FENIL GAJJAR

KUBERNETES

DAILY TASKS

KUBERNETES SERVICES

- COMPREHENSIVE GUIDE
- THEORY + PRACTICAL TASKS
- REAL TIME SCENARIO TASKS
- FOLLOW FOR MORE TASKS

CONTACT US

fenilgajjar.devops@gmail.com

🌐 Unlocking Kubernetes Services: The Key to Seamless Pod Communication! 🚀

🚀 Welcome to the Next Chapter in Our Kubernetes Journey – Kubernetes Services! 🎯

First of all, I want to express my **heartfelt gratitude** for the overwhelming support on my previous posts in this Kubernetes series. Your engagement, feedback, and enthusiasm make this journey even more exciting! 💙

So far, we've explored Kubernetes Pods – the fundamental building blocks of applications running inside Kubernetes. But have you ever wondered **how these Pods communicate with each other efficiently, even as they get replaced, rescheduled, or scaled?** 🤔 That's exactly where **Kubernetes Services** come into play!

In this post, we will **unlock the power of Kubernetes Services** – understanding how they enable seamless networking, load balancing, and service discovery across dynamic environments. Whether you're a beginner or an experienced Kubernetes practitioner, mastering Services is essential to deploying reliable and scalable applications.

🔥 **Stay tuned, keep supporting, and let's continue this learning journey together!** More **daily Kubernetes tasks** with real-world practical insights are on the way. 🚀

💡 **Follow me to stay updated with hands-on Kubernetes content every day!** Let's build, learn, and grow together! 💪

Kubernetes Services

1️⃣ What is a Kubernetes Service?

A **Service** in Kubernetes is an abstraction that provides a **stable network endpoint** to connect a set of pods. Since pods in Kubernetes are **ephemeral** (they can be created, deleted, or replaced dynamically), their IP addresses are not fixed. A Service ensures that applications can communicate with pods reliably, even if their IPs change.

🛑 Problem Without Services:

- Each pod gets a **new IP** when restarted or rescheduled.
- Other pods **cannot reliably communicate** with a specific pod.
- There is **no built-in mechanism** for load balancing traffic across multiple pod replicas.

✅ Solution: Kubernetes Services

- **Provides a stable, unchanging IP & DNS name** for a group of pods.

- **Automatically routes requests** to healthy pods using selectors.
- **Load balances** traffic among multiple pod replicas.
- **Exposes applications** within the cluster or to the external world.

Why Do We Need Services in Kubernetes?

In Kubernetes, **Pods are ephemeral**—they can be created, deleted, or rescheduled dynamically. Since each pod gets a unique IP address that changes when it restarts, this creates challenges in communication. This is where **Kubernetes Services** come into play.

1️⃣ Problem Without Services:

- **Pods have dynamic IPs** – When a pod restarts or scales, its IP changes.
- **No built-in discovery** – Other pods can't directly find or connect to a specific pod reliably.
- **Load balancing is needed** – If multiple pod replicas exist, traffic must be distributed properly.

2️⃣ How Services Solve These Problems:

A **Kubernetes Service** provides a **stable IP and DNS name** for a group of pods, making them accessible reliably.

✅ **Key Benefits of Services:**

1. **Stable Networking** – Services ensure a consistent way to communicate with pods, even if pod IPs change.
2. **Automatic Load Balancing** – Distributes traffic across multiple pod replicas.
3. **Service Discovery** – Kubernetes assigns a DNS name (`my-service.default.svc.cluster.local`) so other pods can find it easily.
4. **External Access** – Services allow external users to access applications running inside the cluster.

3️⃣ Types of Kubernetes Services

Service Type	Use Case

ClusterIP (Default)	Exposes the service **internally** within the cluster. Used for pod-to-pod communication.
NodePort	Exposes the service on a **static port on each node**. Used for basic external access.
LoadBalancer	Integrates with a cloud provider's **load balancer** (AWS, GCP, Azure) for external access.
ExternalName	Maps a service to an **external domain name** (e.g., `example.com`).

4 . How Services Work with DNS in Kubernetes

Kubernetes assigns **a DNS name to each service**, making communication simple.

For example:

- A pod can access the service using **`http://web-service`** instead of using pod IPs.
- The actual DNS name is **`web-service.default.svc.cluster.local`**, making it discoverable inside the cluster.

ClusterIP Service in Kubernetes

1️⃣What is a ClusterIP Service?

A **ClusterIP** service in Kubernetes is the **default type of service** that enables internal communication between different pods within a Kubernetes cluster. It assigns a **stable internal IP** that is accessible only from within the cluster but **not from the outside world**.

🛑 Why Do We Need ClusterIP?

- **Pods are ephemeral** – Their IP addresses change when restarted.
- **Pods need to communicate** with each other inside the cluster.
- **Stable Networking** – ClusterIP ensures a consistent way to access pods.

📌 **Use Case:** Internal microservices communication, backend-to-database connections, etc.

2️⃣How Does a ClusterIP Service Work?

1. A **Service** is created with type `ClusterIP` and a **label selector** to target specific pods.

2. Kubernetes assigns a **stable virtual IP (ClusterIP)** to the service.
3. Any pod inside the cluster can access the service using:
   - **ClusterIP**
   - **Service name (via DNS)**
4. Kubernetes **automatically load balances** traffic across the selected pods.

📌 **Important:** The ClusterIP is only accessible **inside the cluster**.

3️⃣ Example: Creating a ClusterIP Service

Step 1: Create a Deployment

Let's deploy an **Nginx-based web app** running in multiple pods.

```
apiVersion: apps/v1

kind: Deployment

metadata:

  name: web-app

spec:

  replicas: 3
```

```yaml
selector:
  matchLabels:
    app: web-app
template:
  metadata:
    labels:
      app: web-app
  spec:
    containers:
    - name: nginx
      image: nginx
      ports:
      - containerPort: 80
```

