusterIP.

Let’s look into each of the serviceTypes:

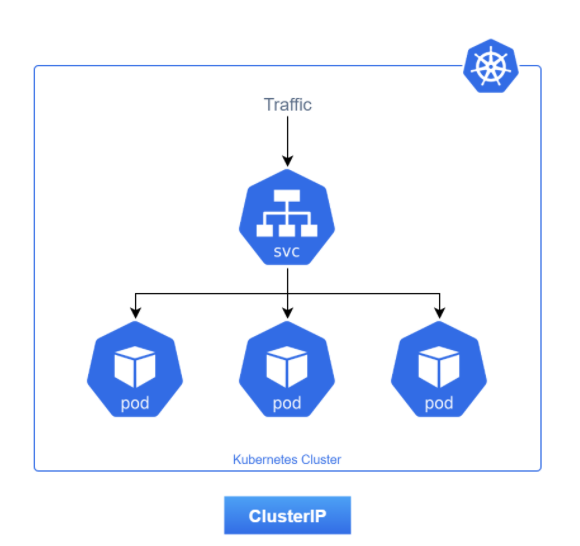




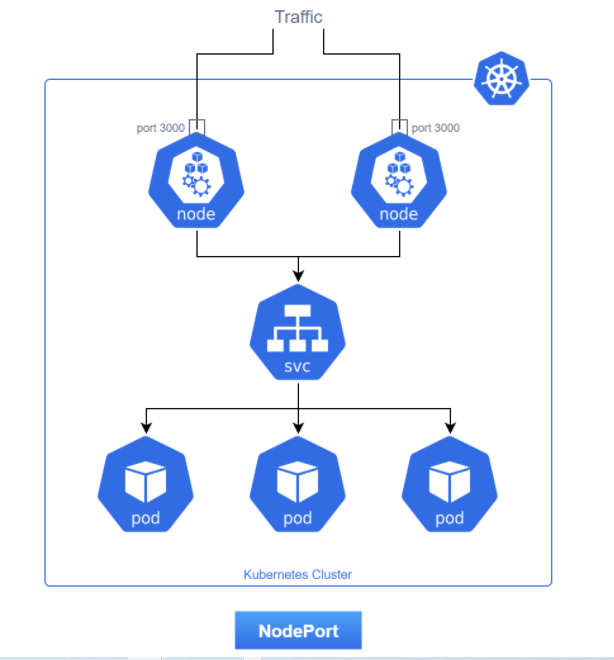
There can be seen 3 different ports

* NodePort: 30038
* Service Port : 80
* Target Port(Pod level): 80

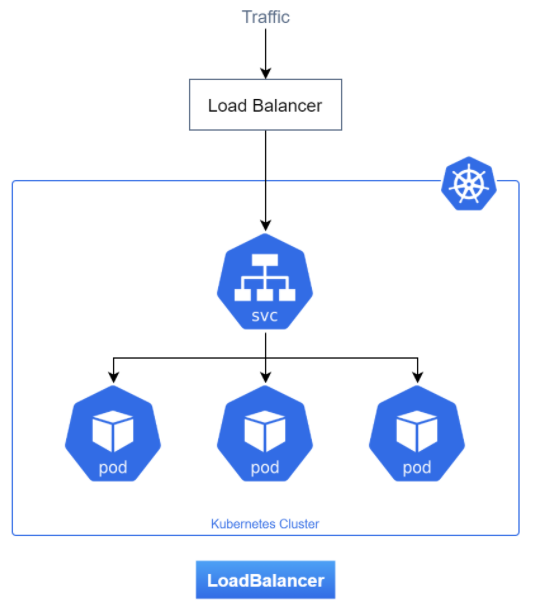
Cluster IP:



Nodeport:



Loadbalancer:



NodePort service helps expose the Service on each Node’s IP at a static port (the NodePort). NodePort The port is available to all the workers in the cluster. A ClusterIP Service, to which the NodePort Service routes are automatically created. One would be able to contact the NodePort Service, from outside the cluster, by requesting <NodeIP>:<NodePort>.

The port on the POD is called the targetPort and the one connecting the NodePort service to the POD is called port.

All this means if any request coming into port 30080 to the cluster on any worker node will be forwarded to the “Node Port Service,” which in turn will forward the request to the underlying Pod at port 80.

NodePort example

Let’s try to understand the concepts here with some practical examples.

