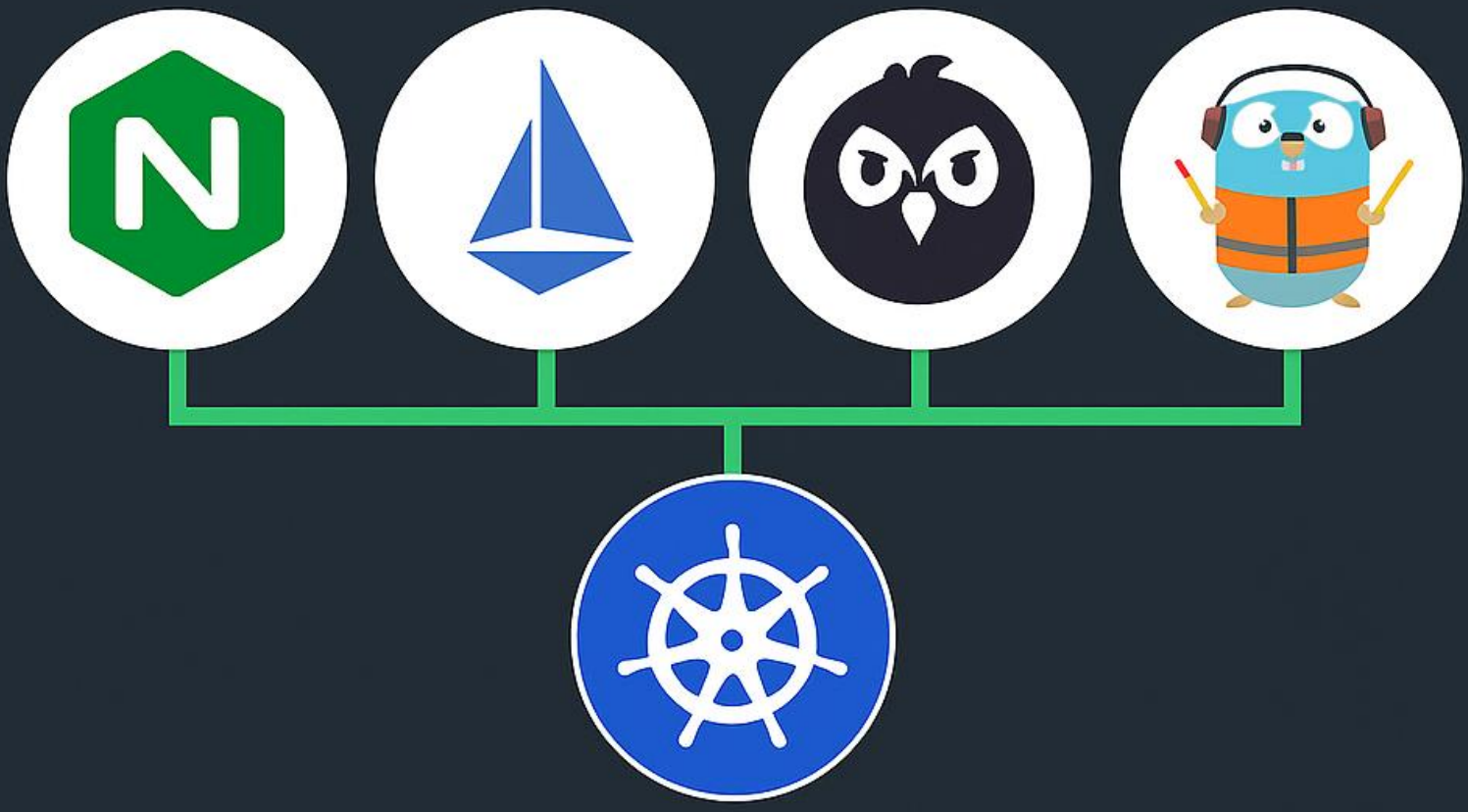
Ingress Controller



NGINX

Venkatesh Jilakarra

What is Ingress?

Ingress is a Kubernetes API object that defines rules for routing external HTTP(S) traffic to internal Kubernetes services.

It functions as a smart router that decides how requests reach your services based on URL, hostname, headers, etc.