Step 2: Create a ClusterIP Service

Now, let's expose this deployment using a **ClusterIP service**.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: web-service
spec:
  selector:
    app: web-app  # Targets pods with this label
  ports:
    - protocol: TCP
      port: 80       # Port exposed by the service
      targetPort: 80 # Port inside the pod
  type: ClusterIP
```

4️⃣ Accessing a ClusterIP Service

Once the service is created, Kubernetes assigns it a **ClusterIP**, which can be seen using:

kubectl get services

📌 **Example Output:**

NAME TYPE CLUSTER-IP PORT(S) AGE

web-service ClusterIP 10.96.15.200 80/TCP 5m

Ways to Access the Service:

✅ **Using ClusterIP:**

curl http://10.96.15.200:80

✅ **Using Kubernetes DNS Name (Preferred):**

curl http://web-service

- DNS Name: `web-service.default.svc.cluster.local`
- Automatically resolves to `10.96.15.200` within the cluster.

5️⃣ How Kubernetes Handles ClusterIP Internally

Kubernetes uses **iptables** or **IPVS** to route traffic to the backend pods.

🔹 **Steps:**

1. A request to `http://web-service` is received.
2. The request is routed via `kube-proxy`.
3. It forwards traffic to one of the **healthy pods** running `web-app`.
4. **Load balancing** happens automatically.

📌 **Round-Robin Load Balancing:**

If `web-service` has 3 pods running, requests will be distributed evenly among them.

6️⃣ Real-World Use Cases of ClusterIP

✅ Microservices Communication

- A frontend service (`frontend-service`) calls a backend API (`backend-service`).
- Backend interacts with a database service (`db-service`).

✅ Database Access Inside Cluster

- A MySQL pod is exposed as `mysql-service`.
- Backend pods use `mysql-service:3306` to connect instead of direct pod IPs.

✅ Internal Caching Systems

- Redis or Memcached can be exposed via ClusterIP (`redis-service`).
- Other applications access it for caching.

7️⃣ ClusterIP vs. Other Service Types

Service Type	Accessibility	Use Case
ClusterIP (default)	Only inside the cluster	Internal communication (microservices, databases)
NodePort	Exposes on a fixed port on each node	Direct external access (debugging, testing)
LoadBalancer	Exposes via cloud load balancer	Production applications requiring public access
ExternalName	Maps to an external DNS	Connecting to external databases (AWS RDS, etc.)

📌 **Without ClusterIP, internal communication in Kubernetes would be unreliable!** 🚀

Real-Life Scenario: ClusterIP Service in Kubernetes

Scenario: E-commerce Application with Microservices

Imagine you are deploying an **e-commerce platform** in Kubernetes. The architecture consists of multiple microservices:

1. **Frontend Service** (React/Angular)
2. **Backend API Service** (Node.js/Java)
3. **Database Service** (MySQL)
4. **Redis Cache** (For faster lookups)

Since the backend, database, and caching system **only need to communicate internally**, we use **ClusterIP services** to expose them.

Step 1: Deploy Backend API Service

The backend API service is responsible for handling requests from the frontend and communicating with the database.

Backend Deployment

```yaml
apiVersion: apps/v1

kind: Deployment

metadata:

  name: backend-api

spec:

  replicas: 3

  selector:

    matchLabels:

      app: backend-api

  template:

    metadata:

      labels:

        app: backend-api

    spec:
```

```yaml
      containers:
      - name: backend
        image: mycompany/backend-service:latest
        ports:
        - containerPort: 8080
```

Backend ClusterIP Service

This exposes the backend service **only inside the cluster**.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: backend-service
spec:
  selector:
    app: backend-api
  ports:
```

```yaml
    - protocol: TCP
      port: 80
      targetPort: 8080
  type: ClusterIP
```

📌 **How It Works?**

- Kubernetes assigns a **ClusterIP** (e.g., `10.96.15.100`).
- Any pod inside the cluster can access it using `http://backend-service`.

Step 2: Deploy MySQL Database

Since the database should **only be accessible inside the cluster**, we use another **ClusterIP service**.

MySQL Deployment

```yaml
apiVersion: apps/v1
kind: Deployment
```

```yaml
metadata:

  name: mysql-db

spec:

  replicas: 1

  selector:

    matchLabels:

      app: mysql-db

  template:

    metadata:

      labels:

        app: mysql-db

    spec:

      containers:

      - name: mysql

        image: mysql:5.7

        env:
```

```yaml
        - name: MYSQL_ROOT_PASSWORD
          value: "mypassword"
        - name: MYSQL_DATABASE
          value: "ecommerce"
        ports:
        - containerPort: 3306
```

MySQL ClusterIP Service

```yaml
apiVersion: v1
kind: Service
metadata:
  name: mysql-service
spec:
  selector:
    app: mysql-db
```

```yaml
  ports:
    - protocol: TCP
      port: 3306
      targetPort: 3306
  type: ClusterIP
```

📌 **How It Works?**

- Backend API accesses MySQL using `mysql-service:3306` instead of a changing pod IP.
- The database is **protected from external access**.

Step 3: Deploy Redis Cache

The backend API needs **fast access** to Redis, so we create another **ClusterIP service**.

Redis Deployment

```yaml
apiVersion: apps/v1
```

```yaml
kind: Deployment
metadata:
  name: redis-cache
spec:
  replicas: 1
  selector:
    matchLabels:
      app: redis-cache
  template:
    metadata:
      labels:
        app: redis-cache
    spec:
      containers:
      - name: redis
        image: redis:latest
```

```
        ports:

        - containerPort: 6379
```

Redis ClusterIP Service

```
apiVersion: v1

kind: Service

metadata:

  name: redis-service

spec:

  selector:

    app: redis-cache

  ports:

    - protocol: TCP

      port: 6379

      targetPort: 6379

  type: ClusterIP
```

📌 **How It Works?**

- Backend API connects to `redis-service:6379`.
- The Redis service is **not accessible from outside the cluster**.

Step 4: Frontend Access

Since the **frontend needs to be accessible externally**, we use a **LoadBalancer or NodePort service** for it.

Frontend Service (LoadBalancer)

```
apiVersion: v1

kind: Service

metadata:

  name: frontend-service

spec:

  selector:
```

```yaml
    app: frontend
  ports:
    - protocol: TCP
      port: 80
      targetPort: 3000
  type: LoadBalancer
```

📌 **Frontend accesses services as:**

- `backend-service:80`
- `mysql-service:3306`
- `redis-service:6379`

NodePort Service in Kubernetes

What is a NodePort Service?

A **NodePort service** in Kubernetes exposes an application **on every node in the cluster** using a specific port (called a **NodePort**). This allows external traffic to access the service **directly via any node's IP and the assigned port**.

Unlike a **ClusterIP service**, which is only accessible within the cluster, a **NodePort service** makes the application accessible from outside the cluster.

How NodePort Works?

1. **Kubernetes assigns a static port (30000-32767)** to the service.
2. **Each node in the cluster listens on this port** and forwards traffic to the service.

Requests can be made using:

```
http://<NodeIP>:<NodePort>
```

3. where:
   - `<NodeIP>` is the IP of any node in the cluster.
   - `<NodePort>` is the assigned port (e.g., `30007`).

Example: Exposing a Web Application using NodePort

Let's say we have a **Node.js web application** running inside a Kubernetes pod. We want to expose it externally using **NodePort**.

Step 1: Create a Deployment

```
apiVersion: apps/v1

kind: Deployment

metadata:

  name: web-app

spec:

  replicas: 2

  selector:

    matchLabels:
```

```yaml
      app: web-app
  template:
    metadata:
      labels:
        app: web-app
    spec:
      containers:
      - name: web-app
        image: mycompany/web-app:latest
        ports:
        - containerPort: 8080
```

- This will create **two pods** running the web application on port `8080`.

