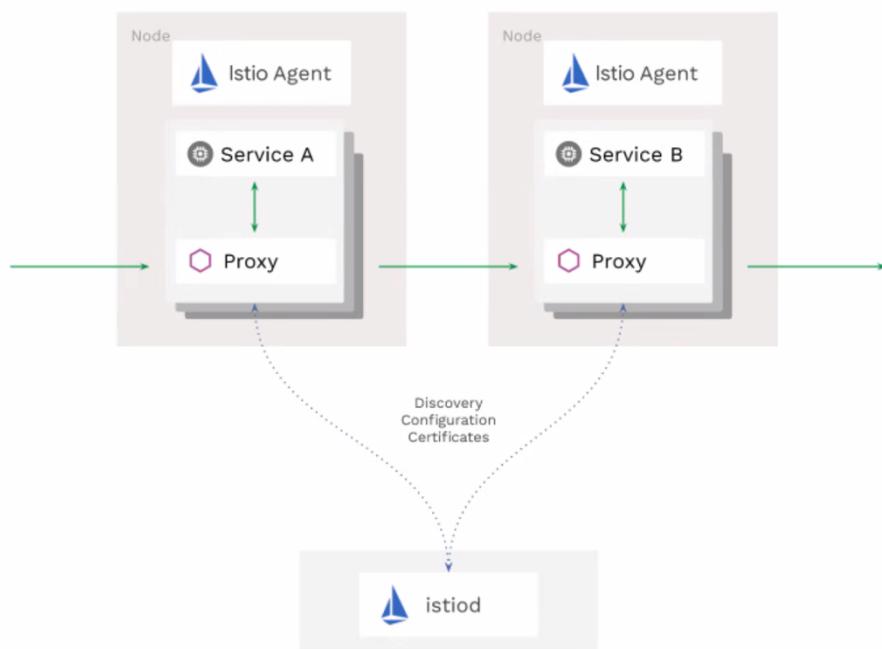


Istio

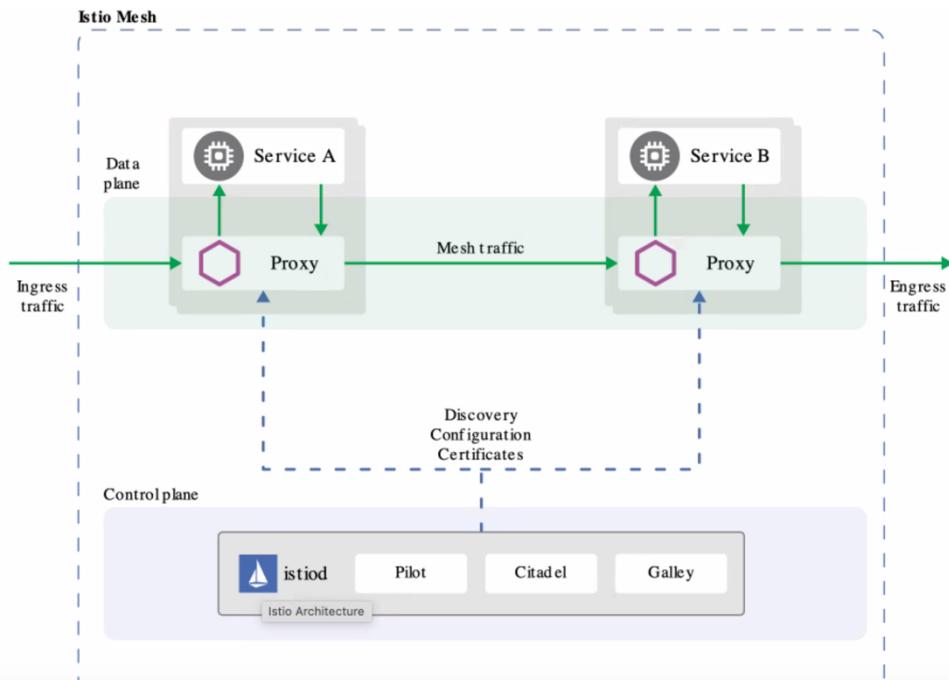
Service Mesh Introduction

Pattern który ma common network services jako cechy infrastruktury. Wszystkie rzeczy jak autentyfikacja, autoryzacja, enkrypcja są wymagane przez aplikację żeby dobrze działać, ale nie jest to jej oryginalne, docelowe działanie, więc zdecydowano to wyciągnąć na zewnątrz żeby nie stanowiło dodatkowej części aplikacji. Do tego używane jest Istio, aby implementować te funkcje jako infrastrukturę. Istio to najpopularniejszy Service Mesh i ma najwięcej dostępnych funkcjonalności. Pozwala na komunikację service-to-service pomiędzy microserwisami, umożliwia tworzenie canary deployment, ma load balancing, failure recovery, policy enforcement, monitoring ruchu paczek, telemetry itp.

