services

In Kubernetes, a *Service* is an abstraction that defines a logical set of Pods and a policy to access them. Services allow you to expose your application, enabling it to communicate with other parts of your Kubernetes cluster or with external clients.

Types of Services in Kubernetes

1. **ClusterIP**: Exposes the Service on an internal IP in the cluster. This makes the Service accessible only within the cluster. This is the default type.
2. **NodePort**: Exposes the Service on each node's IP at a static port (the NodePort). A ClusterIP Service is automatically created, and the NodePort Service route traffic to the ClusterIP Service. This makes the Service accessible from outside the cluster by requesting `<NodeIP>:<NodePort>`.
3. **LoadBalancer**: Exposes the Service externally using a cloud provider's load balancer. The cloud provider allocates a fixed, external IP address that sends traffic to the backend Pods.
4. **ExternalName**: Maps the Service to a DNS name. Instead of forwarding traffic, it returns a CNAME record with the value set by `externalName`.

Example of a ClusterIP Service

Imagine you have a simple application with Pods running an Nginx server. You want other applications within the Kubernetes cluster to access the Nginx server by using a Service.

Deploy the Nginx Pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  labels:
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx:latest
    ports:
    - containerPort: 80
```

1.

Create a ClusterIP Service to Expose Nginx:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
  - protocol: TCP
    port: 80        # Port on the Service
    targetPort: 80   # Port on the Pod
  type: ClusterIP    # Default type
```

2. ## Explanation of the Service

- **Selector**: This matches the Pods with the label `app: nginx`. The Service will route traffic to Pods with this label.
- **Port**: This is the port on the Service that other applications within the cluster will use.
- **TargetPort**: This is the port on the Pod to which traffic should be forwarded. In this case, it's the port where the Nginx server is listening.

Accessing the Service

Once you've created the Service, other Pods in the same Kubernetes cluster can access the Nginx server by using the Service name `nginx-service` on port `80`. Kubernetes DNS will automatically resolve the Service name (`nginx-service`) to the internal IP of the Service, allowing other Pods to access it easily.

Testing the Service

If you create another Pod, say, `busybox`, you can test the connectivity to `nginx-service`:

```
kubectl run busybox --image=busybox --restart=Never --command -- sleep
3600
```

```
kubectl exec -it busybox -- wget -O- nginx-service
```

This would fetch the default Nginx page from the `nginx-service`.

NodePort Service

A *NodePort* Service in Kubernetes exposes the application on each node's IP at a static port, allowing external access to the application via `<NodeIP>:<NodePort>`. This is useful when you want to access your application from outside the cluster without a LoadBalancer.

Example of a NodePort Service

Let's set up an example where we have a simple Nginx application that we want to expose using a NodePort service.

Step 1: Create a Deployment for Nginx

Deploying an Nginx Deployment with multiple replicas gives us a set of Pods to work with.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

Step 2: Create a NodePort Service to Expose Nginx

Here, we'll create a NodePort service to expose our Nginx Pods on a static port, accessible from outside the cluster.

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-nodeport-service
spec:
  selector:
    app: nginx
  ports:
  - protocol: TCP
    port: 80             # Port on the Service
    targetPort: 80       # Port on the Pod
    nodePort: 30007      # Port on each node where the Service is
exposed
  type: NodePort         # Type of Service
```

Explanation of the Service

- **Selector**: This targets Pods with the label `app: nginx`, so traffic to the Service is routed to these Pods.
- **Port**: The port that other services within the cluster would use to access this service (80).
- **TargetPort**: The port on the container (Pod) that the service forwards traffic to (80, where Nginx is listening).
- **NodePort**: The static port (30007) where the service is accessible on each node's IP.
- **Type**: Setting it to `NodePort` exposes the service on each node's IP address.

  Note: If `nodePort` is not specified, Kubernetes will automatically allocate a random port within the range 30000-32767.

Accessing the NodePort Service

After deploying this NodePort Service, you can access the Nginx application from outside the Kubernetes cluster by visiting `http://<NodeIP>:30007`.

Testing the NodePort Service

Get the IP address of any node in your Kubernetes cluster (if you're using a single-node setup, use the IP of that node).

