Kubernetes Services Explained

Understand now Kubernetes Services ensure reliable communication in modern applications

What is a Kubernetes Service?

A Kubernetes Service provides a stable IP and DNS name to access a dynamic set of pods. It abstracts the logic of connecting to changing pods behind a single endpoint.

Why is it Used?

- Ensures reliable access to applications even when pods change.
- Supports load balancing among pod replicas.
- Enables inter-service communication in microservices architecture

When is it Used?

- When deploying a replicated application.
- When exposing an app inside or outside the cluster.
- When routing traffic to different backend pods

Types of Services

- **ClusterIP** Exposess the service on an internal cluster IP.
- **NodePort** Exposes the service on a static port sucporle
- **LoadBalancer** Creates an external load balancer (if suppt

Labels and Selectors

Labels attach metadata to pods. Selectors define which pods a service targets based on labels.

What is a Kubernetes Service?

A **Kubernetes Service** is an **abstraction** that defines a logical set of Pods and a policy by which to access them. It provides a **stable endpoint (IP address and DNS)** to access dynamically changing pod IPs.

Why is a Service Needed?

Pods are **ephemeral.** Their IP addresses change when:

1. A pod is restarted (due to failure or rolling update)

2. The node fails

3. Auto-scaling or auto-healing kicks in

Without a service:

1. **You can't reliably access your app**, because the IP will change

2. Manual IP updates or tight coupling to pod IPs lead to brittle deployments

Real-World Scenario (DevOps vs Dev Argument)

Situation:

- Developer deploys an app.

- DevOps has set up auto-healing in the deployment (replicaSets manage pod health).

- Pod crashes and a new pod is auto-created—but **with a new IP address.**

- Dev can't access the app and blames DevOps.

- DevOps replies: “The pod is up and running!”

- Issue? No **Kubernetes Service** in place—so the frontend/backend is still trying to access the old pod IP.

Solution: Kubernetes Service

- DevOps creates a **Service**, which **maintains a consistent virtual IP (ClusterIP)**.

- No matter which pod is alive, the service **routes the request** to the right pod.

- Developer now accesses the service IP or DNS, **not the pod directly**.

Diagram: Without vs With Service

```
[WITHOUT SERVICE]
Client --> Pod (10.1.1.3)
           ↳ Pod crashes → New Pod (10.1.1.8)
           ↳ Client still hits 10.1.1.3 → Failure

[WITH SERVICE]
Client --> Service (10.96.0.1) --> Routes to Current Pod (e.g., 10.1.1.8)
           ↳ Pod crashes → New Pod (10.1.1.9)
           ↳ Service updates routing → Still works
```

✅ Problems Solved by Services

Pod IP changes → Solved with stable access via service

Scaling → Service load balances across all pods

Discoverability → Auto DNS registration for services

Inter-pod communication → Uses label selectors to match pods

1. Load Balancing in Kubernetes Services

What it does:

Automatically **distributes traffic** across multiple pods behind the service.

Example:

```
apiVersion: v1
kind: Service
metadata:
  name: my-app-service
spec:
  selector:
    app: my-app
  ports:
    - port: 80
      targetPort: 8080
  type: ClusterIP
```

Suppose 3 pods are running with label app: my-app

Incoming request to the service goes to **any one** of the 3 pods

---

2. Service Discovery

What it is:

Automatically allows services to **find and communicate** with each other using **DNS**.

Kubernetes runs a DNS service (CoreDNS)

Each service gets a name like: my-app-service.default.svc.cluster.local

Other pods/services can call it using that DNS name

When it's used:

Microservices calling each other (e.g., frontend → backend → database)

Labels and Selectors

Labels: Key-value pairs attached to pods (or any K8s resource)
Selectors: How a service **finds the pods** it should route traffic to

Example:

```
# Deployment
metadata:
  labels:
    app: my-app

# Service
spec:
  selector:
    app: my-app
```

The service automatically routes to pods with label app: my-app.

---

3. Exposing Services to External World

To make a Kubernetes service accessible **outside** the cluster, use:

Types of Services

1. ClusterIP (default)

Usage: Internal communication only (within the cluster)

Example:

```
type: ClusterIP
```

Can be accessed from inside the cluster using curl http://my-service.

---

2. NodePort

Usage: Exposes service on each Node's IP at a static port (external access via <NodeIP>:<NodePort>)

Example:

```
type: NodePort
ports:
  - port: 80
    targetPort: 8080
    nodePort: 30007
```

You can access this using:

http://<NodeIP>:30007

---

3. LoadBalancer

Usage: Provisions an external load balancer (supported in cloud platforms like AWS, GCP, Azure)

Example:

```
type: LoadBalancer
```

Automatically assigns a public IP, and external clients can access it.

Comparison Table

Service Type	Internal/External	Use Case	Access Method
ClusterIP	Internal only	Microservices within the cluster	DNS or Cluster IP
NodePort	External	Basic access to service from outside	NodeIP:NodePort
LoadBalancer	External	Scalable, production-ready access	Public IP (cloud provider assigned)

Key Advantages of Services

1. **Consistent access to dynamic pods**

2. **Load balancing** between multiple replicas

3. **Automatic DNS registration** and discovery

4. **Isolation** (internal vs external traffic control)

5. **Simplified microservices communication**

🧾 Kubernetes Services Command Cheat Sheet

1. View All Services

```
kubectl get svc
kubectl get services
kubectl get svc -n <namespace>
```

2. Describe a Specific Service

```
kubectl describe svc <service-name>
kubectl describe svc <service-name> -n <namespace>
```

3. View Service Details in YAML Format

```
kubectl get svc <service-name> -o yaml
```

4. Create a Service (Imperatively)

```
# ClusterIP
kubectl expose deployment <deployment-name> --port=80 --target-port=8080 --name=<service-name> --type=ClusterIP
```

```
# NodePort
kubectl expose deployment <deployment-name> --port=80 --target-port=8080 --name=<service-name> --type=NodePort
```

```
# LoadBalancer
kubectl expose deployment <deployment-name> --port=80 --target-port=8080 --name=<service-name> --type=LoadBalancer
```

5. Edit a Service

```
kubectl edit svc <service-name>
```

6. Delete a Service

```
kubectl delete svc <service-name>
```

7. Apply Service from YAML

```
kubectl apply -f service.yaml
```

8. Port Forward to a Service

Useful for debugging without exposing externally.

```
kubectl port-forward svc/<service-name> 8080:80
```

9. Access Service via DNS (within a pod)

```
curl http://<service-name>.<namespace>.svc.cluster.local
```

10. Get External IP of LoadBalancer

```
kubectl get svc <service-name>
```

Look under the `EXTERNAL-IP` column.

Useful Debugging Tips

Check Endpoints (What pods a service routes to)

```
kubectl get endpoints <service-name>
```

View Labels on a Pod

```
kubectl get pods --show-labels
```

Manually Test Connectivity Between Pods/Services

```
kubectl exec -it <pod-name> -- curl http://<service-name>:<port>
```
