Kubernetes Ingress, Service Mesh (Istio) & Mtls

Kubernetes Ingress và Ingress Controller

Ingress là một tập hợp các quy tắc định tuyến HTTP và HTTPS giúp chuyển hướng lưu lượng từ bên ngoài vào các **Service nội bộ** trong cụm Kubernetes.

Nó hoạt động như một lớp điều phối traffic ở **Layer 7 (Application Layer)**.

Ingress cung cấp một cách tiếp cận hiệu quả hơn và linh hoạt hơn so với việc tạo một LoadBalancer riêng biệt cho mỗi dịch vụ.

Tiêu chí	**LoadBalancer**	**Ingress**
Cơ chế hoạt động	Tạo một IP tĩnh và route traffic đến Service	Sử dụng một Ingress Controller để route traffic dựa trên domain/path
Số lượng public IP cần	Một IP/public LoadBalancer cho mỗi Service	Một IP duy nhất cho toàn bộ cluster
Khả năng quản lý route	Không hỗ trợ quản lý route chi tiết	Hỗ trợ route chi tiết theo path/domain
Tính linh hoạt	Thấp	Cao

Ingress Controller

- Ingress chỉ là một resource, để nó hoạt động cần có **Ingress Controller**, một chương trình chạy trong cluster có nhiệm vụ xử lý các đối tượng Ingress.
- Có nhiều loại Controller khác nhau như:
    - **NGINX Ingress Controller** (phổ biến nhất)

- **Traefik**
- **HAProxy**
- **Istio Gateway** (nếu sử dụng service mesh)

Ingress Controller sẽ:

- Nghe và áp dụng các quy tắc từ resource Ingress
- Tự động cấu hình lại để phản ánh các thay đổi routing

