HOW KUBERNETES SERVICES WORK

A Kubernetes service is a logical collection of pods in a Kubernetes cluster. We can define a K8s service as an abstract way to load balance across the pods and expose an application deployed on a set of Pods. Moreover, using the inbuilt service mechanism in Kubernetes eliminates the need for implementing a separate service discovery mechanism.

In this article, we will discuss the structure of a Kubernetes service and how to utilize it.

(This article is part of our Kubernetes Guide. Use the right-hand menu to navigate.)

What are Kubernetes services?

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

Pods in a Kubernetes deployment are regularly created and destroyed, causing their IP addresses to change constantly. It will create discoverability issues for the deployed, application making it difficult for the application frontend to identify which pods to connect.

This is where the strengths of Kubernetes services come into play: services keep track of the changes in IP addresses and DNS names of the pods and expose them to the end-user as a single IP or DNS.

Kubernetes services utilize selectors to target a set of pods:

- **For native Kubernetes applications** (which use Kubernetes APIs for service 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.

Attributes of a Kubernetes service

Here are general attributes of a service:

- A service is assigned an IP address ("cluster IP"), which the service proxies use.
- A service can map an incoming port to any targetPort. (By default, the targetPort is set to the same value of the port field, and it 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

Now let's see how services work.

Defining a Kubernetes service

As mentioned above, we can define Kubernetes services with or without selectors—let's do both!

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
```

Kubernetes will automatically assign an IP address ("Cluster IP") which will then be used by service proxies to route the traffic. The controller for the selector will consistently monitor for Pods matching the defined label.

Some applications will require multiple ports to be exposed via the service. Kubernetes facilitates this using multi-port services where a user can define multiple ports in a single service object. When defining a multi-port service, the main requirement is to provide names for each port exposed so

that they are unambiguous. These port names can:

- Only contain alphanumeric characters and the dash symbol
- Should start and end with an alphanumeric character.

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
```

Defining a service without a selector

When defining a service without a selector, you need to manually map the service to the corresponding IP address and port by adding an endpoints object. The reason is that the endpoint objects are not created automatically like with a selector since Kubernetes does not know to which Pods the service should be connected.

Service:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

Endpoint:

```
apiVersion: v1
kind: Endpoints
metadata:
  name: nginx-service
subsets:
```

```
  - addresses:
      - ip: 192.10.0.1
    ports:
      - port: 8080
```

When defining a service and the endpoint, the main consideration is that both the name of the service and the endpoint must be an exact match.

Some use cases for services without selectors include:

- Connecting to a different service on another Namespace or another cluster
- Communicating with external services, data migration, testing services, deployments, etc.

Creating a Kubernetes service

Now that we know the basic structure of a Kubernetes service, let's use a practical example to better understand how to create a service. First, we'll create a small deployment with four replicas of an apache web server.

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
```

Result:

```
❯ kubectl apply -f apache-deployment.yaml
deployment.apps/apache-deployment created
```

Check the status of the deployment.

```
kubectl get deployment apache-deployment
```

Result:

```
❯ kubectl get deployment apache-deployment
NAME                READY   UP-TO-DATE   AVAILABLE   AGE
apache-deployment   4/4     4            4           3m30s
```

Now we have deployed the application to the apache server.

Next, we will create a service to access our deployed application. There are two methods to do that, either:

1. Create a YAML manifest for a service
2. Use the “kubectl expose” command

The expose command allows users to create the service from the command line directly.

We can use the following command to expose the "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
```

Result:

```
❯ kubectl expose deployment apache-deployment --name=apache-http-service --type=ClusterIP
--port=8080 --target-port=8080
service/apache-http-service exposed
```

Check whether the service is created correctly.

```
kubectl get service apache-http-service
```

Result:

```
❯ kubectl get service apache-http-service
NAME                 TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
apache-http-service   ClusterIP   10.101.0.133   <none>        8080/TCP   94s
```

