BY FENIL GAJJAR

KUBERNETES
DAILY TASKS

KUBERNETES INGRESS

- COMPREHENSIVE GUIDE
- THEORY + PRACTICAL TASKS
- REAL TIME SCENARIO TASKS
- FOLLOW FOR MORE TASKS
- linkedin.com/in/fenil-gajjar

CONTACT US

fenilgajjar.devops@gmail.com

🚀 Kubernetes Ingress Unlocked: Seamless Traffic Routing Like a Pro! 🚀

🌟 Welcome to Another Milestone in Our Kubernetes Journey! 🚀

First, I want to express my **deep gratitude** for all the support and engagement you've shown throughout this Kubernetes series. 🙌 Your enthusiasm and encouragement drive me to bring even more valuable insights every day. This journey wouldn't be the same without your support, so **thank you for being part of it!**

🌍 Today's Focus: Kubernetes Ingress

As we continue mastering Kubernetes, today we're unlocking a **powerful networking component—Ingress!** 💡 In the real world, applications need a way to be **accessible from the outside world**, whether through **HTTP, HTTPS, or custom routes**. Kubernetes provides multiple ways to expose services, but Ingress takes it **a step further**, offering a more **efficient, flexible, and centralized approach** for managing external access.

In this guide, we'll dive deep into:

✅ What Kubernetes Ingress is and why it's essential

✅ How it works behind the scenes

✅ Key configurations and best practices

✅ Hands-on examples to set up and manage Ingress effectively

Whether you're a beginner or already familiar with Kubernetes networking, this will **level up your understanding and skills**. So, get ready to explore, experiment, and enhance your Kubernetes expertise! 🔥

Stay tuned, and let's keep growing together. 💪 **Keep supporting, keep learning, and let's master Kubernetes one step at a time!** 🚀

🚀 Why Do We Need Ingress?

Before Kubernetes Ingress came into play, exposing applications to the outside world was **not so easy**. Let's take a quick trip back in time and see how things worked before Ingress and why it became a game-changer! 🎯

🌍 The Pre-Ingress Era: How Did We Expose Services Before?

In Kubernetes, every Pod has its **own private IP**. But here's the problem:

👉 **Pods are ephemeral**—they come and go!

👉 Their IPs **keep changing**, making direct access impossible.

So, Kubernetes introduced **Services** to solve this issue by providing a **stable way** to connect to Pods. But how do we expose these services outside the cluster?

🚪 Option 1: NodePort

Kubernetes allowed us to expose services using **NodePort**, which opens a fixed port on each node. The issue?

❌ The port range is limited (30000-32767).

❌ It's **not user-friendly**—imagine remembering multiple ports for different services!

❌ Not scalable for production.

🌐 Option 2: LoadBalancer

Next, we had **LoadBalancer Services**, which brought us a step closer to real-world usage. A cloud provider assigns an **external IP**, and traffic is routed to the service.

But still...

❌ Every service needs a **separate LoadBalancer**.

❌ **Expensive**—creating a LoadBalancer for each service is costly.

❌ No built-in traffic routing features like **path-based or domain-based routing**.

🌟 Enter Ingress: The Game Changer!

To solve these challenges, **Kubernetes Ingress** was introduced! 🚀

✨ Instead of exposing each service separately, Ingress acts as a **smart gateway** that routes external traffic **inside the cluster** based on:

✅ **Hostnames (e.g., `app.example.com`)**

✅ **Paths (e.g., `/api`, `/dashboard`)**

✅ **TLS (HTTPS) termination**

✅ **Load balancing** between multiple services

Think of **Ingress as the front door** to your applications—it lets you define flexible, user-friendly routing rules in a **single place**.

So now, instead of managing multiple LoadBalancers or dealing with NodePorts, we just set up an **Ingress Controller** and define simple rules. **Boom!** 🎯

🚀 **Less cost, less complexity, more power!**

🚀 What Exactly is Ingress in Kubernetes?

In Kubernetes, **Ingress** is a powerful API object that manages how external users access services within a cluster. It acts as a **smart gateway** that routes traffic based on **hostnames, paths, and rules**—eliminating the need for multiple LoadBalancers and messy NodePort configurations.