Content-Based Routing

- **Host-based routing** – Routes based on domain names like contactapp.example.com or homeapp.example.com.
- **Path-based routing** – Routes based on URIs such as /login or /payment.

What is an Ingress Controller?

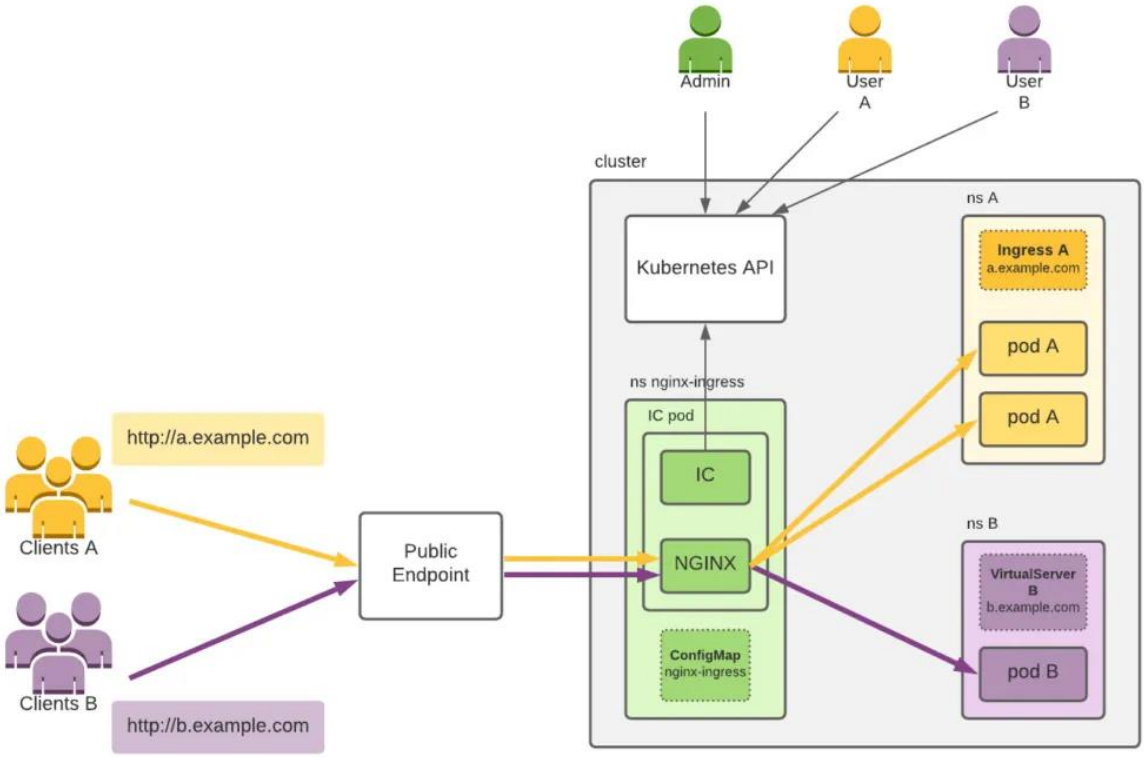
An Ingress Controller is an application that runs inside the Kubernetes cluster and implements HTTP load balancing based on Ingress rules.

It configures reverse proxy tools (like NGINX, HAProxy, Traefik, or Envoy) using the defined rules.

Think of it this way:

- Ingress = Rules
- Ingress Controller = The component that enforces those rules

---



Why Use an Ingress Controller?

- Centralized route management for multiple applications
- Handles TLS/SSL certificates (SSL termination)
- Enables host/path-based traffic routing
- Supports authentication and rate limiting (using annotations or plugins)

Popular Ingress Controllers

- NGINX Ingress Controller (official and community versions)
- Traefik
- HAProxy
- Kong
- Istio Gateway (used in service meshes)

Ingress Controller Workflow

1. A client sends a request to a domain or IP.
2. The request reaches the NGINX Ingress Controller Pod.
3. The controller checks the Ingress resource for the relevant routing rule.
4. The request is sent to the corresponding Kubernetes Service.
5. The service load balances the request to the appropriate backend Pods.

