**istio 란?**

오픈 소스 Service Mesh 솔루션

**Service Mesh:**

마이크로서비스 간 통신, 보안, 모니터링, 트래픽 관리 등을 담당하는 네트워크 인프라 계층

서비스(Pod) 간 통신을 직접 하지 않고, 프록시(Sidecar Proxy)를 통해 수행

컨테이너는 프록시를 통해 다른 서비스(Pod)와 통신하며, 컨테이너끼리는 직접 통신하지 않음.

**구성요소:**

\* 데이터 플레인: 아키텍처에서 실행되는 모든 Proxy, (네트워크 데이터를 처리하는 계층으로 이해)

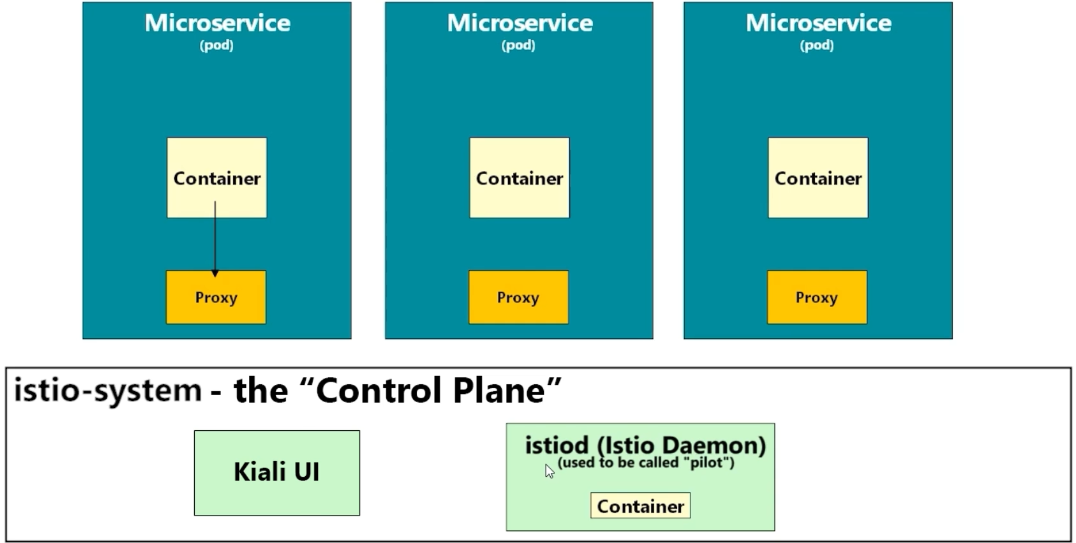
\* 컨트롤 플레인: Istio의 관리 및 정책 적용을 담당하는 istiod

**istiod(Istio Deamon):**

istio의 중앙 제어 컴포넌트이다.

Envoy 프록시 설정, 서비스 디스커버리, 인증, 정책 등을 관리 한다.

이전에는 pilot, citadel, galley 등 여러 컴포넌트로 나뉘었지만 지금은 istiod로 통합됐다.



**istiod 주요 기능**

| **기능** | **설명** |
| --- | --- |
| **xDS 서버** | Envoy 프록시에게 라우팅, 서비스 정보 등을 실시간 전달 |
| **서비스 디스커버리** | Kubernetes API 서버를 통해 서비스/Pod 정보를 가져옴 |
| **구성 관리** | VirtualService, DestinationRule 같은 Istio 리소스를 수집해 Envoy 설정 생성 |
| **보안 (Citadel)** | mTLS를 위한 인증서 발급 및 갱신 |
| **정책/모니터링 연계** | Telemetry, tracing 설정도 일부 관리 |

**쿠버네티스와 Istio의 관계:**

Istio는 “pod간 통신을 가능하게” 하는 도구가 아니라 “통신을 운영하기 쉽게” 하는 도구이다.

pod간 통신 자체는 k8s의 service만 이용해서도 가능하다.

**서비스 메시와 MSA와의 관계:**

MSA는 마이크로서비스 아키텍처이고, 서비스 메시는 그 운영을 자동화/표준화하는 보완 기술이다.

Istio는 서비스 메시 구현체 중 하나로, 서비스 간 통신, 보안, 라우팅 등을 담당한다.

**istio의 프록시의 정체 (Envoy 프록시):**

Istio는 데이터 플레인으로 Envoy 프록시를 사용한다.

Envoy는 Lyft가 개발한 고성능 L4/L7 프록시이며, Istio와는 독립적으로 개발되었다.

Envoy는 circuit breaking, retry, timeout 등의 기능을 내장하고 있다.

