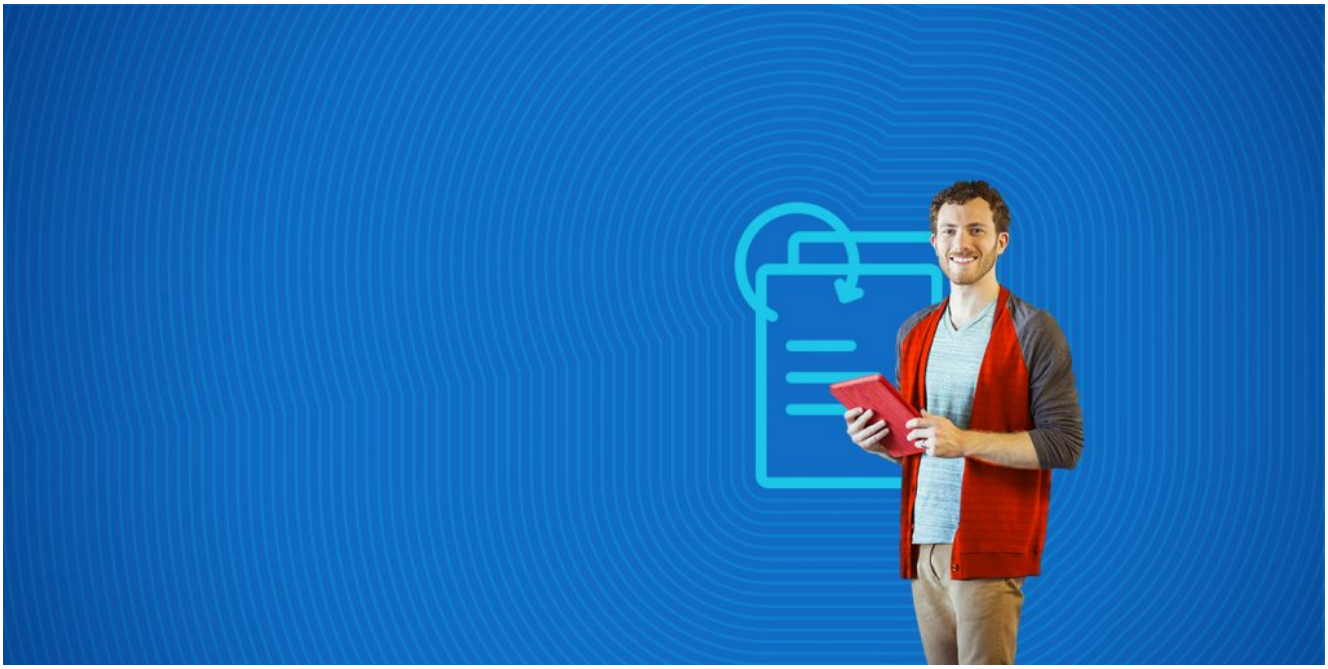

Kubernetes Services: A Beginner's Guide

In this blog post, we will talk about services, what it is used for and how to create and work with it. To understand this article, you'll want to have a decent understanding of *minikube*, *kubectl*, and deployment kind.

What is a Kubernetes service?

Like a pod, a Kubernetes service is a REST object. A service is both an abstraction that defines a logical set of pods and a policy for accessing the pod set.

Here are general attributes of a Kubernetes service:

- A label selector can find pods that are targeted by a service.
    - For K8S-native applications, the endpoints API will be updated whenever there are changes to a set of pods in a service.
    - For non-native applications, a virtual-IP-based

bridge to services redirects to backend pods.

- A service is assigned an IP address (“cluster IP”), which the service proxies use.
- A service can map an incoming port to any targetPort. (The targetPort is set, by default, to the port field’s same value. The targetPort can be defined as a string.)
- The port number assigned to each name can vary in each backend pod.
    - For example, you can change the port number that pods expose in the next version of your backend software, without breaking clients.
- Services support TCP (default), UDP and SCTP for protocols.
- Services can be defined with or without a selector.
- Services support a variety of port definitions.

Types of Kubernetes services

There are four types of Kubernetes services:

1. **ClusterIP.** This default type exposes the service on a cluster-internal IP. You can reach the service only from *within* the cluster.
2. **NodePort.** This type of service exposes the service on each node’s IP at a static port. A ClusterIP service is created automatically, and the NodePort service will route to it. From *outside* the cluster, you can contact the NodePort service by using “<NodeIP>:<NodePort>”.
3. **LoadBalancer.** This service type exposes the service externally using the load balancer of your cloud provider. The external load balancer routes to your NodePort and ClusterIP services, which are created automatically.
4. **ExternalName.** This type maps the service to the contents of the externalName field (e.g., foo.bar.example.com). It does this by returning a value for the CNAME record.