🌍 Understanding Ingress in Simple Terms

Think of **Ingress** as the receptionist in a large office building. When visitors arrive, the receptionist:

✅ **Greets them** (accepts incoming traffic)

✅ **Checks their purpose** (examines the requested hostname or path)

✅ **Directs them to the right department** (routes traffic to the correct service)

Without Ingress, visitors would have to **directly enter individual rooms** (NodePort) or **use separate entrances** (multiple LoadBalancers), which is inefficient.

🏆 Key Features of Ingress (Made Simple & Fun!)

🎯1️⃣Smart Traffic Routing

Imagine a **mall entrance** with different signs leading visitors to the right store. Ingress works the same way by directing users to the correct service based on:

✅ **Paths** (`/shop → frontend`, `/api → backend`)

✅ **Hostnames** (`shop.myapp.com → shop service`, `api.myapp.com → API service`)

🔒2️⃣Built-in Security with HTTPS (SSL/TLS Termination)

Ever noticed the 🔒 lock icon in your browser? That's **HTTPS encryption**, keeping data safe. Ingress makes it easy to handle SSL certificates, ensuring secure communication without configuring it in each service.

⚖️3️⃣Load Balancing for Performance

Just like a traffic cop managing cars at a busy intersection, Ingress **distributes** traffic evenly across multiple instances of a service. This prevents overload and ensures **smooth performance**.

🌎4️⃣Single External IP (Saves Cost & Complexity)

Without Ingress: Every service needs a **separate LoadBalancer**, increasing cloud costs.

With Ingress: One **central LoadBalancer** handles all traffic, making it **cost-effective & scalable**.

🔄5️⃣Easy Traffic Control (Rewrites, Redirects, & More!)

Ingress lets you:

🔄 **Redirect users** (e.g., `http → https`)

✂️ **Rewrite URLs** (`/old-page → /new-page`)

🚦 **Rate-limit requests** to prevent overload

📊6️⃣Works with Ingress Controllers

Ingress itself is just a set of rules, but it needs an **Ingress Controller** to enforce them. Popular controllers include:

- 🌐 **NGINX Ingress Controller** (Most common)

- 🚀 **Traefik** (Lightweight & dynamic)
- 🔥 **Istio Gateway** (For advanced networking)

🎯 Why Use Ingress? The Ultimate Traffic Manager in Kubernetes!

Imagine running a **bustling online store** with different sections like the **shop**, **API**, and **admin dashboard**. Without **Ingress**, every section would need its own separate **door** (LoadBalancer or NodePort), making things **messy, expensive, and hard to manage**.

But with **Ingress**, you get **one grand entrance** that intelligently routes visitors to the right place—saving money, improving security, and keeping traffic under control. 🚀

💡 Key Advantages of Using Ingress (The Problem It Solves!)

🏠1️⃣ Single Entry Point Instead of Multiple Exposures

🚫 **Without Ingress:** Each service needs its own LoadBalancer or NodePort, making it expensive and difficult to manage.

✅ **With Ingress:** A **single entry point** smartly routes traffic inside the cluster. No need for multiple external IPs!

> **Example:**
>
> - `shop.example.com` ➜ Routes traffic to the frontend service
> - `api.example.com` ➜ Routes traffic to the backend service
> - `admin.example.com` ➜ Routes traffic to the admin panel

🔒2️⃣Secure Communication with HTTPS & SSL/TLS Termination

🚫 **Without Ingress:** Each service needs separate SSL certificates, making security hard to manage.

✅ **With Ingress:** You can manage **SSL/TLS certificates centrally**, ensuring all traffic is **secure & encrypted**.

> **Think of it like this:** Instead of every store in a mall installing their own security system, the mall provides a **centralized security gate** for all!

⚖️3️⃣Load Balancing for Even Traffic Distribution

🚫 **Without Ingress:** Traffic might overwhelm certain services while others stay idle.

✅ **With Ingress:** Load balancing ensures requests are **distributed evenly**, preventing overload and boosting performance.

> **Example:** If 1,000 users hit your application, Ingress ensures they are routed across multiple backend servers efficiently.

🔄4️⃣Intelligent Traffic Control (Path-Based & Host-Based Routing)

🚫 **Without Ingress:** You need separate configurations for each service's traffic flow.