Cấu trúc YAML của Ingress

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-ingress
spec:
  rules:
    - host: myapp.example.com
      http:
        paths:
          - path: /accounts
            pathType: Prefix
            backend:
              service:
```

name: accounts-service

port:

number: 8080

=> Quy tắc trên định tuyến tất cả request đến myapp.example.com/accounts về service accounts-service cổng 8080.

Ingress vs Spring Cloud Gateway

- Trong microservices với Spring Boot, một số hệ thống sử dụng **Spring Cloud Gateway** để làm API Gateway.
- Tuy nhiên, khi triển khai lên **Kubernetes**, **Ingress có thể đảm nhiệm vai trò tương tự**.
- Trong môi trường production, người ta thường tách rõ:
    - **Ingress (do DevOps quản lý)**: kiểm soát entrypoint cho traffic từ outside vào cluster
    - **API Gateway (do dev quản lý)**: kiểm soát routing, auth, rate limit… ở layer logic ứng dụng.

Tầm quan trọng trong triển khai thực tế

- Ingress giúp **giảm chi phí cloud** bằng cách dùng chung một LoadBalancer.
- Giúp quản lý tập trung các quy tắc routing.
- Tích hợp dễ dàng với các chứng chỉ TLS (HTTPS) và công cụ giám sát traffic.

Lợi ích của Kubernetes Ingress và các loại traffic mà nó xử lý

Kubernetes Ingress là một thành phần quan trọng trong việc triển khai các hệ thống microservices trên Kubernetes. Nó đóng vai trò là **cổng vào duy nhất** (single entry

point) cho toàn bộ cluster, từ đó giúp quản lý lưu lượng truy cập từ bên ngoài vào các dịch vụ nội bộ dễ dàng hơn.

Các lợi ích chính của Kubernetes Ingress

- **Single Entry Point**
  Ingress cho phép cấu hình một điểm vào duy nhất cho nhiều dịch vụ khác nhau trong Kubernetes cluster. Điều này giúp đơn giản hóa việc quản lý truy cập bên ngoài và tương tự như vai trò của Spring Cloud Gateway trong các hệ thống Spring microservices.

- **TLS/SSL Termination**
  Ingress hỗ trợ chấm dứt các kết nối mã hóa TLS/SSL tại lớp biên (edge server). Khi dữ liệu từ client gửi vào, Ingress sẽ giải mã tại điểm này và chuyển tiếp dưới dạng HTTP không mã hóa trong nội bộ cluster, nhằm giảm chi phí hiệu năng vì không cần mã hóa/giải mã bên trong cluster (nơi an toàn hơn).

- **Routing (định tuyến)**
  Ingress hỗ trợ:

  - **Path-based routing**: Ví dụ, các yêu cầu đến /app1 sẽ được chuyển đến service A, còn /app2 đến service B.
  - **Host-based routing**: Ví dụ, các yêu cầu đến app1.example.com sẽ đến một service cụ thể tương ứng.

- **Load Balancing**
  Ingress có khả năng phân phối tải đến các pod khác nhau của cùng một service, thông qua service type ClusterIP trong nội bộ Kubernetes.

- **Annotations (chú thích)**
  Ingress cho phép cấu hình nâng cao như thêm header tùy chỉnh, viết lại đường

dẫn, hoặc tích hợp xác thực/ủy quyền (authentication & authorization) bằng cách sử dụng annotations.

So sánh với Spring Cloud Gateway

Cả Kubernetes Ingress và Spring Cloud Gateway đều cung cấp các khả năng tương đương như:

- Định tuyến theo path/host
- Cân bằng tải
- Chấm dứt TLS/SSL
- Xử lý các yêu cầu cắt ngang (cross-cutting concerns)

Việc lựa chọn công cụ nào phụ thuộc vào quyết định của kiến trúc sư hệ thống hoặc trưởng nhóm kỹ thuật (project leadership). Nếu tổ chức chọn Spring Cloud Gateway thì developer sẽ tích hợp theo hướng đó, hoặc nếu chọn Kubernetes Ingress thì sử dụng theo cách tương ứng.

So sánh Ingress Controller và Service Type LoadBalancer

Cả hai đều có thể được dùng để expose service ra ngoài, nhưng:

- **Ingress Controller**: cung cấp khả năng định tuyến nâng cao, linh hoạt hơn (suitable cho hệ thống lớn, phức tạp).
- **LoadBalancer**: phù hợp cho tổ chức nhỏ với số lượng microservice ít và yêu cầu đơn giản.