Before moving forward, let’s go over the role of kube-proxy. Kube-proxy implements a form of virtual IP for services for all types other than ExternalName. To achieve this, you can set three possible modes:

- **Proxy-mode: userspace.** In this mode, kube-proxy keeps an eye on the Kubernetes master, watching for services and endpoints objects that get added or removed. For each service, the mode opens a random port on your local node. Any connections to this “proxy port” are proxied to a service’s backend pods and reported in the endpoints.
- **Proxy-mode: iptables.** When this mode is on, kube-proxy continues to watch the Kubernetes master for added or removed services and endpoint objects.
    - For each service, unlike in userspace, this mode installs iptables rules in order to capture traffic to the service’s clusterIP (virtual) and port, and then redirects that traffic to a service backend set.
    - For each endpoints object, the mode installs iptables rules to select a random (by default) backend pod.
- **Proxy-mode: ipvs**. In this mode, kube-proxy watches the services and endpoints and calls netlink interface in order to create appropriate ipvs rules. Then, to ensure ipvs status is consistent with its expectations, the mode periodically syncs ipvs rules with services and endpoints. When a service is accessed, traffic gets redirected to a backend pod.

How to discover a Kubernetes service

In Kubernetes, there are two ways to discover a service:

- **DNS.** In this recommended method, the DNS server is added

to the cluster in order to watch the Kubernetes API create DNS record sets for each new service. When DNS is enabled throughout the cluster, all pods should be able to automatically perform name resolution of services.
- **ENV Var.** In this discovery method, a pod runs on a node, so the kubelet adds environment variables for each active service.

Headless services

When you neither need nor want load-balancing and a single service IP, create a headless service by specifying “none” for the cluster IP (.spec.clusterIP). There are two options:

- **Headless service with selectors**. With a headless service that defines selectors, the endpoints controller creates endpoint records in the API, modifying the DNS configuration to return A records (addresses) that point to the pods that back the service.
- **Headless service without selectors.** Headless services don’t define selectors, so the endpoints controller does not create endpoint records. However, the DNS system configures one of the following:
    - For service type ExternalName, CNAME records
    - For the other service types, a record for any endpoints that share names with the service

How to create a service

Creating a service is better understood when we use a simple example. We’ll run a “Hello World” application and use a deployment kind to create the app. Once deployment is up and running, we can create a service, using type ClusterIP, for our app.

First, let’s create a deployment running “kubectl run hello-world –replicas=2 –labels=”run=load-balancer-example”

–image=gcr.io/google-samples/node-hello:1.0 –port=8080”. Without going into too much detail, this command creates a deployment with two replicas of our application.

Next, run “kubectl get deployment hello-world” so see that the deployment is running. Now we can check the replicaset and pods that the deployment created.

```
$ kubectl get deployments hello-world
NAME          DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
hello-world   2         2         2            2           56s
```

With the applications running, we want to access one. So, let’s create a ClusterIP type of service. We can:

- Create a yaml manifest for the service and apply it, or
- Use the “kubectl expose” command, which is the easier option. This expose command creates a service without creating a yaml file.

```
$ kubectl expose deployment hello-world --type=ClusterIP --name=example-service
service "example-service" exposed
```

Here, we’ll create a service called example-service with type ClusterIP.

To access our application, run “kubectl get service example-service” to get our port number. Then, run a special command called port-forward. Because our service type is Cluster IP, which can only be accessed from within the cluster, we must access our app by forwarding the port to a local port.

We can use other types, like “LoadBalanacer”, which will create a LB in AWS or GCP, then we can access the app using the DNS address given to the LB with our port number.

```
$ kubectl get service example-service
NAME              TYPE        CLUSTER-IP      EXTERNAL-IP
PORT(S)    AGE
example-service   ClusterIP   100.20.149.98   <none>
```

```
8080/TCP   1h
```

```
$ kubectl port-forward service/example-service 8080:8080
Forwarding from 127.0.0.1:8080 -> 8080
```

Now we can browse http://localhost:8080 from our workstation and we should see:

```
Hello Kubernetes!
```