Step 2: Expose the Application Using NodePort

```yaml
apiVersion: v1

kind: Service

metadata:

  name: web-app-service

spec:

  type: NodePort

  selector:

    app: web-app

  ports:

    - protocol: TCP

      port: 80        # ClusterIP Port

      targetPort: 8080 # The port in the container

      nodePort: 30007  # Exposed on each node (between 
30000-32767)
```

Step 3: Access the Application

After applying the YAML files, **Kubernetes will expose the service on every node's IP at port `30007`**.

You can access the web application using:

```
http://<NodeIP>:30007
```

- If you have multiple nodes, you can use **any** node's IP.
- Traffic coming to `30007` is forwarded to the **ClusterIP service**, which routes it to the **backend pods**.

Use Cases of NodePort

- **Exposing applications during development or testing**.
- **Accessing applications from outside the cluster** without using an Ingress.
- **Running internal services that need to be accessed without cloud load balancers**.

Limitations of NodePort

🚨 **Not ideal for production** because:

- The port range **(30000-32767)** is limited.
- Requires knowing **node IPs manually**.
- Not suitable for large-scale applications with multiple services.

For production, **LoadBalancer or Ingress** is preferred.

Real-Life Scenario of NodePort Service

Scenario: Exposing an Internal Web Dashboard for a Private Network

Imagine a company has a **Kubernetes cluster running on-premises** (not in the cloud) and hosts an **internal web dashboard** for monitoring system performance. The dashboard is only meant to be accessed by employees within the company's private network.

Since this is an **on-premises setup**, there is **no cloud provider** to provide a **LoadBalancer** service. Instead, the company decides to use **NodePort** to expose the dashboard to employees.

How NodePort Helps in This Scenario?

1. The dashboard application is running inside Kubernetes pods.
2. A **NodePort service** exposes the dashboard on port `32000`.

Employees within the **private network** can access it using:

```
http://<AnyNodeIP>:32000
```

3. The NodePort service forwards the request to the **correct pod** running the dashboard.

Example NodePort YAML for Internal Web Dashboard

Step 1: Create the Deployment

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: dashboard
spec:
  replicas: 2
  selector:
    matchLabels:
      app: dashboard
  template:
    metadata:
```

```yaml
      labels:
        app: dashboard
    spec:
      containers:
      - name: dashboard
        image: mycompany/dashboard:latest
        ports:
        - containerPort: 8080
```

- This deployment creates **two replicas** of the web dashboard application.
- The application runs on **port 8080** inside the container.

Step 2: Expose the Dashboard Using NodePort

```yaml
apiVersion: v1
kind: Service
metadata:
```

```yaml
  name: dashboard-service
spec:
  type: NodePort
  selector:
    app: dashboard
  ports:
    - protocol: TCP
      port: 80        # ClusterIP port
      targetPort: 8080 # Pod container port
      nodePort: 32000  # Exposed on all nodes
```

- The service listens on **port 32000** on **every node**.
- Requests received on `32000` are forwarded to `80`, which then routes to the **pods running the dashboard on port 8080**.

Step 3: Access the Dashboard

Employees can access the dashboard from their browsers by visiting:

```
http://<NodeIP>:32000
```

- `<NodeIP>` can be **any Kubernetes worker node IP** in the cluster.
- The request reaches `32000`, which **forwards traffic to the pods** running the dashboard.

Why Use NodePort in This Case?

✅ **No need for a cloud-based LoadBalancer** (as this is an on-premises cluster).
✅ **Internal employees can access it via private network without extra networking setup**.
✅ **Simple way to expose services within an organization**.

However, for **external access over the internet**, a **LoadBalancer or Ingress** is a better choice. 🚀

Default Kubernetes Service in the Cluster:

`kubernetes` Service

When a **Kubernetes cluster** is created, a default service named **`kubernetes`** is automatically created.

1️⃣What is the `kubernetes` Service?

- It is a **ClusterIP service** created by Kubernetes itself.
- It provides a **stable entry point** to the Kubernetes API server.
- It allows **internal cluster components** (like Pods) to communicate with the **Kubernetes API** without knowing its actual IP.

2️⃣Why Does Kubernetes Create This Service by Default?

The `kubernetes` service exists to: ✅ Allow Pods to **discover and interact with the Kubernetes API**.

✅ Enable internal components like **Controllers, Schedulers, and Custom Operators** to interact with the API server.

✅ Provide a **fixed internal DNS name**

(`kubernetes.default.svc.cluster.local`) that always resolves to the API server.

3️⃣ Details of the Default `kubernetes` Service

🟢 Checking the Service

You can verify the default service using:

```
kubectl get svc
```

This will show:

```
NAME         TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)
AGE
kubernetes   ClusterIP   10.96.0.1     <none>        443/TCP
10d
```

- The `kubernetes` service is of type **ClusterIP**.
- It has a fixed internal IP (e.g., `10.96.0.1`).
- It listens on port **443 (HTTPS)** for API requests.

4️⃣ How Pods Use This Service?

Every pod in the cluster can access the API using:

```
curl -k https://kubernetes.default.svc.cluster.local
```

OR

```
curl -k https://10.96.0.1
```

Example Use Case:

Suppose a pod wants to list all services in the cluster. It can send an API request like this:

```
curl -k -H "Authorization: Bearer $(cat
/var/run/secrets/kubernetes.io/serviceaccount/token)" \
     https://kubernetes.default.svc/api/v1/services
```

- This request reaches the Kubernetes API server via the **default service**.
- The API server processes the request and returns the list of services.

5️⃣How is This Service Created?

- Kubernetes **automatically creates** this service during cluster setup.
- It uses a **virtual IP (ClusterIP)** to route traffic to the API server.
- It is managed by **kube-apiserver**.

6️⃣Can You Delete the `kubernetes` Service?

🚨 **No, you should not delete it!**

- It is **critical** for internal Kubernetes operations.
- Many components (like controllers and operators) rely on it.
- Without it, **internal pods cannot communicate with the API server**.

However, if accidentally deleted, you can **recreate it manually**:

```
apiVersion: v1

kind: Service

metadata:

  name: kubernetes
```

```yaml
  namespace: default
spec:
  type: ClusterIP
  clusterIP: 10.96.0.1  # Ensure this matches your cluster's settings
  ports:
    - port: 443
      targetPort: 443
  selector:
    component: apiserver
```

Then, apply it using:

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

Default ClusterIP Kubernetes Service: Why Is It Useful?

In every Kubernetes cluster, a **default ClusterIP service** is automatically created to enable essential **internal communication** between cluster components. One such default service is **`kubernetes`**, which provides an internal DNS entry (`kubernetes.default.svc.cluster.local`) that allows pods to communicate with the Kubernetes API server.

🔹 Why Is the Default ClusterIP Service Useful?

1️⃣ Kubernetes API Access for Internal Components

📌 **Scenario:** Pods running in the cluster need to interact with the Kubernetes API for operations like scaling, retrieving configurations, or updating resource status.

✅ **How it helps:** The `kubernetes` service provides a stable internal IP (`ClusterIP`), so workloads can securely communicate with the API server without needing to hardcode its IP address.

💡 **Example:**

A pod using a service account to fetch information about available nodes:

```
curl -k -H "Authorization: Bearer $(cat
/var/run/secrets/kubernetes.io/serviceaccount/token)" \
https://kubernetes.default.svc/api/v1/nodes
```

🔹 **Why?** This allows applications to discover and interact with cluster resources dynamically.

2️⃣ Service Discovery for Cluster Components

📌 **Scenario:** Internal services (like CoreDNS, monitoring tools, or admission controllers) need to find and communicate with the Kubernetes API.

✅ **How it helps:** The `kubernetes` service ensures that no matter where a pod is scheduled, it can always find the API server at `kubernetes.default.svc`.

💡 **Example:**

- CoreDNS uses the Kubernetes API to resolve service names.
- Metrics servers query the API to collect resource usage data.

3️⃣ Secure Cluster-Level Communication

📌 **Scenario:** Applications or system services need **authenticated** and **secure** access to cluster data.

✅ **How it helps:** The service provides **TLS-secured access** to the API, enforcing RBAC (Role-Based Access Control) policies.

💡 **Example:**

- A monitoring agent in a pod queries Kubernetes for node health.
- A CI/CD pipeline interacts with Kubernetes to deploy applications.

4️⃣ Custom Controllers & Operators Need API Access

📌 **Scenario:** Custom controllers or operators running in the cluster need to **watch Kubernetes resources** and react to changes.

✅ **How it helps:** The service allows controllers to interact with the API server via the `watch` mechanism.

💡 **Example:**

A **custom Kubernetes operator** that automatically scales resources based on a database load.

```
apiVersion: batch/v1

kind: Job
```

```yaml
metadata:
  name: scale-job
spec:
  template:
    spec:
      containers:
      - name: scaler
        image: myorg/scaler:latest
        command: ["python", "/scripts/scale_resources.py"]
        env:
        - name: KUBERNETES_SERVICE_HOST
          value: "kubernetes.default.svc"
```

🔹 **Why?** The operator doesn’t need to worry about API server IP changes.

🔹 When Does This Service Become Critical?

✔️ When internal Kubernetes components (like CoreDNS, kube-proxy, or controllers) need to communicate with the API server.

✔️ When applications inside the cluster need to interact with Kubernetes dynamically.

✔️ When running CI/CD pipelines that deploy applications using `kubectl` or Kubernetes API calls.

✔️ When monitoring or logging tools like Prometheus and Fluentd need to query Kubernetes metadata.

LoadBalancer Service in Kubernetes (Using Kops)

🔹 What Is a LoadBalancer Service?

A **LoadBalancer** service in Kubernetes automatically provisions an **external load balancer** to expose your application to the internet or external networks. It integrates with cloud provider load balancers (e.g., AWS ELB, GCP Load Balancer, Azure Load Balancer).

If you are using **Kops** to create a Kubernetes cluster, it means you're likely running on **AWS**, and Kubernetes will provision an **AWS Elastic Load Balancer (ELB)** when you create a `LoadBalancer` service.

🔹 How LoadBalancer Works in a Kops Cluster (AWS Example)

1️⃣ You Create a LoadBalancer Service

- When you define a `LoadBalancer` service in Kubernetes, Kops automatically provisions an **AWS ELB**.
- The ELB forwards traffic to worker nodes where the app is running.

2️⃣ **Traffic Is Distributed to Worker Nodes**

- The ELB sends traffic to worker nodes on the specified ports.
- Kubernetes **`kube-proxy`** ensures that the traffic is forwarded to the correct pod.

3️⃣ **Pods Receive Requests via NodePort Service**

- Internally, Kubernetes **creates a NodePort service** and assigns a port on each worker node.
- The ELB routes external traffic to this **NodePort**, which then directs traffic to the correct pods.

🔹 Example: LoadBalancer Service in Kops (AWS ELB Example)

Here's how you define a `LoadBalancer` service for an Nginx deployment:

```
apiVersion: v1

kind: Service

metadata:
```

```yaml
  name: nginx-service
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-type: "nlb"
# Use Network Load Balancer (NLB)
spec:
  type: LoadBalancer
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80        # External Port
      targetPort: 8080  # Container Port
```

🔹 What Happens When You Apply This YAML?

1️⃣ Kubernetes Requests an AWS ELB

- Kops provisions an ELB in AWS.
- The ELB gets a **public IP / DNS name**, making the service accessible externally.

2️⃣ Traffic Flows Through the Load Balancer

- The ELB forwards traffic to the worker nodes on a dynamically assigned **NodePort**.
- Kubernetes then routes traffic to the correct pods using `kube-proxy`.

3️⃣ DNS Name for Access

AWS assigns a DNS name like:

```
a1b2c3d4e5-1234567890.us-east-1.elb.amazonaws.com
```

- You can access the service using this DNS name instead of manually configuring an ingress.

🔹 Key Features & Benefits

✔️ **Automatic External Access** – No need for manual configuration; Kubernetes provisions an ELB automatically.

✔️ **High Availability** – ELB distributes traffic across multiple nodes, ensuring redundancy.

✔️ **Integration with Cloud Providers** – Works seamlessly with AWS, GCP, Azure, etc.

✔️ **Supports Layer 4 (NLB) & Layer 7 (ALB) Load Balancers** – You can choose between TCP (NLB) or HTTP/HTTPS (ALB) balancing.

🔹 Use Cases for LoadBalancer Services

✔️ **Publicly Exposing Web Applications** – Websites, APIs, or services that need external access.

✔️ **Multi-Region or Multi-Zone Deployments** – When you want a **highly available** setup.

✔️ **Integrating with DNS & CDN** – Can be used with Route 53 and CloudFront for global load balancing.

✔️ **Secure HTTPS Access** – Supports TLS termination when using an Application Load Balancer (ALB).

🛠️ Troubleshooting & Best Practices

🔹 **Check the Created Load Balancer:**

```
kubectl get svc nginx-service
```

🔹 **Find External IP/DNS of Load Balancer:**

```
kubectl describe svc nginx-service
```

🔹 **Use Ingress for Advanced Routing** – Instead of directly using a LoadBalancer, consider **Ingress controllers** for better path-based routing and TLS termination.

🔹 **Enable Connection Draining** – Ensures smooth traffic shifting when pods are replaced or scaled.

🔹 Understanding the YAML Configuration

This YAML defines:

1️⃣ A Pod (`nginx`) running the `nginx` container.

2️⃣ A Service (`nginx-service`) exposing the `nginx` Pod.

🔹 Key Points in the Pod Definition

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app.kubernetes.io/name: proxy  # Label used for service selection
spec:
```

```yaml
  containers:
  - name: nginx
    image: nginx:stable
    ports:
      - containerPort: 80
        name: http-web-svc  # Named port inside the
container
```

- The **Pod is labeled** as `app.kubernetes.io/name: proxy`, which the Service uses to **select and route traffic**.
- The **container exposes port `80`**, but importantly, the port is given a **name** (`http-web-svc`).
- Named ports allow **flexibility** when referring to ports in Services.