Các loại traffic mà Ingress xử lý

- **Ingress Traffic**: Lưu lượng đi từ bên ngoài vào trong cluster.
- **Egress Traffic**: Lưu lượng đi từ trong cluster ra ngoài.

- **North-South Traffic**: Lưu lượng đi vào hoặc ra khỏi cluster (khác với East-West traffic là giao tiếp giữa các pod bên trong cluster).

Tổng quan về Traffic trong Kubernetes

- **North-South traffic**: Là lưu lượng giữa client (ngoài cluster) và các service trong cluster, thường được quản lý bởi **Ingress Controller**.
- **East-West traffic**: Là lưu lượng giữa các **microservices nội bộ** trong cùng một Kubernetes cluster.

Cần ai để quản lý East-West traffic?

- Câu trả lời: **Service Mesh**
- **Service Mesh** là một **lớp hạ tầng chuyên biệt** để quản lý toàn bộ giao tiếp giữa các microservices.

Service Mesh làm được gì?

- Đảm bảo **giao tiếp nội bộ (east-west)** được:
    - **Bảo mật (secure)**
    - **Đáng tin cậy (reliable)**
    - **Có thể quan sát được (observable)**
- Các khả năng chính của Service Mesh:
    - **Service Discovery**
    - **Load Balancing**
    - **Circuit Breaking & Fault Tolerance**
    - **Metrics & Tracing**

- **Security**

So sánh với cách tiếp cận hiện tại trong khoá học

- Trước khi dùng Service Mesh, các chức năng trên đã được xử lý bằng:
  - **Eureka, Spring Cloud Gateway, Resilience4j, Prometheus, Grafana**
- Những logic phi nghiệp vụ (non-business logic) như security, resilience, metrics,... được **viết thủ công** trong từng service.

Nhược điểm của cách thủ công

- Tăng độ phức tạp: mỗi microservice phải tự xử lý phần **non-functional requirements**.
- Nếu cần thay đổi về logging, tracing, security,… phải sửa **tất cả** service.
- Lệch trọng tâm của developer khỏi **business logic**.

Khi nào nên dùng Service Mesh?

- Khi tổ chức đủ **kỹ năng DevOps** và **ngân sách**.
- Nếu không đủ nguồn lực, developers sẽ phải tiếp tục đảm nhiệm các phần logic này thủ công.

Lời khuyên từ giảng viên

- Là developer, bạn nên:
  - **Hiểu rõ cả hai hướng**: dùng Service Mesh và không dùng.
  - Sẵn sàng cho nhiều tình huống khác nhau tùy vào tổ chức, ngân sách và mức độ quan trọng của service.

Service Mesh gồm hai thành phần chính

- **Data Plane** (mặt phẳng dữ liệu):
    - Chịu trách nhiệm **định tuyến traffic** giữa các microservice.
    - Được triển khai qua các **proxy (sidecar containers)** như **Envoy**.
    - Mỗi microservice pod đều có một sidecar container để **chặn và xử lý request/response** trước khi đến main container.
- **Control Plane** (mặt phẳng điều khiển):
    - Quản lý và cấu hình các proxy trong data plane.
    - Tự động tạo sidecar container khi một pod mới được tạo.
    - Gồm các thành phần như: **API, Service Discovery, Configuration Management**.

Cách hoạt động trong Kubernetes

- Mỗi pod chứa container business logic (ví dụ: accounts, loans, cards) sẽ được **inject thêm một Envoy proxy** khi dùng Istio.
- **Control Plane của Istio** chịu trách nhiệm inject, giám sát và cập nhật các envoy proxy này.
- Tất cả các proxy tạo thành **Istio Data Plane**.

Một số Service Mesh phổ biến

- **Istio** (được ưa chuộng nhất)
- **Linkerd**
- **Consul**

- **Kong**
- **AWS App Mesh**
- **Azure Service Mesh**

Tại sao developer nên biết về Service Mesh?

- Không cần biết sâu như một DevOps/Platform engineer.
- Nhưng cần hiểu **Service Mesh là gì, khả năng của nó, và cách nó hoạt động**.
- Có thể bị hỏi trong phỏng vấn: "Bạn có dùng Service Mesh trong môi trường production không?"

Service Mesh và bảo mật

- Hỗ trợ **Mutual TLS (mTLS)** để bảo vệ **giao tiếp giữa các service nội bộ**.
- mTLS giúp xác thực cả hai đầu (client & server) để tăng bảo mật.

mTLS và phân tích chi tiết giao thức TLS