✅ **With Ingress:** You define **routing rules** in one place for organized traffic management.

> **Example:**
>
> - `example.com/shop` ➜ Goes to the **frontend**
> - `example.com/api` ➜ Goes to the **backend**
> - `example.com/admin` ➜ Goes to the **admin dashboard**

This keeps things **structured, simple, and easy to scale!**

🌎5️⃣Cost-Effective (No More Expensive LoadBalancers!)

🚫 **Without Ingress:** You need a LoadBalancer for each exposed service, increasing cloud costs.

✅ **With Ingress:** A **single** LoadBalancer handles all services, drastically reducing expenses.

> **Imagine paying for multiple toll gates vs. having one central highway entrance for all roads!**

🔥6️⃣URL Rewriting & Redirects (Better User Experience!)

🚫 **Without Ingress:** If a URL structure changes, users might get broken links.

✅ **With Ingress:** You can set up **redirects & rewrites** to ensure smooth navigation.

> **Example:**
>
> - Redirect `http://example.com` ➡ `https://example.com` 🔒

- Rewrite `/old-page` ➜ `/new-page` without breaking links

🚀7️⃣ Traffic Splitting & Canary Deployments

🚫 **Without Ingress:** Rolling out new features requires downtime or complex setups.

✅ **With Ingress:** You can **split traffic** between different versions of your app (e.g., 80% to v1, 20% to v2) for safe deployments.

> **Perfect for testing new features without impacting all users!**

🎯8️⃣ Works with Ingress Controllers for Extra Power!

Ingress itself is just a set of rules, but it works with **Ingress Controllers** to enforce them:

✅ **NGINX Ingress Controller** – The most popular & widely used

✅ **Traefik** – Lightweight & dynamic routing

✅ **Istio Gateway** – Advanced traffic management for microservices

These controllers enhance **performance, security, and scalability**!

🏗️ Ingress Architecture – Key Components

1️⃣ Ingress Controller 🛠️

- The brain behind **Ingress** that processes incoming traffic and applies routing rules.
- Common controllers: **NGINX, Traefik, HAProxy, AWS ALB Controller**.

2️⃣ Ingress Resource 📜

- A **YAML configuration file** that defines how traffic should be routed.
- Example: Redirect `app.example.com` to **Service A** and `shop.example.com` to **Service B**.

3️⃣ Ingress Class 🏷️

- Helps define **which Ingress Controller** should handle a specific Ingress resource.
- Example: Use **NGINX** for general apps and **AWS ALB** for cloud-based services.

4️⃣ Service 🏢

- Connects the **Ingress** to actual backend applications running in **Pods**.
- Works as a bridge between the **Ingress Controller** and applications.

5️⃣ **Pod (Application)** 🚀

- The final destination where requests are processed.
- Runs the application that serves the actual content to users.

🚀 What is an Ingress Controller & Why is it Important?

If **Ingress is the traffic manager** of Kubernetes, then **Ingress Controller is the engine** that makes it all work! 💡

Think of it like this: **Ingress is the road system**, but without traffic lights and signboards, it's chaos. 🚦 That's where **Ingress Controller** comes in – it ensures traffic flows smoothly to the right places!

📌 What is an Ingress Controller?

An **Ingress Controller** is a specialized **Kubernetes component** that processes the rules defined in the **Ingress resource** and handles incoming traffic accordingly. 🌍

💡 **Without an Ingress Controller, an Ingress resource alone does NOTHING!**

It's like having a train track (Ingress) but **no train operator** (Ingress Controller). 🚆

🎯 How Ingress Controller Works (Step-by-Step)

Let's say you have an e-commerce app with different services:

- **Frontend:** `shop.example.com`
- **Backend API:** `api.example.com`

Here's how an Ingress Controller handles this traffic:

1️⃣ **User requests `shop.example.com` 🌍**

2️⃣ The request **first reaches the Ingress Controller**. 🚦

3️⃣ The Controller **checks the Ingress rules**. 📜

4️⃣ It **routes traffic** to the correct service (`Frontend Service`). 🏢

5️⃣ The **service forwards traffic** to the right **Pods** (your app). 🚀

✅ **Boom! Your app works seamlessly!**

🔥 Why is an Ingress Controller Important?