Jaeger(트레이싱)와 Kiali(모니터링)는 별도 도구이며, Envoy의 기능은 아니다.

**k8s Service는 "주소록(DNS)" 역할만 한다:**

애플리케이션은 reviews.default.svc.cluster.local 같은 FQDN으로 요청함

이 FQDN은 DNS를 통해 ClusterIP로 변환됨

그러나 Istio가 설치된 환경에서는 실제 트래픽은 ClusterIP를 지나지 않음

**실제 트래픽은 Envoy → Pod IP 간 직접 통신:**

각 Pod에 주입된 Envoy 프록시는 트래픽을 가로채어 처리하며, k8s DNS를 직접 조회하지 않고, Istiod로부터 해당 서비스의 엔드포인트(Pod IP) 목록을 전달받아 유지한다. 이 목록은 Istiod가 Kubernetes API를 통해 동기화하며, Envoy는 이를 기반으로 트래픽을 라우팅 한다.

**istiod가 중심이 되어 서비스 디스커버리 수행:**

istiod는 Kubernetes API 서버로부터 Endpoints 정보를 수집

이를 바탕으로 각 Envoy에 "이 FQDN은 어떤 Pod IP들과 매핑된다"는 데이터를 전파함

Envoy는 이 정보를 동기화 해두고, 이후 직접 Pod IP를 선택하여 요청을 전달함 (LB 포함)

**VirtualService가 없을 때,** **Istio 트래픽 흐름:**

1. 클라이언트 앱 → 서비스 도메인(FQDN)으로 요청
2. Envoy는 DestinationRule이나 VirtualService가 없으니,  
   그냥 FQDN 기준 기본 라우팅 수행할 것이다.
3. Envoy는 그 FQDN에 대응하는 Pod IP 목록을 istiod로부터 이미 알고 있음
4. Envoy는 그 중 하나를 골라 Round-Robin 방식 등으로 라우팅

**VirtualService 있을 때, istio 트래픽 흐름:**

1. 클라이언트 앱 → 서비스 도메인(FQDN)으로 요청
2. Envoy가 VirtualService 규칙 확인  
    → 라우팅 대상(FQDN 기준)을 결정
3. Envoy는 그 FQDN에 대응하는 Pod IP 목록을 istiod로부터 이미 알고 있음  
    → 실제로는 DNS 조회 없이 Pod IP를 선택
4. Envoy가 해당 Pod IP로 직접 요청 전송

**Istio에서도 k8s Service가 필요한 이유:**

Endpoint 정보의 기준이 되기 때문이다.

Istio의 Pilot은 Kubernetes API 서버에서

Service → Endpoints → Pod IP 목록을 조회함

즉, Service가 없으면 어떤 Pod들이 묶여 있는지 알 수 없음

FQDN(도메인 이름)을 정의하기 위해서도 필요

reviews.default.svc.cluster.local 같은 도메인은

Kubernetes Service 객체가 있어야 의미가 있음

VirtualService/DestinationRule의 기준도 Service이다.

Istio의 라우팅 리소스(VirtualService 등)도

Service 이름을 기준으로 설정됨

**kube-proxy vs envoy-proxy:**

kube-proxy는 노드 수준에서 단순 로드밸런싱만 해주고,

Envoy는 Pod 수준에서 L7 트래픽을 제어하고, 관찰·보안 기능까지 제공한다.

| **항목** | **kube-proxy (k8s)** | **Envoy (Istio-proxy)** |
| --- | --- | --- |
| 위치 | 노드 단위 | Pod 단위 (사이드카) |
| 계층(Layer) | L3/L4 (IP, 포트 기반) | L7 (HTTP, 헤더/경로 등 인식 가능) |
| 주요 기능 | ClusterIP로 로드밸런싱, NAT | 트래픽 라우팅, 정책, 관찰성, 보안 |
| 로드밸런싱 대상 | Pod (Service 뒤의) | Pod (Endpoint IP), 세분화된 Subset 가능 |
| 라우팅 제어 | 없음 (단순 round-robin 등) | VirtualService로 세밀한 제어 가능 |
| 트래픽 흐름 | iptables/IPVS 기반  → kube-proxy 경유 | Envoy → Pod IP 직접 요청 |
| 가시성/관찰성 | 낮음 (별도 설정 필요) | 높음 (mTLS, Telemetry 내장) |

**kube-proxy 기반 (기본 K8s):**

앱이 reviews.default.svc.cluster.local 요청

DNS → ClusterIP

트래픽이 kube-proxy를 거쳐 Pod IP로 전달 (iptables/netfilter 활용)