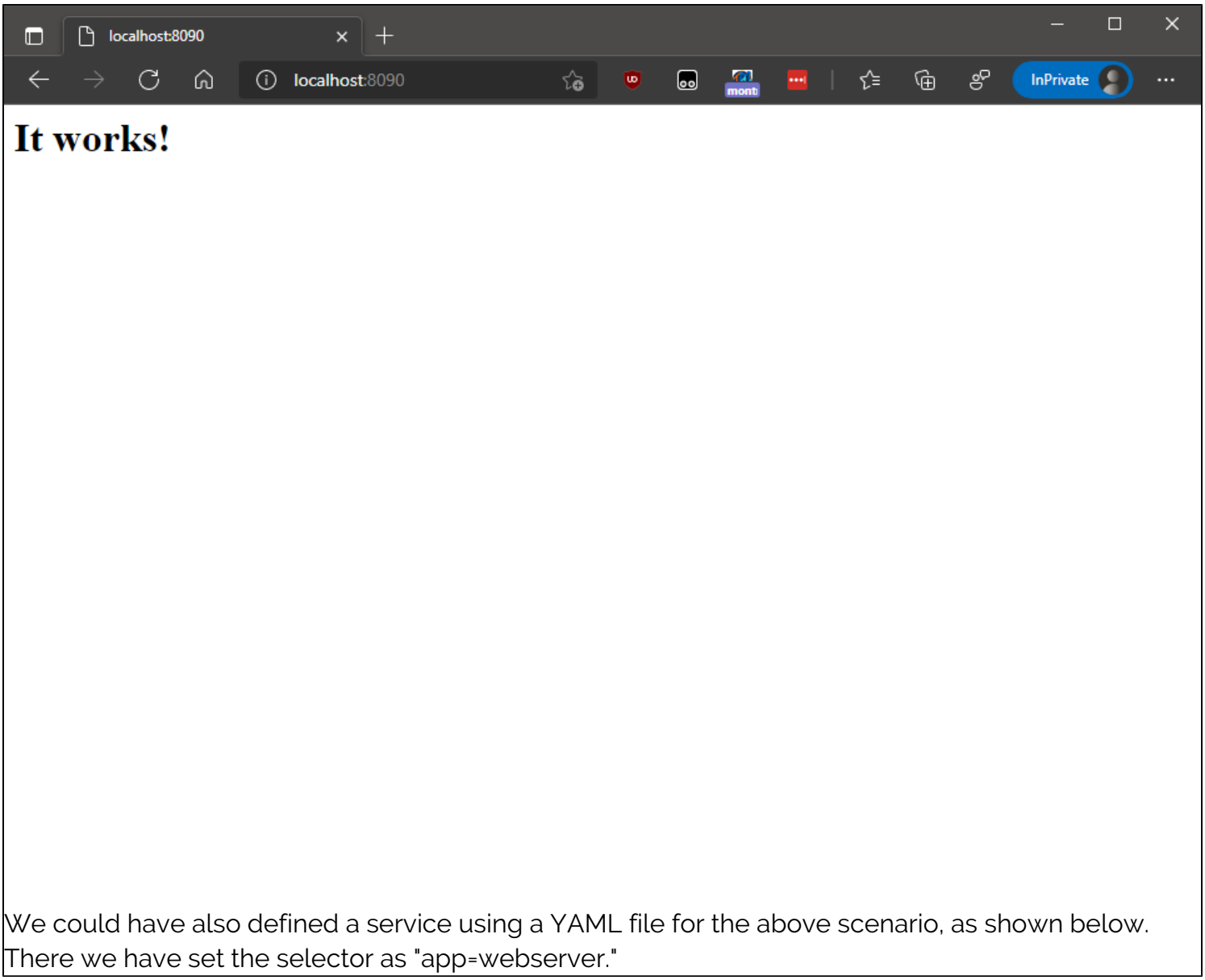
Now we need to forward the traffic to the service we created, and we can use the "kubectl port-forward" command for that. We will be directing traffic from host port 8090 to service port 8080.

```
kubectl port-forward service/apache-http-service 8090:8080
```

Result:

```
> kubectl port-forward service/apache-http-service 8090:8080
Forwarding from 127.0.0.1:8090 → 8080
Forwarding from [::1]:8090 → 8080
```

When we navigate to the "http://localhost:8090", we will be directed to an Nginx landing page like the following.