Trong hệ thống microservices hiện đại, bảo mật trong quá trình giao tiếp giữa các dịch vụ là một yêu cầu bắt buộc. Một trong những giải pháp bảo mật hiệu quả là sử dụng **mTLS (Mutual TLS)** – một biến thể của TLS (Transport Layer Security), cho phép cả hai bên trong quá trình giao tiếp xác thực lẫn nhau thông qua chứng chỉ số. Trước khi hiểu mTLS, cần nắm rõ nguyên lý hoạt động của TLS.

TLS là giao thức bảo mật kế thừa từ SSL (Secure Socket Layer) – vốn đã lỗi thời – và hiện đang được sử dụng phổ biến trong các giao tiếp qua Https. Mục tiêu của TLS là mã hóa dữ liệu truyền giữa client và server nhằm ngăn chặn các cuộc tấn công nghe lén (eavesdropping). Trong quá trình giao tiếp, client (chẳng hạn trình duyệt) yêu cầu server cung cấp chứng chỉ số (certificate). Chứng chỉ này phải được cấp bởi

một tổ chức tin cậy gọi là Certificate Authority (CA). Trình duyệt sẽ kiểm tra tính hợp lệ của chứng chỉ và chỉ thiết lập kết nối nếu chứng chỉ được xác thực.

Tuy nhiên, trong môi trường microservices, không có trình duyệt nào làm nhiệm vụ xác thực. Các dịch vụ giao tiếp trực tiếp với nhau thông qua API, và nếu chỉ dùng TLS thì chỉ có phía server được xác thực, client thì không. Điều này dẫn đến rủi ro bảo mật, ví dụ như một dịch vụ giả mạo có thể cố tình gọi đến dịch vụ khác mà không được phép. Vì vậy, **mTLS** ra đời nhằm giải quyết vấn đề này bằng cách yêu cầu **cả client và server đều phải có chứng chỉ và xác thực lẫn nhau**.

mTLS thường được sử dụng trong mô hình bảo mật **Zero Trust** – tức là không tin cậy bất kỳ thực thể nào mặc định, kể cả khi nó nằm trong cùng một mạng nội bộ hay cụm Kubernetes. Mỗi yêu cầu đều phải được xác minh danh tính và cấp quyền cụ thể. Điều này giúp tăng cường bảo mật nội bộ, ngăn chặn các hành vi truy cập trái phép hay các dịch vụ độc hại tiềm ẩn bên trong hệ thống.

Cách hoạt động của mTLS (Mutual TLS)

Trong các hệ thống sử dụng trình duyệt, TLS là đủ vì chỉ máy chủ cần xác thực. Tuy nhiên, trong các môi trường nội bộ như **hệ thống microservices trong Kubernetes**, nơi các dịch vụ cần xác thực lẫn nhau để đảm bảo an toàn, mTLS là lựa chọn phù hợp. Đây là một phần quan trọng trong chiến lược bảo mật "zero trust", nơi không có thành phần nào được tin cậy sẵn.

Tuy nhiên, việc sử dụng bên thứ ba để cấp phát chứng chỉ như TLS truyền thống là không khả thi trong môi trường mTLS. Lý do là vì:

- Các pod hoặc container có thể bị xoá và tạo lại liên tục.
- Việc yêu cầu chứng chỉ từ CA bên ngoài tốn thời gian và chi phí.
- Phải quản lý hàng loạt chứng chỉ sẽ phức tạp và không hiệu quả.

Do đó, trong mTLS, **tổ chức hoặc hệ thống sẽ tự đóng vai trò Certificate Authority (CA)**. Trong môi trường Kubernetes, điều này thường được thực hiện thông qua **service mesh** như Istio. Service mesh sẽ tự động cấp phát chứng chỉ cho các microservice thông qua sidecar proxy.

Cụ thể, khi microservice A (ví dụ: Accounts) muốn gọi microservice B (Loans), dữ liệu HTTP sẽ được chuyển qua proxy. Proxy của A sẽ bắt đầu quá trình TLS handshake với proxy của B, yêu cầu B cung cấp chứng chỉ. Proxy của A xác thực chứng chỉ này với CA nội bộ (do service mesh quản lý). Sau khi xác thực thành công, dữ liệu sẽ được mã hoá và gửi đến B thông qua proxy của B, rồi được giải mã và chuyển đến container chính.

Quan trọng là các microservice như Accounts và Loans **không hề biết đến quy trình xác thực và mã hoá này**, vì toàn bộ được xử lý "đằng sau hậu trường" bởi các sidecar proxy.

Nhờ mTLS, chúng ta không chỉ mã hoá dữ liệu truyền mà còn kiểm soát được **dịch vụ nào được phép giao tiếp với dịch vụ nào**, giảm thiểu rủi ro bị giả mạo hay truy cập trái phép trong hệ thống.