```
kubectl get nodes -o wide
```

1. Access the Nginx application using the node's IP address and the `NodePort`.
   shell
   Copy code
   `curl http://<NodeIP>:30007`
2. You should see the default Nginx welcome page, confirming that the application is accessible externally through the NodePort.

LoadBalancer Service

A *LoadBalancer* Service in Kubernetes allows you to expose your application externally through a cloud provider's load balancer. When using a LoadBalancer Service, Kubernetes automatically provisions an external load balancer (e.g., from AWS, GCP, or Azure) that routes external traffic to the Service, and subsequently, to the Pods within the cluster.

Example of a LoadBalancer Service

Let's set up an example where we deploy an Nginx application and expose it using a LoadBalancer service.

Step 1: Create a Deployment for Nginx

First, we create a Deployment for Nginx to run our application in multiple Pods.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
```

```
  metadata:
    labels:
      app: nginx
  spec:
    containers:
    - name: nginx
      image: nginx:latest
      ports:
      - containerPort: 80
```

Step 2: Create a LoadBalancer Service to Expose Nginx

Now, we create a LoadBalancer Service to expose the Nginx application externally through a cloud provider's load balancer.

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-loadbalancer-service
spec:
  selector:
    app: nginx
  ports:
  - protocol: TCP
    port: 80             # Port on the Service
    targetPort: 80       # Port on the Pod
  type: LoadBalancer     # Type of Service
```

Explanation of the Service

- **Selector**: This targets Pods with the label `app: nginx`, ensuring that traffic is routed to the Nginx Pods.
- **Port**: The port that will be used on the Service (80).
- **TargetPort**: The port on the container (Pod) where Nginx is listening (80).
- **Type**: Setting it to `LoadBalancer` tells Kubernetes to provision an external load balancer.

Note: LoadBalancer Services typically work only on cloud providers that support load balancers (e.g., AWS, GCP, Azure). Running this on a local Kubernetes cluster

(like Minikube) may not create a load balancer unless you enable Minikube's load balancer addon or use a similar service.

Accessing the LoadBalancer Service

Once this Service is created, the cloud provider will automatically provision an external load balancer and assign it a public IP address. You can check the external IP by running:

```
kubectl get service nginx-loadbalancer-service
```

The output will show an `EXTERNAL-IP` once the load balancer is provisioned. It may take a minute or two.

You can now access the Nginx application by navigating to `http://<EXTERNAL-IP>`

Testing the LoadBalancer Service

To confirm the Service is reachable, run:

```
curl http://<EXTERNAL-IP>
```

This should return the default Nginx welcome page, confirming that the application is accessible externally through the cloud load balancer.

ExternalName Service

An *ExternalName* Service in Kubernetes is a special type of Service that acts as an alias for an external DNS name. Instead of routing traffic to a set of Pods, an ExternalName Service maps the Service name to a CNAME record (DNS alias), allowing other Services and Pods within the Kubernetes cluster to access external services by using an internal Kubernetes Service name.

This is useful when you want to refer to an external resource (like a database or an API hosted outside the Kubernetes cluster) with a Kubernetes Service name.

Example of an ExternalName Service

Imagine you have an external database hosted at `db.example.com`. You want to make this database accessible to applications within your Kubernetes cluster using the Service name `external-db`.

Step 1: Create the ExternalName Service

Here's the YAML configuration for an ExternalName Service that maps the Service name `external-db` to the external DNS name `db.example.com`.

```
apiVersion: v1
kind: Service
metadata:
  name: external-db
spec:
  type: ExternalName
  externalName: db.example.com
```

Explanation of the Service

- **Type**: The type is set to `ExternalName`, indicating that this Service will map to an external DNS name rather than a set of Pods.
- **externalName**: Specifies the external DNS name (`db.example.com`) that this Service will alias.

How It Works

Once this Service is created, any Pods within the Kubernetes cluster that attempt to reach `external-db` will be redirected to `db.example.com`. The Kubernetes DNS automatically resolves `external-db` to `db.example.com`.

Testing the ExternalName Service

If you create a Pod, like `busybox`, you can use it to test connectivity to the external service through the ExternalName Service.

Start a `busybox` pod in your cluster:

```
kubectl run busybox --image=busybox --restart=Never --command -- sleep 3600
```

1. Use the `nslookup` or `curl` command within the `busybox` pod to verify the DNS mapping:

   ```
   kubectl exec -it busybox -- nslookup external-db
   ```

2. You should see an output indicating that `external-db` resolves to `db.example.com`. You can also try:

   ```
   kubectl exec -it busybox -- curl external-db
   ```

> **Note**: This type of Service only provides DNS resolution; it doesn't handle traffic routing. For example, if `db.example.com` is listening on a specific port (e.g., `3306` for MySQL), the application using the `external-db` Service would need to connect to that port explicitly.