What are Kubernetes Services?

Kubernetes - Beginners | Intermediate | Advanced

View on GitHub Join Slack

What are Kubernetes Services?

Say, you have pods running nginx in a flat, cluster wide, address space. In theory, you could talk to these pods directly, but what happens when a node dies? The pods die with it, and the Deployment will create new ones, with different IPs. This is the problem a Service solves.

Kubernetes Pods are mortal. They are born and when they die, they are not resurrected. 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 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?

Enter Services

A Kubernetes Service is an abstraction which defines a logical set of Pods running somewhere in your cluster, that all provide the same functionality. When created, each Service is assigned a unique IP address (also called clusterIP). This address is tied to the lifespan of the Service, and will not change while the Service is alive. Pods can be configured to talk to the Service, and know that communication to the Service will be automatically load-balanced out to some pod that is a member of the Service.

Deploying a Kubernetes Service

Like all other Kubernetes objects, a Service can be defined using a YAML or JSON file that contains the necessary definitions (they can also be created using just the command line, but this is not the recommended practice). Let's create a NodeJS service definition. It may look like the following:

```
git clone https://github.com/collabnix/kubelabs
cd kubelabs/Services101/
kubectl apply -f nginx-svc.yaml
```

This specification will create a Service which targets TCP port 80 on any Pod with the run: my-nginx label, and expose it on an abstracted Service port (targetPort: is the port the container accepts traffic on, port: is the abstracted Service port, which can be any port other pods use to access the Service). View Service API object to see the list of supported fields in service definition. Check your Service

```
kubectl get svc my-nginx
```

As mentioned previously, a Service is backed by a group of Pods. These Pods are exposed through endpoints. The Service's selector will be evaluated continuously and the results will be POSTed to an Endpoints object also named my-nginx. When a Pod dies, it is automatically removed from the endpoints, and new Pods matching the Service's selector will automatically get added to the endpoints. Check the endpoints, and note that the IPs are the same as the Pods created in the first step:

```
kubectl describe svc my-nginx
```

You should now be able to curl the nginx Service on : from any node in your cluster. Note that the Service IP is completely virtual, it never hits the wire. If you're curious about how this works you can read more about the service proxy.

Accessing the Service

Kubernetes supports 2 primary modes of finding a Service - environment variables and DNS

Environment Variables

When a Pod runs on a Node, the kubelet adds a set of environment variables for each active Service. This introduces an ordering problem. To see why, inspect the environment of your running nginx Pods (your Pod name will be different):

```
kubectl exec my-nginx-3800858182-jr4a2 -- printenv | grep SERVICE
```

```
KUBERNETES_SERVICE_HOST=10.0.0.1
KUBERNETES_SERVICE_PORT=443
KUBERNETES_SERVICE_PORT_HTTPS=443
```

Note there's no mention of your Service. This is because you created the replicas before the Service. Another disadvantage of doing this is that the scheduler might put both Pods on the same machine, which will take your entire Service down if it dies. We can do this the right way by killing the 2 Pods and waiting for the Deployment to recreate them. This time around the Service exists before the replicas. This will give you scheduler-level Service spreading of your Pods (provided all your nodes have equal capacity), as well as the right environment variables:

```
kubectl scale deployment my-nginx --replicas=0; kubectl scale deployment my-
```

```
kubectl get pods -l run=my-nginx -o wide
NAME                        READY     STATUS    RESTARTS   AGE     IP
my-nginx-3800858182-e9ihh   1/1       Running   0          5s      10.244.2.
```

```
my-nginx-3800858182-j4rm4   1/1       Running   0          5s      10.244.3.
```

You may notice that the pods have different names, since they are killed and recreated.

```
kubectl exec my-nginx-3800858182-e9ihh -- printenv | grep SERVICE
```

```
KUBERNETES_SERVICE_PORT=443
MY_NGINX_SERVICE_HOST=10.0.162.149
KUBERNETES_SERVICE_HOST=10.0.0.1
MY_NGINX_SERVICE_PORT=80
KUBERNETES_SERVICE_PORT_HTTPS=443
```

DNS

Kubernetes offers a DNS cluster addon Service that automatically assigns dns names to other Services. You can check if it's running on your cluster:

```
kubectl get services kube-dns --namespace=kube-system
```

```
NAME       TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)         AGE
kube-dns   ClusterIP   10.0.0.10    <none>        53/UDP,53/TCP   8m
```