**envoy-proxy 기반 (Istio):**

앱은 똑같이 FQDN 요청

Envoy 프록시가 요청 가로챔

Envoy는 istiod로부터 받은 Pod IP 목록(캐싱됨) 기준으로 직접 라우팅

**istio 다운로드:**

<https://istio.io/latest/docs/setup/install/istioctl/>

| curl -L https://istio.io/downloadIstio | sh - cd istio-1.25.1 export PATH=$PWD/bin:$PATH |
| --- |

**istio 기본 설치 방법:**

| istioctl install -y |
| --- |

**설치 시 프로파일 지정:**

프로파일은 [링크](https://istio.io/latest/docs/setup/additional-setup/config-profiles/)에서 확인 가능하다.

| istioctl install --set profile=default -y |
| --- |

**istio 삭제 방법:**

| istioctl uninstall --purge -y kubectl delete namespace istio-system |
| --- |

**플러그인 설치:**

<https://istio.io/latest/docs/ops/integrations/>

아래는 Quick Start 설치 방법이고, 대규모 메시 용으로는 옵션 설정이 필요하다.

| kubectl apply -f https://raw.githubusercontent.com/istio/istio/release-1.25/samples/addons/grafana.yaml  kubectl apply -f https://raw.githubusercontent.com/istio/istio/release-1.25/samples/addons/kiali.yaml  kubectl apply -f https://raw.githubusercontent.com/istio/istio/release-1.25/samples/addons/jaeger.yaml  kubectl apply -f https://raw.githubusercontent.com/istio/istio/release-1.25/samples/addons/prometheus.yaml |
| --- |

**kaili의 경우 아래 yaml도 적용해줘야 한다.**

apiVersion: v1

kind: Secret

type: Opaque

metadata:

name: kiali

namespace: istio-system

labels:

app: kiali

data:

username: YWRtaW4= # admin을 base64로 인코딩

passphrase: YWRtaW4= # admin을 base64로 인코딩

**minikube 환경에서 istio 설치 시 주의할 점:**

적어도 4GB의 램이 할당되어 있어야 한다.

이미 실행되는 minikube가 있다면 아래 명령어로 중지 시킨다.

| minikube delete |
| --- |

그리고 아래 명령어로 실행 시킨다.

| minikube start --memory 4096 |
| --- |

**istio를 minikube 에 적용된지 확인하기:**

Istio 시스템 네임스페이스는 istio-system 이라

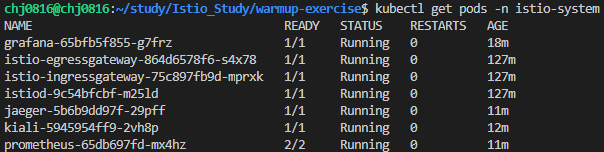
다음과 같이 네임스페이스를 지정해야 관련 리소스를 찾아볼 수 있다.

| kubectl get pods -n istio-system  kubectl get services -n istio-system |
| --- |

아래처럼 출력이 되는데, istiod가 있는 것을 확인할 수 있다.

기타 지원 pod는 다음과 같다.

* grafana: 대시보드
* prometheus: 매트릭
* kiali: 서비스 매쉬 시각화, 모니터링
* jagger: 분산 추적
* ingressgateway: inbound 트래픽 통제
* egressgateway: outbound 트래픽을 통제



**envoy proxy를 pod에 추가하는 방법:**

**1. yaml 파일의 pod에 직접 프록시 컨테이너를 추가하는 방법: 비추**

**2. yaml 파일을 건드리지 않고, istio에서 플래그를 설정하는 방법 : 추천**

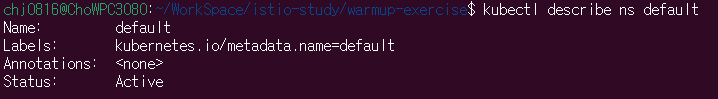
플래그를 설정하면, 특정한 네임스페이스에 속한

모든 pod에 프록시 컨테이너를 사이드카로 자동으로 추가한다.

**default 네임스페이스의 상세 정보를 출력:**

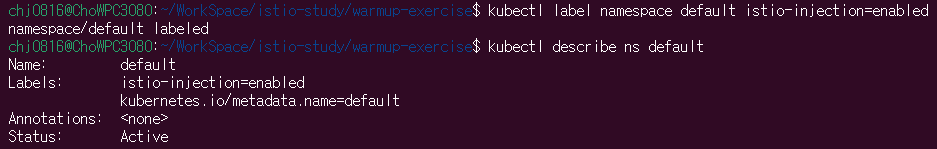
| kubectl describe ns default |
| --- |

**여기서 Lable은 리소스를 구분하거나 설정을 붙일 수 있는 메타 데이터다.**

****

**default 네임스페이스에 istio 주입 플래그를 킨다.**

| kubectl label namespace default istio-injection=enabled |
| --- |



**이를 yaml 파일로 보려면 다음과 같이 한다.**

| kubectl get ns default -o yaml |
| --- |

직접 플래그를 지정하지 않고 아래 파일을 적용해도 된다.