Architektura



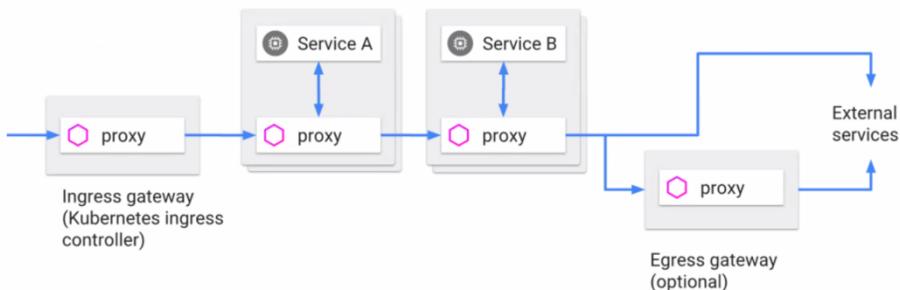
Mamy dwa główne komponenty: controlplane istiowe (istiod) - działa jako pod w istio-system namespace i dataplane - Envoy proxy i istio Agent, które są zaimplementowane jako kontener w każdym podzie.



Składowe istiod to Pilot (główny serwis, który rozmawia z Envoyami (rozмова odbywa się przy pomocy xDS API), mówimy jak chcemy mieć ustawioną sieć i Pilot ją przedstawia i tłumaczy do Envoy proxy), Citadel (zajmuje się Security, może on podpisać certyfikaty dla workloadów, zarządza autentyką i credentialami, to takie CA dla Istio), Galley (część, która rozumie komunikaty Kuberntesera, robi validację).

Data Plane

Envoy



Dataplane to przede wszystkim Envoy proxy, która może być oddzielnym obiektem, lub kontenerem w podzie. Ma ona wszystkie nasze ustawienia sieciowe jak enkrypcja, autentyka itp. Możemy ją ustawić jako ingress lub egress gateway dla całego clustra i wtedy cały ruch wejściowy lub wyjściowy będzie przez niego przechodzić.

Deployment Istio może być przeprowadzony przy pomocy Helm, Istio Operator lub command line istioctl. Poniżej będzie użyty istioctl korzystający z IstioOperator API. Najpierw pobieramy najnowszy release istio:

```
export ISTIO_VERSION=1.12.2
```

```
curl -L https://istio.io/downloadIstio | sh -
```

Po pobraniu mamy tutaj kilka folderów:

- samples (jest tutaj kilka aplikacji, których można użyć do testowania opcji Istio np. Prometheus, Kiali, Grafana i Jaeger. Należy pamiętać, że są to zewnętrzne aplikacje i przed użyciem lepiej sprawdzić ich dokumentację)
- manifests (instalowanie Istio przy pomocy istioctl to użycie Istio Helm chartu. W tym folderze właśnie znajdziemy fork Istio Helm chartu. Istio pozwala też na użycie profili, czyli możemy użyć wyedytowanej wersji takiego chartu i samemu zdefiniować istioOperator)
- tools (są tu skrypty mające Istio autocomplete i skrypty do generowania certyfikatów)

Teraz trzeba ustawić zmienną precyzującą lokalizację istio:

```
export ISTIO_INSTALL_DIR=$PWD
```

Teraz instalujemy istio przez istioctl z użyciem domyślnego profilu co jest rekomendowane. Mamy następujący istioOperator yaml file:

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  profile: default
  meshConfig:
    accessLogFile: "/dev/stdout"
```

Zaletą tej metody jest to, że możemy sobie sami określić, jak chcemy żeby nasz Service Mesh wyglądał, co pozwala na łatwiejsze śledzenie zmian. Wszystko czego nie ustawimy w tym pliku yaml będzie miało domyślne wartości. Więcej tutaj: [Istio / IstioOperator Options](#)

Instalacja Istio:

```
 ${ISTIO_INSTALL_DIR}/bin/istioctl install -f istio.yaml
```

Domyślnie Istio jest deployowane do istio-system namespace, sprawdzając powinniśmy tam widzieć 2 działające pody.

Envoy

On robi całą robotę dla Istio. Jest to open source, edge i service proxy mający wiele funkcjonalności sieciowych. Stworzony by pracować jako application sidecar proxy, czyli proxy na zewnątrz aplikacji. Może pracować z różnymi językami programowania. Dzięki niemu nie trzeba konfigurować oddziennie żadnej aplikacji sieciowo i pod względem autentykacji, tylko wystarczy go raz ustawić oddzielnie i aplikacja będzie mogła go używać.

Składowe Envoy:

Listeners - port, na którym Envoy będzie akceptował połączenia od klientów.

Listeners

A port in which Envoy will accept connections from clients.

```
$ istioctl -n istio-system proxy-config listener deploy/istio-ingressgateway
ADDRESS PORT MATCH DESTINATION
0.0.0.0 8080 ALL Route: http.8080
0.0.0.0 15021 ALL Inline Route: /healthz/ready*
0.0.0.0 15090 ALL Inline Route: /stats/prometheus*
```

istioctl -n istio-system proxy-config listener deploy/istio-ingressgateway

Ta komenda pokazuje wszystkie listenery dla deploymentu ingressgateway. Tu mamy 8080 15021 i 15090. Gdy ruch przyjdzie na tym porcie jest przerzucany do Route, która mówi, gdzie dalej go przekazać:

Routes

Rules for how to handle traffic that came in a listener.

```
$ istioctl -n istio-system proxy-config routes deploy/istio-ingressgateway

NAME      DOMAINS          MATCH          VIRTUAL SERVICE
http.8080 bookinfo.com    /productpage   bookinfo.default
           *                  /healthz/ready*
           *                  /stats/prometheus*
```



```
$ istioctl -n istio-system proxy-config routes deploy/istio-ingressgateway --name=http.8080 -o json | jq

[
  {
    "name": "http.8080",
    "virtualHosts": [
      {
        "name": "bookinfo.com:80",
        "domains": [
          "bookinfo.com",
          "bookinfo.com:80"
        ],
        "routes": [
          {
            "match": {
              "path": "/productpage",
              "caseSensitive": true
            },
            "route": {
              "cluster": "outbound|9080||productpage.default.svc.cluster.local",
              "weight": 100
            }
          }
        ]
      }
    ]
}
```

Istioctl -n istio-system proxy-config routes deploy/istio-ingressgateway

Jeśli wyświetlimy output w JSON to będzie pokazane więcej szczegółów. Jak robimy troubleshooting to dobrze jest to zobaczyć w JSON, ale trzeba to pofiltrować, bo output będzie duży. Następnie Route przekazuje ten ruch do clustra. W tym przykładzie mamy np. cluster na dole productpage.default.svc.cluster.local. Cluster jest trochę odpowiednikiem service z Kubernetesa, który ma za sobą jakąś pulę podów oznaczonych Selectorem. Tak samo cluster może mieć za sobą wiele endpointów. Jest to service, do którego Envoy wysyła ruch. Downstream - obiekt, który wysyła request, a upstream to obiekt docelowy.

Clusters

Upstream services to which Envoy can send traffic to.

