How does Ingress act as Kubernetes' smart traffic controller?

Imagine an **office building** with one **main reception (Ingress)** that manages all incoming visitors:

- **The receptionist at the front desk** asks each visitor where they need to go.
- If a visitor says, **"I need to meet HR,"** the receptionist directs them to the **HR Department**.
- If a visitor says, **"I have a meeting in the IT department,"** they are guided there.

Now, in the **Kubernetes world**:

- A visitor types **mycompany.com/hr** → The **Ingress forwards** them to the **HR Service**.
- Another visitor types **mycompany.com/it** → The **Ingress routes** them to the **IT Service**.

An **Ingress** in Kubernetes is an API object that **manages external access to services inside the cluster**, typically over **HTTP and HTTPS**. It acts as a **reverse proxy** and routes incoming requests to the correct **Services** based on rules like **hostnames, paths, and headers**.

https://www.armosec.io/wp-content/uploads/2021/11/617fae2d495f2526b09ce6fd_k8s-ingress-01-100841247-large.jpg

Reverse proxy

A **reverse proxy** sits **in front of multiple servers** and:

1. **Handles incoming requests** from users.
2. **Forwards them** to the correct **backend server** (depending on the URL, load, or other rules).
3. **Sends back the response** from the server to the user.

🔷 **Why Use a Reverse Proxy?**

- **Hides** the backend servers (for security).
- **Distributes traffic** evenly (load balancing).
- **Handles HTTPS** encryption (SSL/TLS termination).
- **Caches responses** to make things faster.

🔷 **Example in Kubernetes**

- **Ingress in Kubernetes is a type of reverse proxy!**
- If you visit `mywebsite.com/shop`, **Ingress (reverse proxy)** routes the request to the **Shop Service**.
-

Why is Ingress Needed?

Without Ingress, to expose a service externally, you would use **NodePort** or **LoadBalancer**, but they have limitations:

- **NodePort** exposes services on a static port on each Node, but it's not very flexible.
- **LoadBalancer** provisions a separate cloud load balancer for each service, which can be expensive.

💡 **Ingress provides a more efficient way to expose multiple services using a single external endpoint.**

Key Features of Ingress

✅ **Path-based routing** (e.g., `/shop` → Service A, `/blog` → Service B)
✅ **Host-based routing** (e.g., `shop.example.com` → Service A, `blog.example.com` → Service B)
✅ **SSL/TLS termination** (Handles HTTPS encryption) (The website secure.example.com should **only** accept HTTPS traffic., We have an SSL certificate stored in Kubernetes as a **Secret**.)
✅ **Load balancing** (Distributes traffic across multiple Pods)

✅ **Rewrite & Redirect Rules** (Modify URLs before forwarding requests) (Users visit `example.com/old-page`, but we want to **redirect them to** `example.com/new-page`.

)

How Ingress Works

1. **Client** sends an HTTP request to the cluster.
2. **Ingress Controller** (e.g., Nginx Ingress Controller) receives the request.
3. Based on the **Ingress rules**, the request is forwarded to the correct **Service**, which then reaches the appropriate **Pod**.

Kubernetes Ingress Setup with Kind on Mac

This guide covers everything from **installing Kind** to **deploying a demo** with **Ingress-based routing**.

🔷 1. Install Pre-requisites

Ensure you have the following installed:

📌 Install Homebrew (if not installed)

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

📌 Install Kind

```sh
brew install kind
```

📌 Install kubectl

```sh
brew install kubectl
```

📌 Verify Installation

```sh
```

```
kind version
kubectl version --client
```

✅ If installed correctly, you will see version details.

🔷 2. Create a Kind Cluster

Run the following command to create a Kind cluster with Ingress enabled:

kind create cluster --config=kind-config.yaml

```
-----
cat <<EOF | kind create cluster --config=-
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
  extraPortMappings:
  - containerPort: 80
    hostPort: 80
  - containerPort: 443
    hostPort: 443
EOF
```

📌 Verify Cluster

```
sh
-----
kubectl cluster-info
kubectl get nodes
```

✅ The cluster should be **Running**.