📌 **Without it, Kubernetes Ingress won't work!**

📌 It **efficiently handles and routes external traffic** to internal services.

📌 It provides **SSL termination**, **authentication**, and **load balancing**.

📌 Reduces **costs** by eliminating multiple cloud load balancers.

📌 Enables **path-based** and **host-based routing**.

Imagine running **10 microservices** – instead of 10 different LoadBalancers, you can just use **one Ingress Controller** to manage all the traffic! 🚀

⚡ Popular Ingress Controllers

- 🔹 **NGINX Ingress Controller** – Most widely used, stable & feature-rich.
- 🔹 **Traefik** – Lightweight & supports auto-discovery.
- 🔹 **HAProxy** – High-performance with advanced traffic control.
- 🔹 **AWS ALB Ingress Controller** – For integrating AWS Load Balancer.
- 🔹 **Istio Gateway** – Used in service meshes for advanced traffic control.

🎯 How Ingress Works in Kubernetes

🔹 **Step 1: User Makes a Request**

When a user tries to access a service (e.g., `app.example.com`), their request first reaches the **Kubernetes cluster's external entry point**.

👉 **Without Ingress:** Every service would need a separate LoadBalancer, increasing complexity and cost.

👉 **With Ingress:** One centralized entry point directs traffic efficiently. 🚦

🔹 **Step 2: The Request Hits the Ingress Controller**

The request doesn't go straight to the application! Instead, it first reaches the **Ingress Controller**, which is responsible for processing the request. 🚀

🔹 **Ingress Controller reads the rules** defined in the **Ingress resource**.

🔹 It determines **where to send the traffic** based on the **domain name, path, or custom rules**.

✅ **Example:**

- Requests to `app.example.com` ➡ Forwarded to the **frontend service**
- Requests to `api.example.com` ➡ Sent to the **backend service**

- Requests to `app.example.com/images` ➜ Redirected to an **image storage service**

🔹 **Step 3: Ingress Controller Routes Traffic**

Once the Ingress Controller knows where to send the request, it **routes the traffic** to the correct **Kubernetes Service**, which then forwards it to the appropriate **Pods** running the application.

💡 **Think of Ingress as an Air Traffic Controller!**

It ensures every request lands at the right destination safely and efficiently. ✈️

✅ **Path-based routing example:**

```
- /home  → Frontend Service
- /api   → Backend API Service
- /static → CDN or Image Service
```

🔹 **Step 4: Response is Sent Back to the User**

Once the Pod processes the request, the response **follows the same path back**:

📌 **Pod → Service → Ingress Controller → User**

This ensures **secure, optimized, and seamless communication** between external users and internal applications. 🔥

📌 Complete YAML Configuration for Kubernetes Ingress Resource

Ingress in Kubernetes allows you to define routing rules to direct external traffic to internal services efficiently. Below is a **complete and well-structured YAML configuration** for an **Ingress resource**, assuming you're using **Kops as your Kubernetes cluster**.

🛠️ Prerequisites Before Applying Ingress

Before you create an Ingress resource, make sure:

✅ You have an **Ingress Controller** (such as Nginx Ingress Controller) installed and running in your cluster.

✅ You have **services running** that need to be exposed externally.

🎯 Complete Ingress YAML Configuration

The following example defines an **Ingress resource** that:

- Routes traffic to **multiple services** based on the request's domain or path.
- Supports **HTTPS with TLS termination** using a Kubernetes secret.
- Handles **path-based and host-based routing** for different microservices.

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

kind: Ingress

metadata:

  name: my-app-ingress

  namespace: default

  annotations:

    nginx.ingress.kubernetes.io/rewrite-target: /

    nginx.ingress.kubernetes.io/ssl-redirect: "true"

spec:

  ingressClassName: nginx

  tls:

  - hosts:

      - app.example.com
```

```
    secretName: tls-secret # TLS certificate stored in a
Kubernetes Secret
  rules:
  - host: app.example.com
    http:
      paths:
      - path: /         # Root path for frontend
        pathType: Prefix
        backend:
          service:
            name: frontend-service
            port:
              number: 80
      - path: /api      # API path for backend service
        pathType: Prefix
```