Ingress Controller Components

- **IC Process**: Configures NGINX using cluster resources.
- **NGINX Master Process**: Manages the worker processes.
- **NGINX Worker Processes**: Handle the actual client traffic and load balancing.

Deployment Overview

To deploy the NGINX Ingress Controller (by NGINX Inc.), you follow these high-level steps:

1. Obtain the container image of the controller.
2. Install necessary tools like Git.
3. Clone the official Kubernetes Ingress repository for NGINX.
4. Create a namespace and service account to isolate the controller.

5. Apply RBAC roles and bindings for permission management.
6. Set up a default TLS secret.
7. Apply a ConfigMap to define NGINX behavior.
8. Apply Custom Resource Definitions (CRDs) for advanced functionality.
9. Deploy the NGINX Ingress Controller Pod.
10. Verify the controller is running.

Why CRDs Are Needed

The standard Ingress resource only supports basic HTTP/HTTPS routing. To implement features like:

- Rate limiting
- IP whitelisting
- Custom header manipulation
- Path rewrites
- Web Application Firewall (WAF)

you need Custom Resource Definitions (CRDs), which extend Kubernetes' capabilities beyond the basic Ingress API.

Handling IngressClass Errors

A common mistake is assigning the wrong controller name in the IngressClass definition. The official NGINX Inc. controller expects a specific controller value.

To fix this:

- Define the correct IngressClass with the expected controller name.
- Restart the NGINX Ingress Controller so it picks up the new configuration.

Exposing the Ingress Controller

In a bare-metal or local Kubernetes setup, you typically expose the Ingress Controller using a NodePort service so it can accept traffic from outside the cluster.

Deploying an Application Behind Ingress

Deployment Overview

- A shared emptyDir volume is defined.
- An init container writes a welcome message (HTML file) into the volume.
- The main NGINX container serves this content by mounting the same volume.

Service Definition

- A ClusterIP service is created to expose the NGINX container internally.

Ingress Configuration for Routing

An Ingress resource is created with host-based routing:

- Requests to homeapp.hawkstack.com go to one service.

- Requests to contactapp.hawkstack.com go to another service.

The configuration includes a rewrite rule that ensures all paths (e.g., /login, /products) are redirected to the root (/), making it compatible with single-page applications or minimal servers.

Testing the Web Pages

1. Check the NodePort value for the Ingress Controller.
2. Edit the /etc/hosts file to map the test domains to your cluster's IP.
3. Access the test application in a browser using the defined hostnames and port numbers.

Real-World Scenarios

1. Multi-Domain Application Hosting

Scenario:
A company wants to host multiple microservices (e.g., accounts, products, and orders) under different subdomains on the same Kubernetes cluster.

Solution:
Use a single NGINX Ingress Controller to route traffic based on the host:

- accounts.example.com → Accounts service
- products.example.com → Product service
- orders.example.com → Order service

This reduces the need for multiple external IPs or load balancers.

2. Blue-Green Deployment with Canary Routing

Scenario:
You want to deploy a new version of a microservice (v2) but want to send only 10% of traffic to it, while 90% goes to the current version (v1).

Solution:
Use CRDs and annotations provided by NGINX Ingress Controller to perform canary deployments:

- Split traffic between v1 and v2
- Gradually increase traffic to v2 while monitoring errors and performance

---

3. Centralized SSL Termination

Scenario:
Your cluster runs 10 different applications that all use HTTPS. Managing certificates for each one separately is inefficient.

Solution:
Offload TLS termination to the Ingress Controller:

- Store TLS certificates in Kubernetes secrets
- Let NGINX handle decryption
- Forward plain HTTP to the internal services

---

4. Staging and Production Environments on the Same Cluster

Scenario:
You have app.example.com for production and app.staging.example.com for staging, both running in the same cluster.

Solution:
Use hostname-based routing with different Ingress resources pointing to the appropriate namespace or service. This ensures isolation while using shared infrastructure.

---

5. Secure External Access to Internal APIs

Scenario:
You want to expose an internal API only to a specific IP range or require authentication for external access.

Solution:
Configure Ingress annotations for:

- IP whitelisting
- Basic authentication
- Rate limiting
- Custom headers

This allows fine-grained access control at the ingress level.