The rest of this section will assume you have a Service with a long lived IP (my-nginx), and a DNS server that has assigned a name to that IP. Here we use the CoreDNS cluster addon (application name kube-dns), so you can talk to the Service from any pod in your cluster using standard methods (e.g. gethostbyname()). If CoreDNS isn't running, you can enable it referring to the CoreDNS README or Installing CoreDNS. Let's run another curl application to test this:

```
kubectl run curl --image=radial/busyboxplus:curl -i --tty
```

```
Waiting for pod default/curl-131556218-9fnch to be running, status is Pendin
Hit enter for command prompt
```

```
Then, hit enter and run nslookup my-nginx:

[ root@curl-131556218-9fnch:/ ]$ nslookup my-nginx
Server:    10.0.0.10
Address 1: 10.0.0.10

Name:      my-nginx
Address 1: 10.0.162.149
```

Exposing the Service

For some parts of your applications you may want to expose a Service onto an external IP address. Kubernetes supports two ways of doing this: NodePorts and LoadBalancers. The Service created in the last section already used NodePort, so your nginx HTTPS replica is ready to serve traffic on the internet if your node has a public IP.

`` kubectl get svc my-nginx -o yaml | grep nodePort -C 5

```

```

uid: 07191fb3-f61a-11e5-8ae5-42010af00002 spec: clusterIP: 10.0.162.149 ports:

- name: http nodePort: 31704 port: 8080 protocol: TCP targetPort: 80
- name: https nodePort: 32453 port: 443 protocol: TCP targetPort: 443 selector: run: my-nginx ```

```
kubectl get nodes -o yaml | grep ExternalIP -C 1
    - address: 104.197.41.11
      type: ExternalIP
    allocatable:
--
    - address: 23.251.152.56
      type: ExternalIP
    allocatable:
...
```

```
$ curl https://<EXTERNAL-IP>:<NODE-PORT> -k
```

```
...
<h1>Welcome to nginx!</h1>
```

Let's now recreate the Service to use a cloud load balancer, just change the Type of my-nginx Service from NodePort to LoadBalancer:

```
kubectl edit svc my-nginx
kubectl get svc my-nginx
NAME       TYPE        CLUSTER-IP     EXTERNAL-IP        PORT(S)
my-nginx   ClusterIP   10.0.162.149   162.222.184.144    80/TCP,81/TCP,82/TC
curl https://<EXTERNAL-IP> -k
...
<title>Welcome to nginx!</title>
The IP address in the EXTERNAL-IP column is the one that is available on the
```

Note that on AWS, type LoadBalancer creates an ELB, which uses a (long) hostname, not an IP. It's too long to fit in the standard kubectl get svc output, in fact, so you'll need to do kubectl describe service my-nginx to see it. You'll see something like this:

```
kubectl describe service my-nginx
...
LoadBalancer Ingress:   a320587ffd19711e5a37606cf4a74574-1142138393.us-east-
...
```

Service Exposing More Than One Port

Kubernetes Services allow you to define more than one port per service definition. Let's see how a web server service definition file may look like:

```
apiVersion: v1
kind: Service
metadata:
  name: webserver
spec:
 selector:
   app: web
```

```
  ports:
  - name: http
    port: 80
    targetPort: 80
  - name: https
    port: 443
    targetPort: 443
```

Notice that if you are defining more than one port in a service, you must provide a name for each port so that they are recognizable.

Kubernetes Service Without Pods?

While the traditional use of a Kubernetes Service is to abstract one or more pods behind a layer, services can do more than that. Consider the following use cases where services do not work on pods:

You need to access an API outside your cluster (examples: weather, stocks, currency rates). You have a service in another Kubernetes cluster that you need to contact. You need to shift some of your infrastructure components to Kubernetes. But, since you're still evaluating the technology, you need it to communicate with some backend applications that are still outside the cluster. You have another service in another namespace that you need to reach. The common thing here is that the service will not be pointing to pods. It'll be communicating with other resources inside or outside your cluster. Let's create a service definition that will route traffic to an external IP address:

```
apiVersion: v1
kind: Service
metadata:
  name: webserver
spec:
 selector:
   app: web
  ports:
  - name: http
    port: 80
    targetPort: 80
```

```
  - name: https
    port: 443
    targetPort: 443
```

Here, we have a service that connects to an external NodeJS backend on port 3000. But, this definition does not have pod selectors. It doesn't even have the external IP address of the backend! So, how will the service route traffic then?

Normally, a service uses an Endpoint object behind the scenes to map to the IP addresses of the pods that match its selector.