```
        backend:
          service:
            name: backend-service
            port:
              number: 8080
      - path: /static   # Static content path (e.g., images,
CSS)
        pathType: Prefix
        backend:
          service:
            name: static-service
            port:
              number: 80
```

📌 Explanation of This Ingress YAML

1️⃣ **Ingress Class & Controller:**

- `ingressClassName: nginx` ➜ Specifies that this Ingress should use the Nginx Ingress Controller.

2️⃣ **TLS Configuration:**

- The Ingress uses **HTTPS** and references a **TLS secret (`tls-secret`)** for securing the domain `app.example.com`.
- You must create this secret using a **TLS certificate** before applying the Ingress.

3️⃣ **Routing Rules:**

- Requests to `app.example.com/` ➜ Routed to **frontend-service** on port **80**.
- Requests to `app.example.com/api` ➜ Routed to **backend-service** on port **8080**.
- Requests to `app.example.com/static` ➜ Routed to **static-service** on port **80**.

4️⃣ **Annotations for Nginx:**

- `nginx.ingress.kubernetes.io/rewrite-target: /` → Ensures correct path forwarding.
- `nginx.ingress.kubernetes.io/ssl-redirect: "true"` → Forces HTTPS redirection.

🚀 Apply the Ingress Configuration

Once the Ingress Controller is running, apply the YAML file using:

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

✅ Verify the Ingress Resource

Check if the Ingress resource is created successfully:

```
kubectl get ingress
```

You should see an output like:

```
NAME                 CLASS     HOSTS                 ADDRESS

PORTS    AGE
```

```
my-app-ingress     nginx    app.example.com    192.168.1.100
80,443    5m
```

✅ Check Ingress Rules

```
kubectl describe ingress my-app-ingress
```

Kubernetes Ingress Example for an E-Commerce Platform

Below is a **Kubernetes YAML configuration** that sets up an **NGINX Ingress Controller** to route traffic to different microservices of an e-commerce application.

Step 1: Deploy Microservices

Let's create Deployments and Services for the **Frontend, Product, Order, and User Services**.

Frontend Service (`frontend-service.yaml`)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend-deployment
spec:
```

```
replicas: 2

selector:

  matchLabels:

    app: frontend

template:

  metadata:

    labels:

      app: frontend

  spec:

    containers:

    - name: frontend

      image: my-ecommerce-frontend:v1

      ports:

      - containerPort: 80
```

```
---

apiVersion: v1

kind: Service

metadata:

  name: frontend-service

spec:

  selector:

    app: frontend

  ports:

    - protocol: TCP

      port: 80

      targetPort: 80
```

Product Service (`product-service.yaml`)

```
apiVersion: apps/v1

kind: Deployment

metadata:

  name: product-deployment

spec:

  replicas: 2

  selector:

    matchLabels:

      app: product

  template:

    metadata:

      labels:

        app: product

    spec:
```

```
      containers:
      - name: product
        image: my-ecommerce-product:v1
        ports:
        - containerPort: 8080


---
apiVersion: v1
kind: Service
metadata:
  name: product-service
spec:
  selector:
    app: product
  ports:
```

```
    - protocol: TCP
      port: 8080
      targetPort: 8080
```

Order Service (`order-service.yaml`)

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

```
  template:
    metadata:
      labels:
        app: order
    spec:
      containers:
      - name: order
        image: my-ecommerce-order:v1
        ports:
        - containerPort: 8081


---
apiVersion: v1
kind: Service
metadata:
```

```
  name: order-service
spec:
  selector:
    app: order
  ports:
    - protocol: TCP
      port: 8081
      targetPort: 8081
```

User Service (`user-service.yaml`)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: user-deployment
spec:
```

```
replicas: 2

selector:

  matchLabels:

    app: user

template:

  metadata:

    labels:

      app: user

  spec:

    containers:

    - name: user

      image: my-ecommerce-user:v1

      ports:

      - containerPort: 8082
```

```
---

apiVersion: v1

kind: Service

metadata:

  name: user-service

spec:

  selector:

    app: user

  ports:

    - protocol: TCP

      port: 8082

      targetPort: 8082
```

Step 2: Create an Ingress Resource (`ingress.yaml`)

This Ingress resource defines routing rules based on paths.

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

kind: Ingress

metadata:

  name: ecommerce-ingress

  annotations:

    nginx.ingress.kubernetes.io/rewrite-target: /

spec:

  ingressClassName: nginx

  rules:

  - host: example.com

    http:

      paths:

      - path: /

        pathType: Prefix

        backend:
```

```
      service:
        name: frontend-service
        port:
          number: 80
  - path: /products
    pathType: Prefix
    backend:
      service:
        name: product-service
        port:
          number: 8080
  - path: /orders
    pathType: Prefix
    backend:
      service:
```

```
            name: order-service
            port:
              number: 8081
      - path: /users
        pathType: Prefix
        backend:
          service:
            name: user-service
            port:
              number: 8082
```

Step 3: Deploy the NGINX Ingress Controller

If you haven't already installed the **NGINX Ingress Controller**, you can deploy it using Helm:

```
helm repo add ingress-nginx

https://kubernetes.github.io/ingress-nginx

helm install nginx-ingress ingress-nginx/ingress-nginx
```

Step 4: Apply the Configurations

Now, apply all the YAML files to your Kubernetes cluster.

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

kubectl apply -f product-service.yaml

kubectl apply -f order-service.yaml

kubectl apply -f user-service.yaml

kubectl apply -f ingress.yaml
```

Step 5: Access the Services

Once everything is deployed:

- Visit `https://example.com/` → **Frontend Service**
- Visit `https://example.com/products` → **Product Service**

- Visit `https://example.com/orders` → **Order Service**
- Visit `https://example.com/users` → **User Service**

If you are running this in **minikube**, enable the ingress controller and get the ingress IP:

```
minikube addons enable ingress

kubectl get ingress
```

Why Is This a Good Setup?

✅ **Single Entry Point:** Users don't need to remember multiple IPs or ports.

✅ **Path-Based Routing:** Traffic is routed correctly to the respective microservice.

✅ **Cost-Effective:** No need for multiple Load Balancers.

✅ **Security & SSL Handling:** TLS termination can be handled at the Ingress level.

✅ **Scalability:** Can add more microservices easily.

Host-Based Routing in Kubernetes Ingress

In **host-based routing**, traffic is routed based on the **domain name (host)** rather than the URL path. This is useful when hosting **multiple applications** or **subdomains** on the same Kubernetes cluster.

📌 Real-Life Example: Hosting Multiple Applications on Different Domains

Imagine you have a **news company** with the following services:

- **Main Website:** `news.example.com`
- **Sports Section:** `sports.example.com`
- **Finance Section:** `finance.example.com`

Instead of creating separate Load Balancers for each, we use a **single Ingress** to route requests based on the hostname.

Step 1: Deploy Services

Each service has its own **deployment and service**.

Main Website Service (`news.example.com`)

```
apiVersion: apps/v1

kind: Deployment

metadata:

  name: news-deployment

spec:

  replicas: 2

  selector:

    matchLabels:

      app: news

  template:

    metadata:

      labels:

        app: news

    spec:
```

```
      containers:
      - name: news
        image: my-news-app:v1
        ports:
        - containerPort: 80


---
apiVersion: v1
kind: Service
metadata:
  name: news-service
spec:
  selector:
    app: news
  ports:
```

```
    - protocol: TCP
      port: 80
      targetPort: 80
```

Sports Section Service (`sports.example.com`)

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

```
    metadata:
      labels:
        app: sports
    spec:
      containers:
      - name: sports
        image: my-sports-app:v1
        ports:
        - containerPort: 80


---
apiVersion: v1
kind: Service
metadata:
  name: sports-service
```

```
spec:
  selector:
    app: sports
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

Finance Section Service (`finance.example.com`)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: finance-deployment
spec:
  replicas: 2
```

```
  selector:
    matchLabels:
      app: finance
  template:
    metadata:
      labels:
        app: finance
    spec:
      containers:
      - name: finance
        image: my-finance-app:v1
        ports:
        - containerPort: 80


---
```

```
apiVersion: v1

kind: Service

metadata:

  name: finance-service

spec:

  selector:

    app: finance

  ports:

    - protocol: TCP

      port: 80

      targetPort: 80
```

Step 2: Create an Ingress for Host-Based Routing

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

kind: Ingress
```

```
metadata:
  name: news-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx
  rules:
  - host: news.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: news-service
```

```
          port:
            number: 80
  - host: sports.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: sports-service
            port:
              number: 80
  - host: finance.example.com
    http:
      paths:
```

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

Step 3: Apply Ingress and Update DNS

Apply the ingress:

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

Check the ingress details:

```
kubectl get ingress
```

1. The output will show an **AWS Load Balancer hostname** (if using Kops on AWS).

   Add this hostname as **CNAME records** in Route 53 (or any DNS provider)

for:

- `news.example.com`
- `sports.example.com`
- `finance.example.com`

📌 **Expected Behavior**

- Visiting **`https://news.example.com/`** ➜ Routes to the **News Service**.
- Visiting **`https://sports.example.com/`** ➜ Routes to the **Sports Service**.
- Visiting **`https://finance.example.com/`** ➜ Routes to the **Finance Service**.

Each service remains independent while being managed under a **single Ingress and Load Balancer**.

Why Use Host-Based Routing?

✔ **Efficient:** No need for multiple Load Balancers.

✔ **Cost-Effective:** Reduces AWS ELB expenses.

✔ **Scalable:** Easily add new subdomains in the Ingress rule.

✔ **Simplifies Management:** Centralized control over multiple services.

🚀 Complete Example Scenario of Using Ingress in Kubernetes (with kOps)

💡 Scenario:

You are running a **Kubernetes cluster created using kOps** and want to expose two different applications (`frontend` and `backend`) using a **single domain** (`example.com`) with **Ingress for routing traffic**.

🛠️ What We Will Cover in This Example

✅ **Host-based Routing:** Route `api.example.com` to backend service & `app.example.com` to frontend service.

✅ **Path-based Routing:** Serve different services based on `/api` and `/app`.

✅ **TLS/HTTPS Support:** Secure traffic using an SSL certificate.

✅ **Nginx Ingress Controller Setup:** Install Ingress in a kOps cluster.

✅ **Complete Deployment, Service & Ingress YAMLs:** To make it fully working.

🔹 Step 1: Install an Ingress Controller on Your kOps Cluster

Since kOps does not come with an Ingress Controller by default, you need to install one:

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

This installs **NGINX Ingress Controller**, which will manage traffic routing.

🔹 Step 2: Create Deployments and Services for Frontend & Backend

We need two applications:

- **Frontend (`frontend-service`)** → Runs on port `80`.
- **Backend (`backend-service`)** → Runs on port `5000`.

📌 Deployment & Service for Frontend (React, Angular, or Vue App)

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

```
    spec:
      containers:
      - name: frontend
        image: myfrontend:latest
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: frontend-service
spec:
  selector:
    app: frontend
  ports:
```

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

📌 Deployment & Service for Backend (Node.js, Django, or Flask API)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend
  labels:
    app: backend
spec:
  replicas: 2
  selector:
```

```
    matchLabels:
      app: backend
  template:
    metadata:
      labels:
        app: backend
    spec:
      containers:
      - name: backend
        image: mybackend:latest
        ports:
        - containerPort: 5000
---
apiVersion: v1
kind: Service
```

```
metadata:
  name: backend-service
spec:
  selector:
    app: backend
  ports:
    - protocol: TCP
      port: 5000
      targetPort: 5000
  type: ClusterIP
```

🔹 Step 3: Create an Ingress Resource for Routing Traffic

Now, we define an **Ingress resource** that:

- Routes **`app.example.com`** to `frontend-service`
- Routes **`api.example.com`** to `backend-service`
- Routes **`example.com/api`** to `backend-service`

- Routes **`example.com/app`** to `frontend-service`
- Enables **HTTPS/TLS termination**

📌 Complete Ingress YAML

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

kind: Ingress

metadata:

  name: my-ingress

  annotations:

    nginx.ingress.kubernetes.io/rewrite-target: /

    nginx.ingress.kubernetes.io/ssl-redirect: "true"

spec:

  ingressClassName: nginx

  tls:

  - hosts:

    - example.com
```

```
    - app.example.com
    - api.example.com
    secretName: tls-secret
  rules:
  - host: app.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: frontend-service
            port:
              number: 80
  - host: api.example.com
```

```
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: backend-service
            port:
              number: 5000
  - host: example.com
    http:
      paths:
      - path: /app
        pathType: Prefix
        backend:
```

```
      service:
        name: frontend-service
        port:
          number: 80
  - path: /api
    pathType: Prefix
    backend:
      service:
        name: backend-service
        port:
          number: 5000
```

🔹 Step 4: Configure TLS for Secure HTTPS

To enable HTTPS for your Ingress, you need to create a TLS secret.