```
$ istioctl -n istio-system proxy-config cluster deploy/istio-ingressgateway

SERVICE FQDN          PORT  SUBSET  DIRECTION  TYPE  DESTINATION RULE
BlackHoleCluster       -     -        -          STATIC
agent                 -     -        -          STATIC
debug-svc.default.svc.cluster.local  8080  -        outbound  EDS
istio-egressgateway.istio-system.svc.cluster.local  80  -        outbound  EDS
istio-ingressgateway.istio-system.svc.cluster.local  443  -        outbound  EDS
istio-ingressgateway.istio-system.svc.cluster.local  80  -        outbound  EDS
istio-ingressgateway.istio-system.svc.cluster.local  443  -        outbound  EDS
istio-ingressgateway.istio-system.svc.cluster.local  15021 -        outbound  EDS
istio-ingressgateway.istio-system.svc.cluster.local  15443 -        outbound  EDS
istio-ingressgateway.istio-system.svc.cluster.local  31400 -        outbound  EDS
istiod.istio-system.svc.cluster.local  443  -        outbound  EDS
istiod.istio-system.svc.cluster.local  15010 -        outbound  EDS
istiod.istio-system.svc.cluster.local  15012 -        outbound  EDS
istiod.istio-system.svc.cluster.local  15014 -        outbound  EDS
kube-dns.kube-system.svc.cluster.local  53  -        outbound  EDS
kube-dns.kube-system.svc.cluster.local  9153 -        outbound  EDS
kubernetes.default.svc.cluster.local  443  -        outbound  EDS
productpage.default.svc.cluster.local  9080  -        outbound  EDS
prometheus_stats          -     -        -          STATIC
reviews.default.svc.cluster.local  9080  -        outbound  EDS  reviews.default
reviews.default.svc.cluster.local  9080  v1     outbound  EDS  reviews.default
sds-grpc                -     -        -          STATIC
xds-grpc                -     -        -          STATIC
zipkin                  -     -        -          STRICT_DNS
```

Istioctl -n istio-system proxy-config cluster deploy/istio-ingressgateway

Na końcu mamy endpoint, jest to IP przypisane np. podowi i cel ruchu.

Endpoints

Hosts that belong to a cluster.

ENDPOINT	STATUS	OUTLIER CHECK	CLUSTER
10.244.0.106:15010	HEALTHY	OK	outbound 15010 istiod.istio-system.svc.cluster.local
10.244.0.106:15012	HEALTHY	OK	outbound 15012 istiod.istio-system.svc.cluster.local
10.244.0.106:15014	HEALTHY	OK	outbound 15014 istiod.istio-system.svc.cluster.local
10.244.0.106:15017	HEALTHY	OK	outbound 443 istiod.istio-system.svc.cluster.local
10.244.0.124:9080	HEALTHY	OK	outbound 9080 reviews.default.svc.cluster.local
10.244.0.124:9080	HEALTHY	OK	outbound 9080 reviews.default.svc.cluster.local
10.244.0.188:8080	HEALTHY	OK	outbound 80 istio-ingressgateway.istio-system.svc.cluster.local
10.244.0.188:8443	HEALTHY	OK	outbound 443 istio-ingressgateway.istio-system.svc.cluster.local
10.244.0.188:15021	HEALTHY	OK	outbound 15021 istio-ingressgateway.istio-system.svc.cluster.local
10.244.0.188:15443	HEALTHY	OK	outbound 15443 istio-ingressgateway.istio-system.svc.cluster.local
10.244.0.188:31400	HEALTHY	OK	outbound 31400 istio-ingressgateway.istio-system.svc.cluster.local
10.244.0.62:53	HEALTHY	OK	outbound 53 kube-dns.kube-system.svc.cluster.local
10.244.0.62:9153	HEALTHY	OK	outbound 9153 kube-dns.kube-system.svc.cluster.local
10.244.0.65:8080	HEALTHY	OK	outbound 80 istio-egressgateway.istio-system.svc.cluster.local
10.244.0.65:8443	HEALTHY	OK	outbound 443 istio-egressgateway.istio-system.svc.cluster.local
10.244.0.68:9080	HEALTHY	OK	outbound 9080 productpage.default.svc.cluster.local
10.244.0.84:9080	HEALTHY	OK	outbound 9080 productpage.default.svc.cluster.local
