**Agenda: Services and Ingress**

* Kubernetes Services
* Service Types
  + ClusterIP Service
  + NodePort Service
  + LoadBalancer Service
  + External Service
* Ingress Controllers and Alternatives
* Setting up Ingress Locally
* Creating the Ingress Config

Kubernetes Services

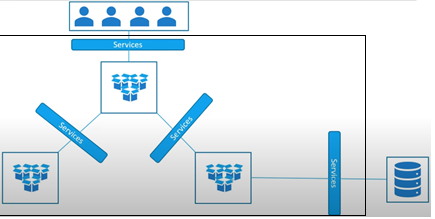
**The Problem:**

* In Kubernetes, if you use a Deployment to run your app, it can create and destroy Pods dynamically. Each Pod gets its own IP address, however in a Deployment, the set of Pods running in one moment in time could be different from the set of Pods running that application a moment later.
* This can lead to a problem: if some set of Pods (call them “backends”) provides functionality to other Pods (call them “frontends”) inside our 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?

This is where **Services** can be helpful.

**Service Resources:**

* Kubenetes Service is an abstract way to **expose** an application(Microservice) running on a **logical set of Pods** and a **policy** by which to access them.
* Services enable a **loose coupling** between dependent Pods. Frontend Pods don’t need to know the direct IP address of the Backend Pods.



* A **Service** in Kubernetes is a **REST object**, similar to a Pod.
* Like all of the REST objects, you can POST a Service definition to the **API server** to create a new instance.
* The name of a Service object must be a valid **DNS label name**.

apiVersion: apps/v1

kind: Deployment

metadata:

  name: nginx-deployment

spec:

  replicas: 10

  selector:

    matchLabels:

      app: **nginx-app**

  template:

    metadata:

      labels:

        app: nginx-app

    spec:

      containers:

      - name: mynginx-container

        image: nginx

        ports:

        - containerPort: 80

---

apiVersion: v1

kind: Service

metadata:

  name: my-service-cip

spec:

  type: ClusterIP

  selector:

    app: nginx-app

  ports:

    - protocol: TCP

      port: 8080

      targetPort: 80

* This specification creates a new Service object named “**my-service-cip**”, which targets TCP port **80** on any Pod with the **app=nginx-app** label.
* Kubernetes assigns this Service an IP address (sometimes called the "cluster IP").
* The controller for the Service selector continuously scans for Pods that match its selector, and then POSTs any updates to an **Endpoint object** also named “my-service”.

**Multi-Port Services**

For some Services, you need to expose more than one port. Kubernetes lets you configure multiple port definitions on a Service object. When using multiple ports for a Service, you must give all of your ports names so that these are unambiguous. For example:

apiVersion: v1

kind: Service

metadata:

  name: my-service-cip

spec:

  type: ClusterIP

  selector:

    app: MyApp

  ports:

    - name: http-port

      protocol: TCP

      port: 8080

      targetPort: 80

    - name: https-port

      protocol: TCP

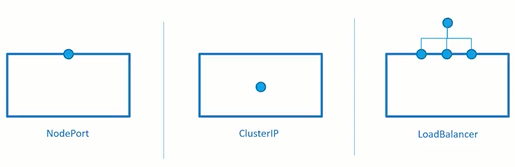
      port: 44433

      targetPort: 443

**ServiceTypes:**

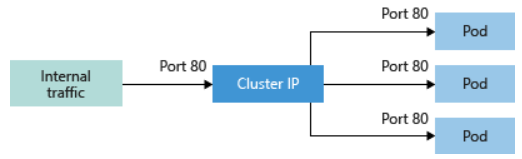
* Pod IP address are not exposed outside the cluster without a Service. Services allow our applications to receive traffic from outside the cluster.
* For some parts of our application (for example, frontends) we may want to expose a Service on to an external IP address, i.e. outside of our cluster.
* Kubernetes ServiceTypes allow us to specify what kind of Service we want.

1. ClusterIP
2. LoadBalancer
3. NodePort
4. ExternalName



**CLUSTER-IP SERVICE**

* Exposes the Service on a **cluster-internal IP**. Choosing this value makes the Service **only reachable** from within the cluster.
* This is the **default** ServiceType.
* ClusterIP is the preferred option for internal/backend service access and uses an internal/private IP address to access the service.
* Some examples of where ClusterIP might be the best option include service debugging during development and testing and internal traffic.



**Steps to create a ClusterIP Service:**

1. kubectl **create** deployment nginx-deployment --image nginx
2. kubectl **expose** deploy/nginx-deployment --name my-service-cip **--type=ClusterIP** --port 8080 --target-port=80

Note: Expose uses service creating generator to create the service.