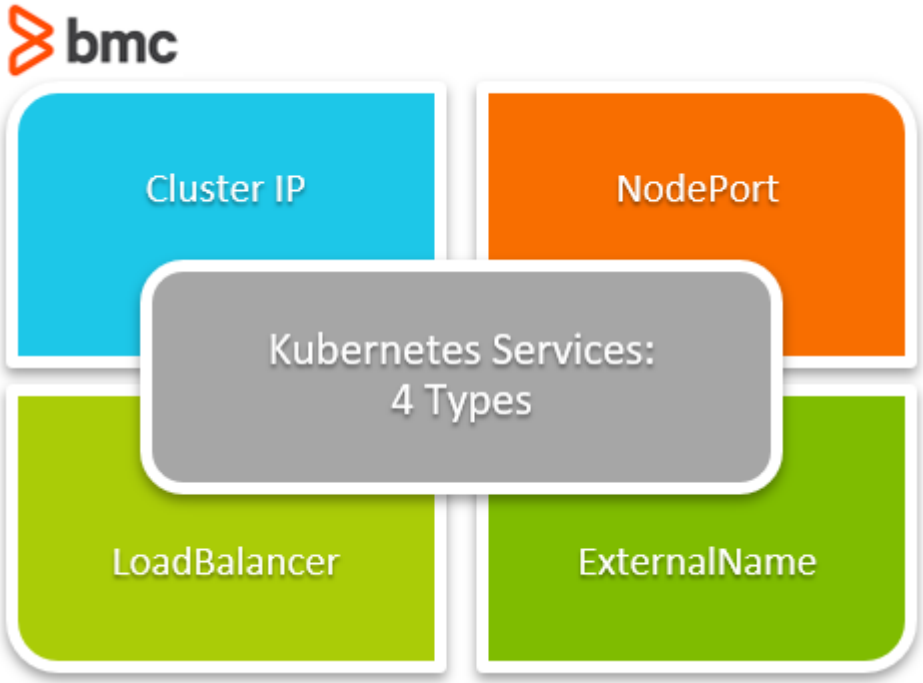
We could have also defined a service using a YAML file for the above scenario, as shown below. There we have set the selector as "app=webserver."

```
apiVersion: v1
kind: Service
metadata:
  name: apache-http-service
spec:
  selector:
    app: webserver
  ports:
    - protocol: TCP
      port: 8080
```

```
    targetPort: 8080
```

Types of Kubernetes services

There are four types of Kubernetes services:



ClusterIP

This is the default type that exposes the service on an internal IP of the cluster. These services are only accessible within the cluster. So, users need to implement port forwarding or a proxy to expose a ClusterIP to a wider ingress of traffic.

NodePort

A NodePort service exposes the service on the IP of each node at a static port. A ClusterIP service is created automatically to route the traffic to the NordPort service. Users can communicate with the service from the outside by requesting <NodeIP>:<NodePort>

LoadBalancer

This is the preferred solution to expose the cluster to the wider internet. The LoadBalancer type of service will create a load balancer (load balancer type depends on the cloud provider) and expose the service externally.

It will also automatically create ClusterIP and NodePort services and route traffic accordingly.

ExternalName

This type of service maps the service to the contents of the externalName field (Ex: app.test.com). It does this by returning a value for the CNAME record.

Service proxy implementations

The kube-proxy process implements a form of virtual IPs for all the service types, excluding the ExternalName type. Using proxying for services enables users to mitigate some DNS resolution issues.

For example, if a DNS record is expired, there might be instances where previous DNS lookups are still cached with the expired record, leading to resolution issues. Similarly, DNS records not getting properly propagated can also lead to resolution issues.

The following three methods can be used by the kube-proxy for routing traffic:

- Proxy-mode: userspace
- Proxy-mode: iptables
- Proxy-mode: ipvs

Proxy-mode: userspace

In this mode, kube-proxy monitors the Kubernetes master, and whenever a service or an endpoint gets created or removed, it allocates or deallocates a random port on the local node.

All the connections to this "proxy port" are proxied to the service and the necessary endpoints.

Proxy-mode: iptables

This mode also monitors the Kubernetes master for services and endpoints. However, it will also install iptable rules to capture traffic to the ClusterIp of the service and port and then redirect the traffic.

Additionally, it will install iptable rules for endpoint objects to select the backend Pod, while the default behavior is to select a random backend. (Pod)

Proxy-mode: ipvs

In this mode, the kube-proxy watches the services and endpoints and then calls the Netlink interface to create appropriate ipvs (IP Virtual Server) rules. This mode will periodically sync the ipvs rules with the services and endpoint to keep everything up to date.

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.

Summing up K8s services

With that, we've covered all the basics of Kubernetes services. We now know about services, including the different types, service discovery, and service proxy usages. Services are an integral part of Kubernetes that provide means to facilitate communication between pods, external services, and users.

Related reading

- BMC DevOps Blog
- Kubernetes Guide, with 20+ articles and tutorials
- Kubernetes Deployments Fully Explained
- The State of Kubernetes Today
- Kubernetes vs Docker Swarm: Comparing Container Orchestration Tools
- The State of Containers Today: A Report Summary