🔹 **Key Points in the Service Definition**

```yaml
apiVersion: v1
kind: Service
```

```yaml
metadata:
  name: nginx-service
spec:
  selector:
    app.kubernetes.io/name: proxy  # Selects Pods with this label
  ports:
  - name: name-of-service-port  # Name for the Service port (optional)
    protocol: TCP
    port: 80   # Exposed port of the Service
    targetPort: http-web-svc  # Refers to the named containerPort inside Pods
```

- The Service **selects Pods** with the label `app.kubernetes.io/name: proxy`.

- The Service **exposes port `80` externally** and forwards traffic to the **named port (`http-web-svc`)** in the selected Pods.
- **Using named ports (`targetPort: http-web-svc`)** instead of hardcoded numbers provides **flexibility when changing ports** in the future.

🔹 Why Is This Flexible?

Scenario: Changing the Container's Port Without Breaking the Service

Imagine you **update the backend software** in the `nginx` container, and now it **listens on port 8080 instead of 80**.

Without named ports, you'd need to manually update the **`targetPort`** in the Service. However, with named ports, you **only update the Pod's containerPort**, and the Service automatically routes traffic correctly.

Updated Pod with New Port

```
apiVersion: v1

kind: Pod

metadata:

  name: nginx
```

```yaml
  labels:
    app.kubernetes.io/name: proxy
spec:
  containers:
  - name: nginx
    image: nginx:stable
    ports:
      - containerPort: 8080  # Changed from 80 to 8080
        name: http-web-svc  # Still the same name
```

- **No need to change the Service definition** because it still refers to `targetPort: http-web-svc`.
- Kubernetes automatically updates the routing based on the new Pod configuration.

🔹 Real-World Benefits of This Approach

✅ **Backward Compatibility:** Clients using the Service don't break when Pod ports change.

✅ **Seamless Rolling Updates:** You can roll out new versions of an app with different ports without breaking existing traffic.

✅ **Multi-Port Flexibility:** If different backend Pods expose different ports, a Service can still route to them using named ports.

Example: Multiple Pods with Different Ports in the Same Service

🔹 **Scenario**

Imagine a **microservices-based backend** where different versions of an application (e.g., v1 and v2) **run in separate Pods**, and each version **exposes a different port**. However, we want **a single Service** to route traffic to all of them **without breaking clients**.

🔹 **Solution Using Named Ports**

Instead of **hardcoding port numbers**, we use **named ports** in the Pods and reference them in the Service.

Step 1: Define Two Pods (v1 & v2) with Different Ports

```
apiVersion: v1

kind: Pod
```

```yaml
metadata:

  name: backend-v1

  labels:

    app.kubernetes.io/name: backend

    version: v1

spec:

  containers:

  - name: backend

    image: my-backend:v1

    ports:

      - containerPort: 8080  # v1 runs on 8080

        name: backend-port


---

apiVersion: v1

kind: Pod
```

```yaml
metadata:
  name: backend-v2
  labels:
    app.kubernetes.io/name: backend
    version: v2
spec:
  containers:
  - name: backend
    image: my-backend:v2
    ports:
      - containerPort: 9090  # v2 runs on 9090
        name: backend-port
```

- `backend-v1` **Pod** listens on **port 8080**.
- `backend-v2` **Pod** listens on **port 9090**.
- **Both Pods use the same named port (`backend-port`)** for consistency.

Step 2: Define a Single Service to Route Traffic

```yaml
apiVersion: v1

kind: Service

metadata:

  name: backend-service

spec:

  selector:

    app.kubernetes.io/name: backend  # Matches both v1 & v2
Pods

  ports:

  - name: service-port

    protocol: TCP

    port: 80  # Service exposes port 80

    targetPort: backend-port  # Routes to named port in Pods
```

- The Service **routes requests coming to port 80** to the Pods' **backend-port**.
- Since each Pod has **different containerPorts (8080, 9090)** but **the same named port (`backend-port`)**, the Service **automatically maps them correctly**.
- Clients can access the backend **without knowing the exact backend Pod ports**.

🔹 **What Happens Internally?**

1️⃣A user makes a request to `backend-service` on **port 80**.

2️⃣The Service **forwards the request to any running Pod** (`backend-v1` or `backend-v2`).

3️⃣Kubernetes **automatically resolves the correct port (8080 for v1, 9090 for v2)** using the named port `backend-port`.

Understanding the Multi-Port Service in Kubernetes

This Kubernetes **Service** definition exposes **two ports (80 and 443)** and forwards traffic to **two different target ports (9376 and 9377)** on selected Pods.

🔹 **Breakdown of the YAML**

```
apiVersion: v1

kind: Service

metadata:

  name: my-service

spec:

  selector:

    app.kubernetes.io/name: MyApp  # Selects Pods with this
label

  ports:

    - name: http
```

```
      protocol: TCP
      port: 80         # Service listens on port 80
      targetPort: 9376 # Forwards traffic to 9376 on
selected Pods
    - name: https
      protocol: TCP
      port: 443        # Service listens on port 443
      targetPort: 9377 # Forwards traffic to 9377 on
selected Pods
```

- **Selector:** Matches all Pods labeled as **`app.kubernetes.io/name: MyApp`**.
- **Ports:**
  - **Port 80** (HTTP) ➔ forwards requests to **9376** inside Pods.
  - **Port 443** (HTTPS) ➔ forwards requests to **9377** inside Pods.
- **Traffic Distribution:** Any request coming to **port 80 or 443 of the Service** gets routed to a **Pod running MyApp**.

🔹 **Example Use Case: Exposing a Web Application**

Imagine we have a **web application with a backend API** running inside Kubernetes.