# label-default-namespace.yaml

apiVersion: v1

kind: Namespace

metadata:

labels:

istio-injection: enabled

name: default # default namespace 대상

**적용 유무 확인 방법:**

1. kubectl describe pod 명령어로 파드 별로 직접 까보는 방법
2. istioctl proxy-status 로 확인 하는 방법

**istio의 분산 추적 툴:**

1. Jaeger 분산 추적 툴:  
   Uber에서 개발함
2. Zipkin 분산 추적 툴:  
   Twitter에서 개발함

\* 주의 \*

Istio에서 Jaeger 대시보드를 열어보면

생각보다 많은 Span에 놀랄 수 있는데,

이는 Proxy 때문에 양이 불어난 것이다.

**헤더 확산(header propagation):**

분산 추적을 사용하려면 헤더 확산을 해야 한다.

request-id 같은 트레이싱 정보의 전달을 헤더 확산이라고 한다.

수신한 요청의 traceId(request-id)를

외부로 보내는 패킷에 그대로 담아 보내는 것을 헤더를 확산 시킨다고 표현한다.

이를 수작업으로 해야하는 것이 단점이지만,

언어에 따라 라이브러리를 잘 찾아보면 이를 자동화 해주는 것이 있다.

ex\_) java spring 환경에서의 fegin

통신을 할 때, traceId(request-id)를 유지할 수 있게 패킷에 넣어주지 않으면,

request-id가 없어 새로 id를 할당 할 것이고 새로운 trace가 만들어진다.

trace는 traceId(request-id) 같은 span들의 집합이다.

**모니터링:**

kiali, Grafana, Jaeger를 통해 할 수 있다.

kiali에 graph로 잘 보이게 할라면, metadata의 label에 app, version을 추가하는게 좋다.

그러면 app graph, versioned app graph를 조금 더 자세히 볼 수 있다.

**트래픽 관리**

**VirtualService:**

라우팅 규칙을 정의하는 리소스.

요청이 들어왔을 때, 어떤 서비스로 보낼지 결정.

HTTP path, header, query 등 조건에 따라 다른 서비스 또는 버전으로 라우팅 가능.

여러 버전이 있을 때 가중치 기반 트래픽 분배도 가능.

virtualService는 옵셔널한 리소스다.

k8s의 service와는 완전 다르다.

k8s의 service는 dns를 이용해서 포드의 ip 주소를 받아오는 역할이고,

virtualService는 istiod에 의해 관리되고, 트래픽 관리를 위해 사용한다.

virtualService가 k8s의 Serivce를 대체하는 역할이 아니다.

| kind: VirtualService  apiVersion: networking.istio.io/v1alpha3  metadata:  # 이 VirtualService 리소스의 이름 (아무 이름이나 가능)  name: a-set-of-routing-rules-we-can-call-this-anything  namespace: default  spec:  hosts:  # 라우팅 규칙을 적용할 K8S 서비스의 FQDN  - fleetman-staff-service.default.svc.cluster.local  http:  - route:  - destination:  # 트래픽이 향할 대상 서비스의 DNS 이름  host: fleetman-staff-service.default.svc.cluster.local  subset: safe-group # DestinationRule에서 정의한 subset 이름  weight: 90  - destination:  # 트래픽이 향할 대상 서비스의 DNS 이름  host: fleetman-staff-service.default.svc.cluster.local  subset: risky-group # DestinationRule에서 정의한 subset 이름  weight: 10 |
| --- |

**만들어진 VirtualService 조회:**

| kubectl get virtualservices // 또는  kubectl get vs |
| --- |

**DestinationRule:**

라우팅 된 요청을 어떻게 처리할지 정의하는 리소스.