Service Discovery

Let's revisit our web application example. You are writing the configuration files for Nginx and you need to specify an IP address or URL to which web server shall route backend requests. For demonstration purposes, here's a sample Nginx configuration snippet for proxying requests:

```
server {
  listen 80;

  server_name myapp.example.com;

  location /api {
      proxy_pass http://??/;
  }
}
```

The proxy_pass part here must point to the service's IP address or DNS name to be able to reach one of the NodeJS pods. In Kubernetes, there are two ways to discover services: (1) environment variables, or (2) DNS. let's talk about each one of them in a bit of detail.

Connectivity Methods

If you reached that far, you are able to contact your services by name. Whether you're using environment variables or you've deployed a DNS, you get the service name resolved

to an IP address. Now you want to be serious about it and make it accessible from outside your cluster? There are three ways to do that:

CLusterIP

The ClusterIP is the default service type. Kubernetes will assign an internal IP address to your service. This IP address is reachable only from inside the cluster. You can - optionally - set this IP in the service definition file. Think of the case when you have a DNS record that you don't want to change and you want the name to resolve to the same IP address. You can do this by defining the clusterIP part of the service definition as follows:

```
apiVersion: v1
kind: Service
metadata:
  name: external-backend
spec:
  ports:
  - protocol: TCP
    port: 3000
    targetPort: 3000
  clusterIP: 10.96.0.1
```

However, you cannot just add any IP address. It must be within the service-cluster-ip-range, which is a range of IP addresses assigned to the service by the Kubernetes API server. You can get this range through a simple kubectl command as follows:

```
kubectl cluster-info dump | grep service-cluster-ip-range
```

You can also set the clusterIP to none, effectively creating a Headless Service.

Headless Service In Kubernetes?

As mentioned, the default behavior of Kubernetes is to assign an internal IP address to the service. Through this IP address, the service will proxy and load-balance the requests to the pods behind. If we explicitly set this IP address (clusterIP) to none, this is like telling Kubernetes "I don't need load balancing or proxying, just connect me to the first available

pod".

Let's consider a common use case. If you host, for example, MongoDB on a single pod, you will need a service definition on top of it to take care of the pod being restarted and acquiring a new IP address. But you don't need any load balancing or routing. You only need the service to patch the request to the backend pod. Hence, the name: headless: a service that does have an IP.

But, what if a headless service was created and was managing more than one pod? In this case, any query to the service's DNS name will return a list of all the pods managed by this service. The request will accept the first IP address returned. Obviously, this is not the best load-balancing algorithm if at all. The bottom line here, use a headless service when you need a single pod.

NodePort

This is one of the service types that are used when you want to enable external connectivity to your service. If you're having four Nginx pods, the NodePort service type is going to use the IP address of any node in the cluster combined with a specific port to route traffic to those pods. The following graph will demonstrate the idea:

You can use the IP address of any node, the service will receive the request and route it to one of the pods.

A service definition file for a service of type NodePort may look like this:

```
apiVersion: v1
kind: Service
metadata:
  name: frontend
spec:
  type: NodePort
  ports:
    - port: 80
      nodePort: 30000
      targetPort: 80
  selector:
    app: web
```

Manually allocating a port to the service is optional. If left undefined, Kubernetes will automatically assign one. It must be in the range of 30000-32767. If you are going to choose it, ensure that the port was not already used by another service. Otherwise, Kubernetes will report that the API transaction has failed.

Notice that you must always anticipate the event of a node going down and its IP address becomes no longer reachable. The best practice here is to place a load balancer above your nodes.

LoadBalancer

his service type works when you are using a cloud provider to host your Kubernetes cluster. When you choose LoadBalancer as the service type, the cluster will contact the cloud provider and create a load balancer. Traffic arriving at this load balancer will be forwarded to the backend pods. The specifics of this process is dependent on how each provider implements its load balancing technology.

Different cloud providers handle load balancer provisioning differently. For example, some providers allow you to assign an IP address to the component, while others choose to assign short-lived addresses that constantly change. Kubernetes was designed to be highly portable. You can add loadBalancerIP to the service definition file. If the provider supports it, it will be implemented. Otherwise, it will be ignored. Let's have a sample service definition that uses LoadBalancer as its type:

```
apiVersion: v1
kind: Service
metadata:
  name: frontend
spec:
  type: LoadBalancer
  loadBalancerIP: 78.11.24.19
  selector:
    app: web
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
```

One of the main differences between the LoadBalancer and the NodePort service types is that in the latter you get to choose your own load balancing layer. You are not bound to the cloud provider's implementation

Next »

Join KubeDaily

kubelabs is maintained by collabnix.

This page was generated by GitHub Pages.