If the service name is not mentioned, it uses the name of deployment. In this case, it would be **nginx-deployment**

1. **kubectl get all** OR **kubectl get services**

Note the ClusterIP for further use (eg: 10.99.158.232)

**Testing if the service is forwarding the traffic to Pods**

1. kubectl get services
2. kubectl get pods -o wide
3. kubectl get endpoints
4. kubectl describe endpoints my-service-cip
5. kubectl **run** myclient -it **--rm** --image=nginx **--restart=Never** -- **sh**

# curl <http://my-service-cip:8080>

# exit

1. kubectl run -it **--rm** myclient --image=nginx --restart=Never -- **curl http://my-service-cip:8080**

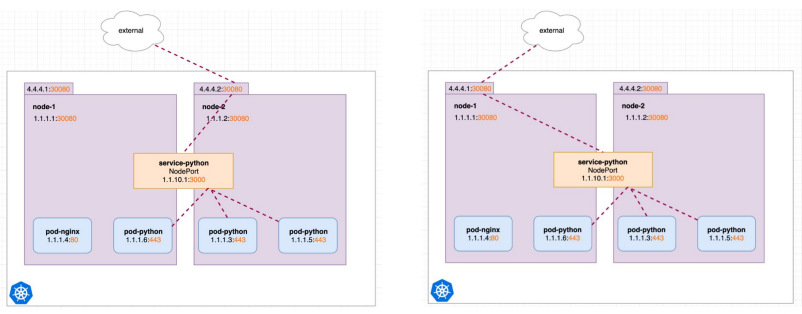
Any of the following curl commands can be used.

* 1. curl <http://10.99.158.232:8080>
  2. curl <http://my-service-cip:8080>
  3. curl <http://my-service-cip.default.svc.cluster.local:8080> (Use this when the POD invoking the service is in different namespace)

Note: To list IP and FQDN**: nslookup my-service-cip**.

**NODEPORT SERVICE**:

* Exposes the Service on each Node’s IP at a static port (the NodePort [**Hi Range Port**]). High range port (30000 to 32767) used for exposing NodePort.
* Each node proxies that port (the same port number on every Node) into your Service
* A ClusterIP Service, to which the NodePort Service routes, is automatically created.
* Useful when front end pods are to be exposed outside the cluster for users to access it.
* We will be able to contact the NodePort Service, **from outside the cluster**, by requesting **<NodeIP>:<NodePort>.**
* Same NodePort will be used across all the nodes in a multi node cluster. Also, same service (single instance) can be used to access container in any Pod or any Node in the cluster.



**Create NodePort type of Service:**

1. **kubectl expose deployment nginx-deployment --target-port=80 --type=NodePort --name=my-service-np1 --selector="app=nginx-app" --port=8080**

Note: the above command **doesn’t have option** to provide node-port and it will be randomly picked up from range of 30000 to 32767

1. kubectl get service

Note the High Port under PORT(S) section (Eg: 30001)

1. **For Docker Desktop**

curl http://localhost:<high-port> or open in web browser

**OR**

**For Minikube:**

minikube ip

curl http://<minikube-ip>:30001

We can access the service through localhost without getting into the cluster. This works only in Docker and not in minicube.

1. kubectl get svc **my-service-np** -o yaml

|  |  |
| --- | --- |
| apiVersion: v1  kind: Service  metadata:    name: my-service-np  spec:    type: NodePort    selector:      app: nginx-app    ports:    - protocol: TCP      nodePort: 30008 #Port of Node      port: 8080 #Port of Service      targetPort: 80 #Port of Container | Note: 3 ports are involved   1. 30008 (High Range Port) if IP of Node is used 2. 8080 if ClusterIP is used 3. 80 is Application port in container |

**LOAD BALANCER SERVICE**

Exposes the Service externally using a **cloud provider’s load balancer**. NodePort and ClusterIP Services, to which the external load balancer routes, are automatically created and are hidden from us. It assigns a fixed external IP to the Service. It is superset of NodePort.

There is no filtering, no routing, etc. This means you can send almost any kind of traffic to it, like HTTP, TCP, UDP or WebSocket's

This works at **Layer4** of OSI layers.

Diagram

Description automatically generated

**Create LoadBalancer type of Service:**

1. kubectl expose deploy/mynginx --port 8080 --target-port=80 --type=LoadBalancer --name my-service-lb
2. **curl localhost:port** or **open in web browser**

We can access the service through localhost without getting into the cluster.

apiVersion: v1

kind: Service

metadata:

  name: my-service-lb

spec:

  type: LoadBalancer

  selector:

    app: nginx-app

  ports:

    - name: http

      protocol: TCP

      port: 8080

      targetPort: 80

**Downside of LoadBalancer**

* Every LoadBalancer service exposed will gets it's own Public IP address.
* It gets very expensive to have external IP for each of the service (application)

**ExternalName Service:**

Maps the Service to the contents of the externalName field (e.g. abc.example.com), by returning a CNAME record with its value. No proxying of any kind is needed. Services of type ExternalName map a Service to a DNS name, not to a typical selector.

A picture containing diagram

Description automatically generated

apiVersion: v1

kind: Service

metadata:

  name: my-service-ext

spec:

  type: ExternalName

  externalName: my.database.com

When looking up the host **my-service-ext.default.svc.cluster.local**, the cluster DNS Service returns a **CNAME** record with the value **my.database.com**.

Accessing my-service works in the same way as other Services but with the crucial difference that redirection happens at the DNS level rather than via proxying or forwarding. Should you later decide to move your database into your cluster, you can start its Pods, add appropriate selectors or endpoints, and change the Service's type.