어떤 버전(subset)이 있는지 정의

로드밸런싱 방식 설정 (round robin, least request 등)

mTLS, 연결 풀, 해시 기반 라우팅 등 세부 제어

| kind: DestinationRule # 각 subset에 포함될 pod 그룹을 정의  apiVersion: networking.istio.io/v1alpha3  metadata:  # 이 리소스의 이름 (아무 이름이나 가능)  name: grouping-rules-for-our-photograph-canary-release  namespace: default  spec:  host: fleetman-staff-service.default.svc.cluster.local # 대상 서비스의 FQDN  subsets:  - labels: # Selector: 아래 라벨을 가진 pod를 이 subset에 포함  version: safe # "version: safe" 라벨을 가진 pod  name: safe-group  - labels:  version: risky # "version: risky" 라벨을 가진 pod  name: risky-group |
| --- |

**만들어진 DestinationRule 조회:**

| kubectl get destinationrules // 또는 kubectl get dr |
| --- |

+

VirtualService와 DestinationRule은 보통 같이 쓰이지만 반드시 한 세트는 아니다.

VirtualService, DestinationRule 이름을 마음대로 지으면 kiali에서 변경 불가능 하다.

kiali에서 VirtualService, DestinationRule 의 유효성 검사에 대한 것을 확인할 수 있다.

**Session Stickiness (세션 고정) 설정**

DestinationRule수준에서의 고정은 가능하나,

VirtualService 수준에서의 고정은 불가능하다.

예를 들어 VirtualService에서 가중치 기반으로 라우팅을 해버리면,

그 뒤에서 DestinationRule에서 아무리 Session Stickness를 해도 세션이 고정되지 않는다.

DestinationRule이 적용되는 그 앞단에서 고정된 라우팅이 안되기 때문이다.

<https://istio.io/latest/docs/reference/config/networking/destination-rule/>

kind: VirtualService

apiVersion: networking.istio.io/v1alpha3

metadata:

# 이 VirtualService의 이름 (임의 지정 가능)

name: a-set-of-routing-rules-we-can-call-this-anything

namespace: default

spec:

hosts:

# 라우팅 규칙을 적용할 K8S 서비스의 DNS 이름

- fleetman-staff-service.default.svc.cluster.local

http:

- route:

- destination:

# 라우팅 대상 서비스 DNS 이름

host: fleetman-staff-service.default.svc.cluster.local

# DestinationRule에서 정의한 subset 이름

subset: all-staff-service-pods

---

kind: DestinationRule

apiVersion: networking.istio.io/v1alpha3

metadata:

# 이 DestinationRule의 이름 (임의 지정 가능)

name: grouping-rules-for-our-photograph-canary-release

namespace: default

spec:

host: fleetman-staff-service # 대상 서비스 이름

trafficPolicy:

loadBalancer:

consistentHash:

# 클라이언트 IP를 기반으로 요청을 해시하여 같은 인스턴스로 보내도록 설정

useSourceIp: true

subsets:

- labels: # 라벨 셀렉터 (이 조건에 맞는 pod들만 이 subset에 포함됨)

app: staff-service # 라벨이 "app=staff-service"인 pod들 대상

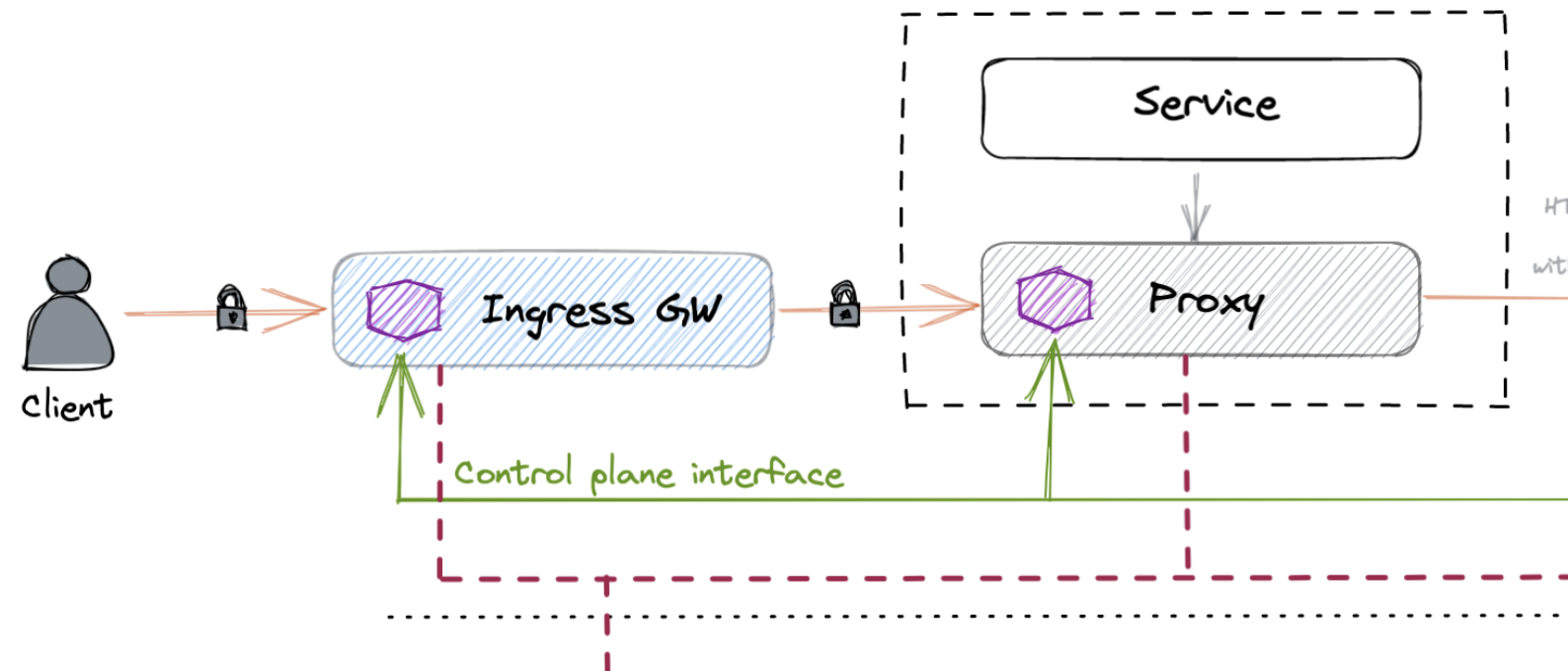
글 name: all-staff-service-pods # subset 이름