Step 1: Create Two Pods (with Different Ports)

```
apiVersion: v1

kind: Pod

metadata:

  name: myapp-pod

  labels:

    app.kubernetes.io/name: MyApp

spec:

  containers:

    - name: web-server

      image: nginx

      ports:
```

```yaml
        - containerPort: 9376  # Handles HTTP requests

    - name: api-server

      image: my-api

      ports:

        - containerPort: 9377  # Handles HTTPS requests
```

- The **web-server container** (e.g., `nginx`) listens on **port 9376**.
- The **API server container** listens on **port 9377** for secure traffic.

Step 2: Create the Multi-Port Service

```yaml
apiVersion: v1

kind: Service

metadata:

  name: myapp-service

spec:

  selector:
```

```yaml
    app.kubernetes.io/name: MyApp
  ports:
    - name: http
      protocol: TCP
      port: 80
      targetPort: 9376  # Web server
    - name: https
      protocol: TCP
      port: 443
      targetPort: 9377  # API server
```

- When users access **`http://myapp-service:80`**, requests go to the **web server** on port **9376**.
- When users access **`https://myapp-service:443`**, requests go to the **API server** on port **9377**.

🔹 Why Is This Useful?

✅ **Single Service, Multiple Ports:** We don't need separate Services for HTTP and HTTPS.

✅ **Load Balancing:** Kubernetes distributes traffic between multiple Pods running the same app.

✅ **Simplified Network Configuration:** Clients only need to know the Service name (`myapp-service`), not the backend port details.

Headless Services in Kubernetes

A **Headless Service** in Kubernetes is a type of Service that **does not provide load balancing or a single cluster IP**, but instead directly exposes the **individual Pods' IP addresses**. This is useful when applications need direct access to each Pod, such as databases or StatefulSets.

🔹 How Does a Headless Service Work?

- A normal Kubernetes **Service (ClusterIP)** assigns a single IP address and load balances traffic to different Pods.
- A **Headless Service (ClusterIP: None)** does **not** assign a single IP.
- Instead, it **returns individual Pod IPs** when queried, allowing clients to connect directly to specific Pods.

🔹 Why Use a Headless Service?

✅ **Direct Communication** - Useful for databases like **MySQL, MongoDB, and Cassandra**, where clients need to communicate with specific Pods.

✅ **Stateful Applications** - When a Pod has persistent data and cannot be randomly load-balanced.

✅ **Service Discovery** - Applications can discover and interact with each Pod individually.

🔹 Example: Headless Service for a Database

Imagine a **MongoDB StatefulSet** where each Pod needs to be directly accessible.

1️⃣ Create a Headless Service

```yaml
apiVersion: v1
kind: Service
metadata:
  name: mongo-service
spec:
  clusterIP: None  # Makes it headless
  selector:
    app: mongo
  ports:
```

```yaml
  - port: 27017
    targetPort: 27017
```

🔹 **`clusterIP: None`** means no virtual IP is assigned, and DNS will resolve to individual Pod IPs.

2️⃣ **Create a StatefulSet with MongoDB**

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mongo
spec:
  serviceName: mongo-service  # Links to the Headless
Service
  replicas: 3  # Three MongoDB replicas
  selector:
```

```yaml
    matchLabels:
      app: mongo
  template:
    metadata:
      labels:
        app: mongo
    spec:
      containers:
        - name: mongo
          image: mongo
          ports:
            - containerPort: 27017
```

🔹 The **StatefulSet** ensures that **Pods get stable hostnames** like:

- `mongo-0.mongo-service.default.svc.cluster.local`
- `mongo-1.mongo-service.default.svc.cluster.local`

- `mongo-2.mongo-service.default.svc.cluster.local`

🔹 How Does It Work?

- Instead of a single **`mongo-service`** IP, DNS resolves to **individual MongoDB Pods**.

An application connecting to MongoDB can reach specific replicas like:

```
mongo --host mongo-0.mongo-service
```

🔹 When Should You Use Headless Services?

🔹 **Databases (MongoDB, MySQL, PostgreSQL, Cassandra)** - When clients need to connect to specific instances.

🔹 **Message Brokers (Kafka, RabbitMQ)** - Where direct communication with nodes is required.

🔹 **Peer-to-Peer Applications** - Services like Elasticsearch that require direct communication.

🔹 **Custom Load Balancing** - If an external load balancer (like Envoy or Nginx) manages traffic instead of Kubernetes.

Real-World Example: Headless Service for a MongoDB Replica Set

Imagine you're deploying **MongoDB** in Kubernetes as a **Replica Set**. This setup ensures **high availability and failover** for your database.

🔹 **Scenario**

You're running a **three-node MongoDB cluster** on Kubernetes using a **StatefulSet**. Your application needs to connect to each MongoDB Pod **individually** instead of a load-balanced service.

1️⃣ Define a Headless Service for MongoDB

```yaml
apiVersion: v1

kind: Service

metadata:

  name: mongo-headless

spec:
```

```yaml
  clusterIP: None  # Makes it headless (no single IP)

  selector:

    app: mongo

  ports:

    - port: 27017

      targetPort: 27017
```

🔹 **Why is this needed?**

- Since **`clusterIP: None`**, it won't provide a single IP.
- Instead, Kubernetes **DNS will return all individual Pod IPs**.

2️⃣ Create a MongoDB StatefulSet

```yaml
apiVersion: apps/v1

kind: StatefulSet

metadata:
```

```
  name: mongo
spec:
  serviceName: mongo-headless  # Connects to the Headless
Service
  replicas: 3  # Three MongoDB replicas
  selector:
    matchLabels:
      app: mongo
  template:
    metadata:
      labels:
        app: mongo
    spec:
      containers:
      - name: mongo
        image: mongo:latest
```

```yaml
        ports:
        - containerPort: 27017
        volumeMounts:
        - name: mongo-storage
          mountPath: /data/db
  volumeClaimTemplates:
  - metadata:
      name: mongo-storage
    spec:
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 1Gi
```

🔹 **How Does This Work?**

- Kubernetes **assigns stable DNS names** to each Pod:

    - `mongo-0.mongo-headless.default.svc.cluster.local`
    - `mongo-1.mongo-headless.default.svc.cluster.local`
    - `mongo-2.mongo-headless.default.svc.cluster.local`
- The **MongoDB replica set** can now communicate using these DNS names.

3️⃣ Initialize MongoDB Replica Set

Once the StatefulSet is running, you need to initialize the **MongoDB replica set** inside one of the Pods.

Step 1: Enter the First MongoDB Pod

```
kubectl exec -it mongo-0 -- mongo
```

Step 2: Initialize the Replica Set

```
rs.initiate({
  _id: "rs0",
  members: [
```

```
    { _id: 0, host:
"mongo-0.mongo-headless.default.svc.cluster.local:27017" },
    { _id: 1, host:
"mongo-1.mongo-headless.default.svc.cluster.local:27017" },
    { _id: 2, host:
"mongo-2.mongo-headless.default.svc.cluster.local:27017" }
  ]
})
```

🔹 **What This Does:**

- It **sets up the replica set** using the individual Pod hostnames.
- Each MongoDB instance **knows the other instances** and can sync data.

4️⃣ Application Connection

Your application can now connect to the MongoDB **replica set** using:

```
mongodb://mongo-0.mongo-headless.default.svc.cluster.local:2
7017,mongo-1.mongo-headless.default.svc.cluster.local:27017,
```

```
mongo-2.mongo-headless.default.svc.cluster.local:27017/?repl
icaSet=rs0
```

🔹 **Why is This Important?**

- The app connects to **all MongoDB instances**.
- If a **Pod goes down**, MongoDB will elect a **new primary node** automatically.
- The system remains **highly available**.

Why Not Use a Normal (ClusterIP) Service for Databases Like MongoDB?

A **normal ClusterIP service** provides a **single IP** that load balances traffic across multiple Pods. While this works for **stateless applications**, it's problematic for **stateful databases like MongoDB, Cassandra, or Kafka**. Here's why:

🔹 1️⃣ **Stateful Databases Require Direct Pod Communication**

- A database like **MongoDB (Replica Set)** or **Kafka (Broker Nodes)** needs each instance to **communicate directly** with others.

- If we use a **ClusterIP Service**, the traffic gets **randomly load-balanced** across all Pods.
- This **breaks** leader elections, replication, and node-to-node communication.

Example Issue with a ClusterIP Service:

```
mongodb://my-service.default.svc.cluster.local:27017
```

- The service forwards requests **randomly** to `mongo-0`, `mongo-1`, or `mongo-2`.
- MongoDB **needs to connect to specific nodes**, but it **can't distinguish them**.
- This **prevents proper Replica Set initialization**.

🔹 2️⃣ Primary and Replica Nodes Need a Stable Identity

- In a **MongoDB Replica Set**, one node is elected as **Primary**, and others as **Replicas**.
- If a **Replica wants to sync data**, it must know the **exact hostname** of the Primary.

- A **ClusterIP Service hides the Pod IPs**, making it **impossible to form a proper Replica Set**.

Example:

✅ **With Headless Service (Correct)**

```
mongodb://mongo-0.mongo-headless.default.svc.cluster.local:27017,mongo-1.mongo-headless.default.svc.cluster.local:27017,mongo-2.mongo-headless.default.svc.cluster.local:27017/?replicaSet=rs0
```

- Each MongoDB instance can **directly talk** to others using **Pod DNS names**.
- If `mongo-0` becomes Primary, replicas know exactly where to send write requests.

❌ **With ClusterIP Service (Incorrect)**

```
mongodb://mongo-service.default.svc.cluster.local:27017
```

- The **service randomly routes traffic**, so Replicas can't find the **Primary node**.

- MongoDB **fails to maintain replication** and **loses consistency**.

🔹 3️⃣ **Stateful Workloads Need Persistent Storage**

- ClusterIP services work well for **stateless applications** like APIs, but databases need **stable storage**.
- **StatefulSets + Headless Services** ensure: ✅ Each database Pod has a **stable identity** (like `mongo-0`, `mongo-1`).
  ✅ Each Pod gets its own **PersistentVolume** for storing data.

🔹 4️⃣ **ClusterIP Service Breaks Database Failover**

- If the **Primary node dies**, MongoDB must **elect a new leader**.
- With a **Headless Service**, all replicas **immediately update their connections** to the new leader.
- With a **ClusterIP Service**, the app **won't know which node is the new Primary**.

How DNS Works in Kubernetes Services?

Kubernetes provides **built-in DNS** to allow services, pods, and other resources to communicate using domain names instead of IP addresses. This is essential because **Pods are dynamic**, and their IP addresses keep changing.

1️⃣ How Does Kubernetes DNS Work?

Kubernetes uses **CoreDNS** as its built-in DNS service. It automatically assigns a **DNS name** to each Service and Pod inside the cluster.

📌 **When a Service is created**, Kubernetes **registers it in the internal DNS**, allowing other Pods to access it using a domain name.

DNS Naming Format for Services

```
<ServiceName>.<Namespace>.svc.cluster.local
```

- **ServiceName** → The name of the Service.
- **Namespace** → The Kubernetes namespace where the Service is deployed.
- **svc.cluster.local** → The default cluster domain.

2️⃣ Example of DNS Resolution in Kubernetes

🔹 **Scenario:**

- You have a **backend service** named `backend-svc` running in the `default` namespace.
- A **frontend pod** wants to communicate with it.

✅ DNS Resolution in Action

The frontend pod can use the **service name** directly to connect:

```
http://backend-svc
```

If the pod is in a different namespace, it must use:

```
http://backend-svc.default.svc.cluster.local
```

- CoreDNS resolves this to the **ClusterIP** of `backend-svc`, forwarding requests to the correct pods.

3️⃣ DNS Resolution for Headless Services

🔹 **Regular Services (ClusterIP, NodePort, LoadBalancer):**

- Resolve to a **single ClusterIP**, which load balances traffic to multiple pods.

🔹 **Headless Services (`clusterIP: None`):**

- Instead of a **single IP**, CoreDNS returns a list of **Pod IPs**.
- This allows direct pod-to-pod communication, useful for databases like MongoDB or Cassandra.

📌 **Example:** If `mongo-service` is headless (`clusterIP: None`), running:

```
nslookup mongo-service.default.svc.cluster.local
```

will return a list of **MongoDB pod IPs** instead of a single ClusterIP.

4️⃣ How DNS Resolves Service Names?

When a pod makes a request to `backend-svc`, Kubernetes DNS resolves it in **three steps**:

1️⃣ **CoreDNS intercepts the request** and checks its records.

2️⃣ **If a match is found,** it returns the ClusterIP (for normal services) or Pod IPs (for headless services).

3️⃣The request is then forwarded to the correct **pod** based on the Service's selector.

What Happens Behind the Scenes When a Kubernetes Service is Created?

When you create a Kubernetes Service, a lot of things happen in the background to ensure network connectivity, load balancing, and service discovery. Let's break it down step by step.

1️⃣Step 1: Service Definition & API Request

When you apply a Service YAML like this:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
```

```yaml
selector:
  app: my-app
ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
```

- **Kubernetes API Server** receives the request.
- It stores the Service definition in **etcd (the Kubernetes key-value store).**
- The Service is now registered in the cluster.

2️⃣Step 2: Service Gets a ClusterIP (if applicable)

- If the Service type is **ClusterIP (default)**, Kubernetes assigns a **virtual IP (ClusterIP)**.
- This **IP never changes** as long as the Service exists.
- The ClusterIP is stored in **kube-proxy** running on each node.