Let’s start with creating a deployment using the YAML file below. Some key things to note, each container is using the port 80 and has a label called app:nginx

apiVersion: apps/v1

kind: Deployment

metadata:

name: my-nginx-deploy

labels:

app: nginx

spec:

replicas: 2

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: test-nginx

image: nginx:alpine

ports:

- containerPort: 80

**kubectl apply -f nginx-deploy.yml**

**kubectl get pods**

To check IPs assigned to the pods are:

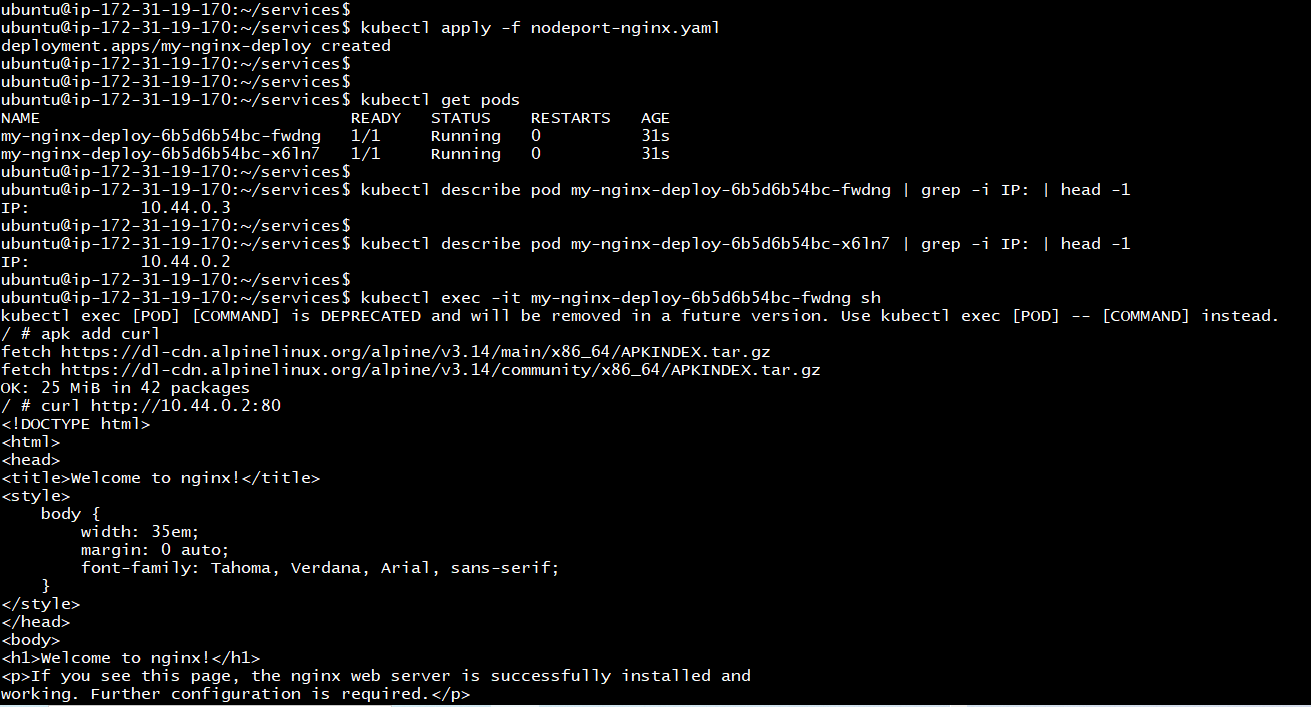
**kubectl describe pod <pod\_name> | grep -i IP: | head -1**

Can Pods above talk to each other? Let’s find out. We will try to connect to a pod and try reaching the other pod from there.

**kubectl exec -it <pod\_name> sh**

**# apk add curl**  
/ # **curl** http://<ipaddress of 2nd pod>:80

**#exit**



t so looks like the pods can talk to each other. Some things to note here

* Pods are ephemeral; the IP address for that pod is most likely to change. Therefore we cannot rely on IPs to communicate.
* Each Pod gets its IP address. Therefore, it would be impossible for the client to know about the IP addresses of all pods.
* Also, the pod IP addresses above cannot be reached externally; try it!