📌 Generate TLS Certificate (Self-Signed)

```
openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout
tls.key -out tls.crt -subj "/CN=example.com/O=example.com"
```

📌 **Create Kubernetes Secret**

```
kubectl create secret tls tls-secret --cert=tls.crt
--key=tls.key
```

🔹 Step 5: Apply Everything to the Cluster

Now, apply the deployments, services, and ingress:

```
kubectl apply -f frontend.yaml

kubectl apply -f backend.yaml

kubectl apply -f ingress.yaml
```

🔹 Step 6: Test the Ingress Setup

Once the Ingress is deployed, test it using `curl`:

```
curl -k https://example.com/api  # Should return backend
response

curl -k https://example.com/app  # Should return frontend
response

curl -k https://app.example.com  # Should return frontend
response

curl -k https://api.example.com  # Should return backend
response
```

Or, open a **web browser** and access:

- `https://app.example.com` (Frontend)
- `https://api.example.com` (Backend)
- `https://example.com/api` (Backend API)
- `https://example.com/app` (Frontend UI)

🚀 Final Summary

✔ **KOps Cluster + Ingress Controller Installed**

✔ **Frontend & Backend Services Deployed**

✔ **Host-based & Path-based Routing Configured**

✔ **TLS/HTTPS Secured**

✔ **Ingress Successfully Implemented & Tested**

With this setup, your **Kubernetes applications are now securely exposed to the internet using Ingress!** 🎉

🔹 In Summary: Mastering Kubernetes Ingress

Kubernetes Ingress is a **powerful way to expose and manage external access** to your services efficiently. Before Ingress, we relied on NodePort and LoadBalancer services, which were either inflexible or expensive. **Ingress solves these challenges** by providing a centralized way to route traffic, apply security policies, and enable load balancing—all within Kubernetes.

In this example, we covered:

✅ **Installing an Ingress Controller** in a Kubernetes cluster (using kOps).

✅ **Deploying and Exposing Services** (Frontend & Backend) inside the cluster.

✅ **Using Host-Based and Path-Based Routing** to direct traffic efficiently.

✅ **Enabling TLS/HTTPS for Secure Access** using a self-signed certificate.

✅ **Testing & Validating Ingress Rules** to ensure smooth traffic flow.

With Ingress, you **simplify traffic management, improve security, and optimize resource usage**, making it an essential tool for production-grade Kubernetes environments! 🚀

🔹 Wrapping Up: Thank You for Being Part of This Journey!

And that's a wrap on **Kubernetes Ingress!** 🎉 We've explored everything from why Ingress is needed, how it works, key features, configuration, and real-world scenarios to implement it effectively. **Ingress is a powerful tool** that simplifies routing, enhances security, and optimizes traffic management in Kubernetes clusters. I hope this guide gave you a **clear and practical** understanding of how to leverage Ingress in your Kubernetes environment. 🚀

But this is just one step in our **Kubernetes learning journey!** 💡 There's so much more to uncover, and I'm excited to continue sharing **daily hands-on tasks, practical insights, and deep dives into key Kubernetes concepts** with all of you.

🙏 **A huge thank you for supporting me throughout this journey!** Your engagement, encouragement, and enthusiasm keep me motivated to bring more valuable content every day. Your likes, comments, and shares make this series even more special! 💙

This journey doesn't stop here—**there's more to come!** 🔥 If you've been finding these posts helpful, **make sure to follow me** to stay updated with more **Kubernetes deep dives, best practices, and real-world hands-on tasks!**

Let's keep learning, growing, and building together. See you in the next one! 🚀✨

Stay tuned, stay curious, and keep exploring Kubernetes! 🔥💡