**Gateway:**

맨 앞단에서 바로 서버에 접근할 때는 k8s의 프록시를 타기 때문에,

envoy 프록시를 타지 않게 되고 그로 인해 VirtualService, DestinationRule의 영향을 받지 않는다.

맨 앞에 Gateway 역할을 하는 pod를 두어, envoyProxy를 타게 한다.



istio를 설정하면 기본적으로 istio-ingressgateway 관련 pod와 service가 존재한다.

이를 사용하면 되는데, gateway가 어느 port를 수신할 지 등을 설정해주면 된다.

개인적인 생각으로는 ingress gateway는 hpa로 구성하는게 좋을 것 같다.

\* 주의: ingressgateway 서비스의 열려있는 port로 클라가 접근해야 gateway를 탄다.

apiVersion: networking.istio.io/v1alpha3

kind: Gateway

metadata:

name: ingress-gateway-configuration

spec:

selector:

istio: ingressgateway

servers:

- port:

number: 80

name: http

protocol: HTTP

hosts:

- "\*" # 실제 운영환경에선 FQDN 사용을 추천

그리고 VirtualService 사용 시, 어떤 gateway를 따를지 지정해주면 된다.

apiVersion: networking.istio.io/v1alpha3

kind: VirtualService

metadata:

name: fleetman-webapp

namespace: default

spec:

hosts:

- "\*" # Gateway에서 넘어온 host 헤더와 매칭됨

gateways:

- ingress-gateway-configuration # 이 gateway로 부터 오는 것만 처리함.

http:

...

**Gateway를 이용한 서브 디렉토리 라우팅 (Prefix 라우팅):**

apiVersion: networking.istio.io/v1alpha3

kind: Gateway

metadata:

name: ingress-gateway-configuration

spec:

selector:

istio: ingressgateway

servers:

- port:

number: 80

name: http

protocol: HTTP

hosts:

- "\*"

---

apiVersion: networking.istio.io/v1alpha3

kind: VirtualService

metadata:

name: fleetman-webapp

namespace: default

spec:

hosts:

- "\*"

gateways:

- ingress-gateway-configuration

http:

- match:

- uri: # IF

prefix: "/experimental"

- uri: # OR

prefix: "/canary"

route: # THEN

- destination:

host: fleetman-webapp

subset: experimental

- match:

- uri :

prefix: "/"

route:

- destination:

host: fleetman-webapp

subset: original

**Gateway를 이용한 서브 도메인 라우팅:**

apiVersion: networking.istio.io/v1alpha3

kind: Gateway

metadata:

name: ingress-gateway-configuration

spec:

selector:

istio: ingressgateway

servers:

- port:

number: 80

name: http

protocol: HTTP

hosts:

- "\*.fleetman.com"

- "fleetman.com"

---

apiVersion: networking.istio.io/v1alpha3

kind: VirtualService

metadata:

name: fleetman-webapp

namespace: default

spec:

hosts:

- "fleetman.com"

gateways:

- ingress-gateway-configuration

http:

- route:

- destination:

host: fleetman-webapp

subset: original

---

apiVersion: networking.istio.io/v1alpha3

kind: VirtualService

metadata:

name: fleetman-webapp-experiment

namespace: default

spec:

hosts:

- "experimental.fleetman.com"

gateways:

- ingress-gateway-configuration

http:

- route:

- destination:

host: fleetman-webapp

subset: experimental

**헤더 매칭**

이를 응용하여 다크 배포 가능하다.