🔷 3. Install Ingress Controller

Run:

```
sh
-----
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/main/deploy/static/provider/kind/deploy.yaml



correct the node label:

kubectl label nodes $(kubectl get nodes -o=jsonpath='{.items[0].metadata.name}')
ingress-ready=true
```

📌 Wait for Ingress to be Ready

```
sh
-----
kubectl wait --namespace ingress-nginx \
  --for=condition=ready pod \
  --selector=app.kubernetes.io/component=controller \
  --timeout=90s
```

📌 Verify

```
kubectl get pods -n ingress-nginx
```

✅ The Ingress controller should be **Running**.

🔷 4. Deploy App1 & App2

📌 Create a Deployment File (`deployments.yaml`)

Create `deployments.yaml` and add the following:

```
yaml
-----
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app1
spec:
  replicas: 1
  selector:
    matchLabels:
      app: app1
  template:
    metadata:
      labels:
        app: app1
    spec:
      containers:
      - name: app1
        image: nginx
        ports:
        - containerPort: 80
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app2
spec:
  replicas: 1
  selector:
    matchLabels:
      app: app2
  template:
    metadata:
      labels:
        app: app2
    spec:
      containers:
      - name: app2
        image: httpd
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: app1-service
spec:
```

```
  selector:
    app: app1
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: app2-service
spec:
  selector:
    app: app2
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

📌 Apply Deployments & Services

```
sh
-----
kubectl apply -f deployments.yaml
```

📌 Verify

```
sh
-----
kubectl get pods
kubectl get svc
```

✅ Both **App1** and **App2** should be **Running**.

🔷 5. Configure Ingress for Path-Based Routing

📌 Create an Ingress File (`ingress.yaml`)

Create `ingress.yaml` and add:

```
yaml
-----
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: path-based-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx
  rules:
  - host: localhost
    http:
      paths:
      - path: /app1
        pathType: Prefix
        backend:
          service:
            name: app1-service
            port:
```

```
              number: 80
      - path: /app2
        pathType: Prefix
        backend:
          service:
            name: app2-service
            port:
              number: 80
```

📌 Apply Ingress

```
sh
-----
kubectl apply -f ingress.yaml
```

📌 Verify

```
sh
-----
kubectl get ingress
```

✅ The Ingress should be **configured**.

🔷 6. Test the Setup

📌 Access Applications in Browser

- **App1:** 👉 http://localhost/app1
- **App2:** 👉 http://localhost/app2

✅ You should see the **Nginx** page for `/app1` and **Apache** page for `/app2`.

🔷 7. Graceful Shutdown

1️⃣ Delete Ingress, Services, and Deployments

```
sh
-----
kubectl delete -f ingress.yaml
kubectl delete -f deployments.yaml
```

2️⃣ Stop & Delete Kind Cluster

```
sh
-----
kind delete cluster
```

✅ This **cleans up everything**.

🔷 8. Restart Everything

1️⃣ Start Kind Cluster

```
sh
-----
kind create cluster --config=- <<EOF
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
  extraPortMappings:
  - containerPort: 80
    hostPort: 80
  - containerPort: 443
    hostPort: 443
EOF
```

2️⃣ Re-Deploy Everything

```
sh
-----
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-
nginx/main/deploy/static/provider/kind/deploy.yaml
kubectl apply -f deployments.yaml
kubectl apply -f ingress.yaml
```

3️⃣ Verify

```
sh
-----
kubectl get pods,svc,ingress
```

✅ Now test `http://localhost/app1` and `http://localhost/app2` again.

🎯 Quick Commands Summary

Action	Command
Create Cluster	`kind create cluster --config=- <config>`
Check Cluster	`kubectl cluster-info`
Install Ingress	`kubectl apply -f <ingress-url>`
Deploy Apps & Services	`kubectl apply -f deployments.yaml`
Deploy Ingress	`kubectl apply -f ingress.yaml`
Verify Resources	`kubectl get pods,svc,ingress`
Test Apps	`http://localhost/app1`, `http://localhost/app2`
Stop Apps & Ingress	`kubectl delete -f deployments.yaml && kubectl delete -f ingress.yaml`
Delete Kind Cluster	`kind delete cluster`

🚀 Conclusion

This guide provides a **step-by-step** method to:

- **Install & Set up** `Kind` **on Mac**
- **Deploy & Expose** `App1` **&** `App2` **with Ingress**
- **Stop & Restart Everything Gracefully**

Which Kubernetes object is responsible for exposing Ingress resources to the outside world?
a) NodePort
b) LoadBalancer
c) Ingress Controller
d) Service

Answer: c) Ingress Controller