***So how do we reach the PODs externally?***

To reach the pods from outside the cluster, one needs to expose the port on the host machine to redirect the traffic to a port of the container. **NodePort Service provides that capability**.

Let’s create a NodePort service.

**apiVersion: v1**

**kind: Service**

**metadata:**

**name: my-service**

**spec:**

**type: NodePort**

**selector:**

**app: nginx**

**ports:**

**# By default the `targetPort` is set to the same value as the `port` field.**

**- port: 80**

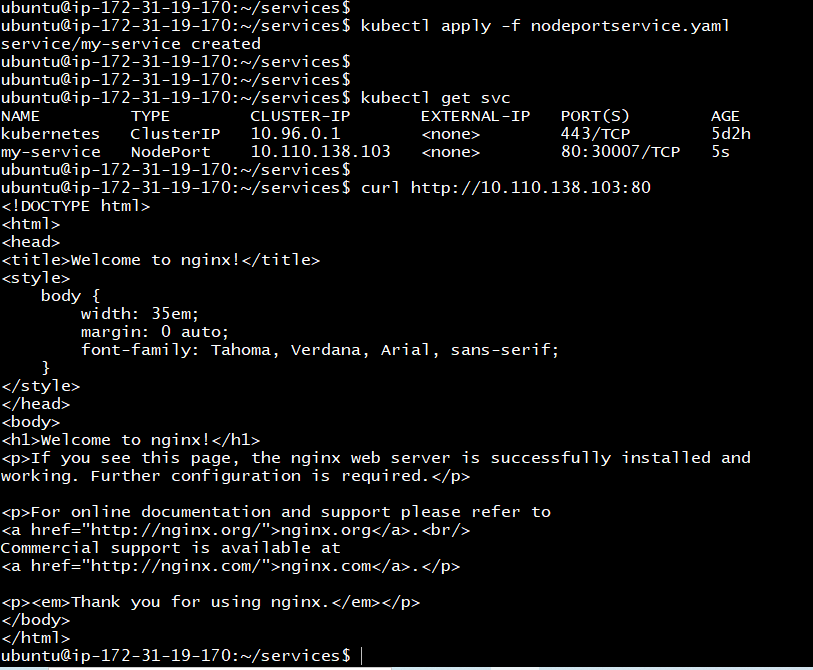
**targetPort: 80**

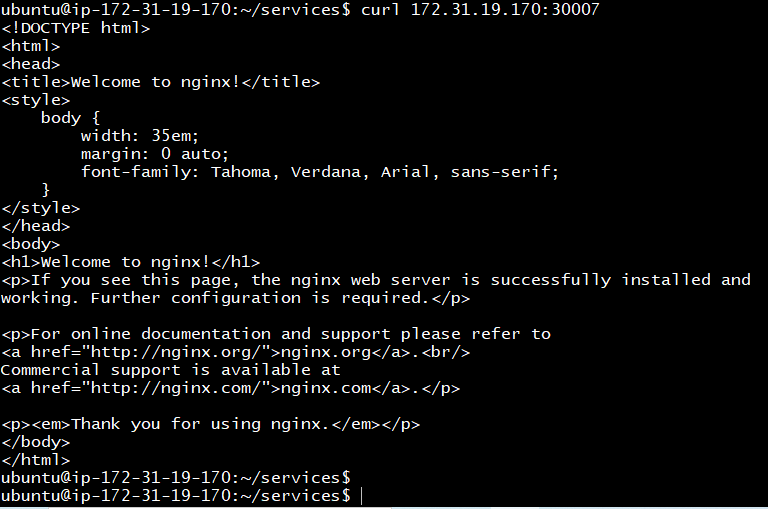
**# Optional field**

**# By default and for convenience, the Kubernetes control plane will allocate a port from a range (default: 30000-32767)**

**nodePort: 30007**

kubectl apply -f







Cluster IP:

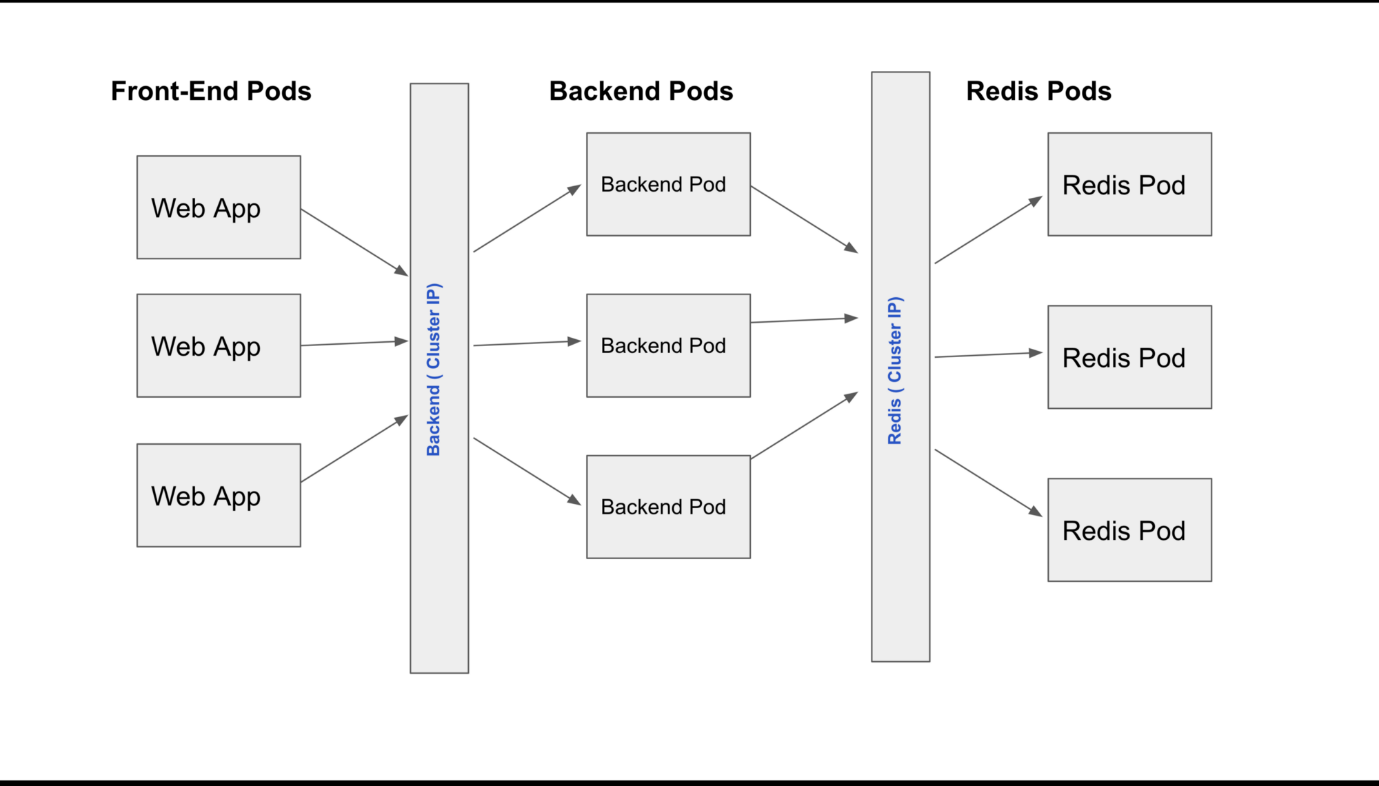
Let’s Understand ClusterIP Further With One More Example:

**In any given cluster node there can be multiple types of Pods viz**

* **Front-end pod**
* **Backend** pods
* **Redis** pods
* **Database** MySQL pods

**etc…**

Each of these types of **pods** lying within an internal cluster will have a different internal network IP, which is liable to change. In order, to talk to each other. For example, the front-end pod may be talking to the backend pods, backend pods, in turn, may require to talk to **Redis** pods.



As can be clearly seen that whenever a front-end pod wants to communicate with backend pods, it has to simply communicate to a ClusterIP service named **backend**, which is a single endpoint service managing the communication to all backend pods. Similarly, if backend pods need to access the Redis cache service it has to make a service call to cluster IP service named **Redis**, which will allow the backend pods to communicate to respective Redis pods.

**To summarize**: These common backend / Redis services are the **virtual IP’s** also known as C**lusterIP type service,** which effectively allows one pod type to communicate to another Pod type in the given cluster Node.

**cluster-ip-demo.yaml file:**

**apiVersion**: v1  
**kind**: Service  
**metadata**:  
 **name**: Backend  
**spec**:  
 **type**: ClusterIP  
 **ports**:  
 - **targetPort**: 80  
 **port**: 80