다크 배포: 새 기능이나 코드를 프로덕션 환경에 배포하되, 실제 사용자에게는 숨겨두는 방식

apiVersion: networking.istio.io/v1alpha3

kind: Gateway

metadata:

name: ingress-gateway-configuration

spec:

selector:

istio: ingressgateway

servers:

- port:

number: 80

name: http

protocol: HTTP

hosts:

- "\*"

---

kind: VirtualService

apiVersion: networking.istio.io/v1alpha3

metadata:

name: fleetman-webapp

namespace: default

spec:

hosts:

- "\*"

gateways:

- ingress-gateway-configuration

http:

- match:

- headers: # IF

my-header:

exact: canary

route: # THEN

- destination:

host: fleetman-webapp

subset: experimental

- route: # Default

- destination:

host: fleetman-webapp

subset: original

**결함 주입**

[링크](https://istio.io/latest/docs/reference/config/networking/virtual-service/#HTTPFaultInjection:~:text=to%20the%20upstream.-,HTTPFaultInjection,-HTTPFaultInjection%20can%20be)

abort 패턴: fleetman-vehicle-telemetry 서비스로의 모든 요청을 실패 시킴.

kind: VirtualService

apiVersion: networking.istio.io/v1alpha3

metadata:

name: fleetman-vehicle-telemetry

namespace: default

spec:

hosts:

- fleetman-vehicle-telemetry

http:

- fault:

abort:

httpStatus: 503

percentage:

value: 100 # 0.0 ~ 100.0

route:

- destination:

host: fleetman-vehicle-telemetry

delay 패턴: fleetman-vehicle-telemetry의 모든 요청을 지연을 지연 시킴.

kind: VirtualService

apiVersion: networking.istio.io/v1alpha3

metadata:

name: fleetman-vehicle-telemetry

namespace: default

spec:

hosts:

- fleetman-vehicle-telemetry

http:

- fault:

delay:

fixedDelay: 5s

percentage:

value: 100 # 0.0 ~ 100.0

route:

- destination:

host: fleetman-vehicle-telemetry

**서킷 브레이커 (Circuit Breaker)**

서킷 브레이커 관련 설명은 32) MSA 문서 참고.

istio에서는 서킷 브레이커를 detinationRule을 통해 사용할 수 있다.

apiVersion: networking.istio.io/v1alpha3

kind: DestinationRule

metadata:

name: circuit-breaker-for-the-entire-default-namespace

spec:

host: "fleetman-staff-service.default.svc.cluster.local"

trafficPolicy:

outlierDetection:

maxEjectionPercent: 100

consecutive5xxErrors: 2

interval: 10s

baseEjectionTime: 30s

| **설정 항목** | **의미** |
| --- | --- |
| consecutive5xxErrors: 2 | 동일 인스턴스에서 5xx 에러가 연속 2번 발생하면 이상으로 간주 |
| interval: 10s | 에러 감지 주기. 10초 간격으로 통계를 초기화하고 새로 측정 |
| baseEjectionTime: 30s | 이상 인스턴스를 30초 동안 트래픽에서 제외 |
| maxEjectionPercent: 100 | 전체 인스턴스 중 최대 100%까지 제외 가능 |

10초 안에 에러가 2번 연속 발생하면 30초 동안 트래픽을 제외함.

pod가 총 2개 있을 때, 2개가 다 맛탱이 가면 트래픽 다 차단됨.

maxEjectionPercent가 감이 잘 안올 수 있는데, 다음과 같이 적용됨.

가령 pod가 4개 있고, 4개 다 맛탱이가 갔는데,

maxEjectionPercent가 50이면 4개다 맛탱이 갔지만 2개의 pod에게는 트래픽을 보내줌.

**상호 TLS (Mutual TLS, mTLS)**

<https://istio.io/latest/docs/reference/config/security/peer_authentication/>

내부 네트워크에서 통신할 때에 암호화를 하기 위해 사용한다.

istio를 쓰지 않는다면 각 컨테이너에서 구현해야 하지만,

통신을 envoy proxy가 대신하기 때문에 직접 구현할 필요가 없다.