3️⃣ Step 3: EndpointSlice Controller Finds Matching Pods

- Kubernetes checks **which Pods match the selector (`app: my-app`)**.
- It creates an **EndpointSlice** that stores the IPs of all matched pods.

Example of an EndpointSlice:

```
apiVersion: discovery.k8s.io/v1

kind: EndpointSlice

metadata:

  name: my-service-abc123

addressType: IPv4

endpoints:

  - addresses:

      - 10.244.1.12  # Pod 1 IP

      - 10.244.2.25  # Pod 2 IP

ports:
```

```
- name: http
  port: 8080
  protocol: TCP
```

-
- **Kube-proxy on each node watches for these EndpointSlices.**

4️⃣ Step 4: Kube-Proxy Updates IPTables/IPVS Rules

- **`kube-proxy`** (running on each node) updates **iptables or IPVS rules** to forward traffic.
- If using **iptables**, a rule is added to forward requests to one of the Pod IPs in a round-robin way.
- If using **IPVS**, it creates a **load balancing table**.

🔹 **Example of iptables rules created:**

```
-A KUBE-SERVICES -d 10.96.0.10 -p tcp --dport 80 -j KUBE-SVC-XXXXXXX

-A KUBE-SVC-XXXXXXX -m statistic --mode random --probability 0.5 -j KUBE-SEP-YYYYYYY
```

```
-A KUBE-SEP-YYYYYYY -s 10.244.1.12 -j DNAT --to-destination
10.244.1.12:8080
```

- Requests to **ClusterIP (10.96.0.10:80)** are forwarded to **Pod IPs (10.244.1.12:8080, etc.).**

5️⃣ Step 5: CoreDNS Updates DNS Records

- **CoreDNS**, the Kubernetes DNS system, gets updated.

A new DNS entry is added:

```
my-service.default.svc.cluster.local → 10.96.0.10
(ClusterIP)
```

-
- Now, any pod can resolve `http://my-service` to its **ClusterIP**.

6️⃣ Step 6: Traffic Flowing to Service

📌 What Happens When a Pod Sends a Request to the Service?

A pod tries to access the service:

```
curl http://my-service
```

1. DNS resolves `my-service` to **ClusterIP (10.96.0.10).**
2. **Kube-proxy intercepts** the request and **forwards it to a Pod IP** (e.g., 10.244.1.12:8080).
3. The target **Pod receives the request and processes it.**

7️⃣ What Happens When Pods Scale Up or Down?

- The **EndpointSlice Controller** constantly monitors Pods.
- If new Pods are added (e.g., autoscaling), their IPs are automatically added to the EndpointSlice.
- If Pods are deleted, their IPs are removed.
- `kube-proxy` updates its rules accordingly.

8️⃣ What Happens If a Service is Deleted?

- **The ClusterIP is removed** from kube-proxy.
- **EndpointSlice is deleted**, so Pods are no longer linked to the Service.
- **DNS records are removed**, so the Service name won't resolve anymore.

Does Kubernetes Create EndpointSlices for Each Service?

Yes, **Kubernetes automatically creates EndpointSlices** for each Service that has a selector. The **EndpointSlice controller** manages them and updates them dynamically as Pods are added or removed.

🔹 How EndpointSlices Work

- When you create a Service, Kubernetes automatically creates an **EndpointSlice** containing all matching Pod IPs.
- If a Pod is deleted, Kubernetes **removes its IP** from the EndpointSlice.
- If new Pods matching the selector come up, Kubernetes **adds their IPs** to the EndpointSlice.

What Happens If You Manually Remove a Pod IP from EndpointSlice?

If you manually edit the EndpointSlice and remove a Pod's IP:

1. **The Pod will still be running**, but the Service will no longer route traffic to it.
2. **Kube-proxy will not forward requests** to that Pod anymore.
3. The Pod will still be accessible **directly via its Pod IP** but **not via the Service**.

💡 **However, Kubernetes will automatically restore the removed Pod IP** because the EndpointSlice controller continuously syncs it. So, the change is temporary.

What Happens If You Manually Add a Pod IP to EndpointSlice?

If you manually add an IP of a non-existing or unrelated Pod:

1. **Kube-proxy will try to send traffic** to that IP.
2. If the IP does not belong to an active Pod, requests will **fail or be misrouted**.
3. Kubernetes **may override your changes** because the EndpointSlice controller continuously updates the list based on actual Pods.

💡 **Manual modifications to EndpointSlices are not recommended** because Kubernetes will eventually correct the changes.

How to Control Which Pods Get Added to the EndpointSlice?

- Instead of modifying EndpointSlices manually, **use proper Service selectors**.
- If you want **manual control over endpoints**, use a **Headless Service** (`ClusterIP: None`) and manage endpoints yourself.

🔹 Summarizing Kubernetes Services 🔹

Kubernetes Services play a **crucial role** in enabling seamless communication between Pods, providing a stable endpoint despite the **dynamic and ever-changing** nature of Kubernetes environments. They abstract the underlying complexity of networking and ensure that applications remain **discoverable, scalable, and resilient**. 🚀

From **ClusterIP for internal communication** to **NodePort and LoadBalancer for external access**, Services offer various options to fit different deployment needs. Additionally, **Headless Services** and **ExternalName Services** provide even more flexibility for handling DNS-based routing and custom networking setups. 🌐

By leveraging **Selectors, Endpoints, and Service Discovery mechanisms**, Kubernetes Services ensure that traffic is efficiently routed to healthy Pods, balancing the load and maintaining high availability. **Integrating them with Ingress and DNS further enhances their power**, making Kubernetes a truly production-ready orchestration system. 💡

As we wrap up this discussion, remember that **understanding and effectively using Services is key to managing Kubernetes networking efficiently**. Keep practicing, experimenting, and deep-diving into Kubernetes – the more you explore, the more powerful your infrastructure becomes! 💪

🔹 Wrapping Up & Looking Ahead 🔹

As we conclude this deep dive into **Kubernetes Services**, I want to take a moment to express my gratitude for your **continuous support and engagement** in this Kubernetes learning journey. 🙌 Your enthusiasm and curiosity drive me to share more valuable insights, ensuring that we all grow together in mastering Kubernetes. 🚀

Kubernetes Services are **a fundamental component of cluster networking**, ensuring seamless communication between Pods and enabling efficient traffic routing. Mastering them is key to building **scalable, resilient, and production-ready applications** in Kubernetes. 🌐

But this is just the beginning! **Stay tuned for more hands-on Kubernetes tasks**, where we'll continue exploring essential concepts, practical implementations, and real-world use cases. 💡

👉 **Follow me for more daily Kubernetes content** filled with practical examples, best practices, and deep insights. Let's keep learning, building, and growing together! 💪🔥

🔹 **Thank you for being part of this journey!** Keep supporting, keep learning, and stay connected for more. 🚀