**TLS vs mTLS**

**TLS (Transport Layer Security):**

* 클라이언트가 서버의 인증서를 확인함 (예: HTTPS 웹사이트 접속).
* 서버는 클라이언트의 인증서 확인 안 함 (보통은 패스워드나 토큰으로 인증).
* 일방향 인증임.

**mTLS (mutual TLS):**

* TLS에 클라이언트 인증이 추가된 형태.
* 클라이언트도 인증서를 가지고 있어야 하며, 서버가 클라이언트 인증서를 확인함.
* 양방향 인증이라 보안이 더 강함.

**내부 통신도 암호화를 하는게 좋은가?**

Zero Trust 원칙으로 생각하자면 적용하는게 좋고,

AWS Well-Architected 프레임워크 보안 기둥 측면에서는 외부 트래픽은 TLS 적용을 권장하나,

내부 트래픽은 보안 정책에 따라 적용하라고 한다. (팀바팀으로 하라는 소리) [링크](https://docs.aws.amazon.com/wellarchitected/latest/security-pillar/sec_protect_data_transit_encrypt.html)

**적용 방법:**

Envoy는 상대 측 Envoy가 mTLS를 지원하면 자동으로 TLS 핸드셰이크 시도한다.  
그래서 서비스 간 통신은 거의 다 mTLS로 자동 적용되는 경우가 많다.

기본적으로 PERMISSIVE 모드로 동작한다.

따라서 mTLS가 적용된 트래픽과 아닌 트래픽 모두 허용 한다.

**확인 방법:**

mTLS 적용 유무는 kiali에서 쉽게 확인할 수 있다.

**엄격하게 적용하기 (STRICT):**

mTLS 적용이 안되는 트래픽은 차단 시킨다.

kind: "PeerAuthentication"

metadata:

name: "default"

namespace: "istio-system"

spec:

mtls:

mode: STRICT # PERMISSIVE, DISABLE

**mTLS 적용 안하기 (Disable):**

... # 바로 위 엄격하게 적용하기랑 똑같은데 mode만 DISABLE로 하면 된다.

spec:

mtls:

mode: DISABLE

**필요 시 더 알아보기:**

**istio 버전 업그레이드 :**

* **in-place upgrade :** <https://istio.io/latest/docs/setup/upgrade/in-place/>
* **canary :** <https://istio.io/latest/docs/setup/upgrade/canary/>
* 신 버전 istio 를 새로 프로비저닝하고, LB를 통해서 트래픽을 바꾸는 전략도 쓰인다.

**istio ambient 모드 :**

기존 Istio는 각 Pod에\*sidecar proxy (Envoy)를 삽입해 트래픽을 제어했는데,

이건 성능과 운영비용 부담이 크다.

Ambient 모드는 이를 없애고, 두 단계의 네트워크 계층 구성으로 메시를 구성한다.

Istio Ambient 모드는 sidecar 없이 서비스 메시 기능을 제공하는 새로운 방식이다.

CNI 기반의 L4 ztunnel과 L7 waypoint proxy로 트래픽을 처리한다.

퍼포먼스와 관리 복잡도 개선이 주요 목표이다.

1. ztunnel (L4 처리)

* 각 Node에 1개씩 존재하는 L4 proxy
* 모든 Pod의 트래픽을 인터셉트하고 목적지로 라우팅
* mTLS, 기본 인증/인가 처리 담당

2. waypoint proxy (L7 처리)

* 네임스페이스 단위로 존재
* HTTP 헤더 기반 라우팅, JWT 인증 등 고급 정책(L7) 적용
* 필요할 때만 트래픽이 여기로 흐름

**tunnel이 병목 지점이 되지 않나?**

맞다. 한 노드에 100개의 Pod가 있으면,

이 노드의 ztunnel 하나가 100개의 In/Out 트래픽을 모두 처리하게 된다.

**Istio ambient 모드 주요 포인트:**

* Envoy → HBone 기반 가벼운 proxy로 교체  
  ztunnel은 Envoy보다 리소스 사용량이 훨씬 적고 빠름
* L4만 처리 → L7 처리 분리  
  인증, 암호화, 라우팅만 하고, 무거운 L7은 waypoint로 분산 처리
* DaemonSet이지만 HPA 고려 가능  
  기본은 DaemonSet으로 Node당 1개 배포되지만,  
  필요 시 가상의 Node를 만들고 ztunnel만 여러 개 띄워서 부하 분산하는 구조도 가능  
  (공식적으로는 아직 정식 HPA 미지원, 커뮤니티에서 실험 중)
* Pod 수가 많은 Node는 수평 확장이 필요