127.0.0.1:15000	HEALTHY	OK	prometheus_stats
127.0.0.1:15020	HEALTHY	OK	agent
192.168.64.53:8443	HEALTHY	OK	sds-grpc
unix:///etc/istio/proxy/SDS	HEALTHY	OK	xds-grpc
unix:///etc/istio/proxy/XDS	HEALTHY	OK	

xDS są to API, pomagające skonfigurować Envoy. Mamy następujące xDS:

- Listener Discovery Service (LDS) - pozwala Envoyowi pytać, jakie listenery powinno być dopuszczone w konkretnej proxy,
- Route Discovery Service (RDS) - część konfiguracji listenera, która określa jaką Route powinna być użyta,
- Cluster Discovery Service (CDS) - API, które pozwala Envoyowi sprawdzić jaki cluster i jaką konfigurację dla clustra powinna mieć proxy,
- Endpoint Discovery Service (EDS) - część konfiguracji clustrów określająca, jaki endpoint powinien być użyty dla danego clustra,
- Secret Discovery Service (SDS) - API używane do dystrybucji certyfikatów,

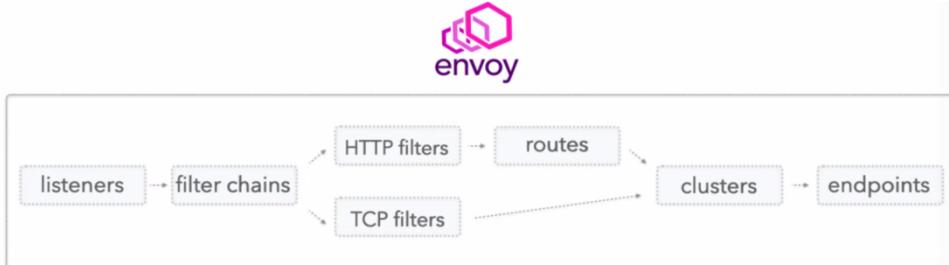
Konfiguracja Envoy jest trudna, więc Istio nam pomaga. Kubernetes podaje Istio to co chce osiągnąć, a Istio tłumaczy te requesty do Envoy, który następnie wdraża je w życie. Dlatego mamy istioctl a nie envoyctl.

Tabela podsumowująca obiekty Kuberentesa i Istio wpływające na poszczególne obiekty w Envoyu:
From Custom Resource to Envoy config

Resource Type	Envoy Configuration	Notes
Kubernetes Services	Listeners Routes Clusters	New listeners if port/protocol combo is unique Add virtual hosts for existing routes One cluster per Service/Port/Subset
Kubernetes Endpoints	Endpoints	
Istio Gateway	Listeners	Apply to Ingress/Egress Gateways
Istio VirtualService	Listeners Routes	Client side proxies TLS/TCP affect listeners HTTP match blocks affect routes
Istio DestinationRule	Clusters Endpoints	Client side proxies Connection/HTTP/TLS settings
Istio ServiceEntry	Clusters Endpoints	Client side proxies
Istio PeerAuthentication	Listeners Clusters	Server side proxies
Istio RequestAuthentication	Listeners	Server side proxies
Istio Authorization Policies	Listeners	Server side proxies
Istio EnvoyFilter	All	Break glass API to directly manipulate Envoy
Istio Sidecar	All	Client or server side proxies Sidecar scope sets config visibility

Istotnym elementem jest tutaj EnvoyFilter, który pozwala na wysłanie konfiguracji Envoyowej do proxy. Jest to trudne, bo tu Istio nie zrobi tego za nas i trzeba dobrze wiedzieć, jak ten fragment konfiguracji napisać. Trzeba też powiedzieć Istio dokładnie, w którym miejscu to dodać.

Droga pojedynczego pakietu w Envoy:



Zanim pakiet dostanie się do poda, przechodzi przez Envoy proxy. Najpierw sprawdzany jest listener, czyli to czy dany port komunikacji sieciowej jest zezwolony. Następnie ruch przechodzi do filter chains (w zależności od konfiguracji mogą to być różne filtry). W większości przypadków będzie to http filter (dla połączeń sieciowych), ma on dodatkowe filter chainy w sobie, na podstawie których wysyła ruch do route. Filtrowanie może się np. odbywać na podstawie sprawdzania headerów poszczególnych połączeń. Filter chain może odrzucić request, jak coś nie będzie spełnione. Ostatni filtr w chainie to tzw. http Router, który przesyła pakiet do Route. Następnie idzie on do Clustra i do Endpointa, czyli np. poda.

Przykłady ustawień Listenera, FilterChaina (tu jest filtr, który konwertuje bajty na http protocol) i HTTP filtra:

```
"dynamic_listeners": [
{
  "name": "0.0.0.0_8080",
  "active_state": {
    "version_info": "2022-01-07T22:34:54Z/14",
    "listener": {
      "@type": "type.googleapis.com/envoy.config.listener.v3.Listener",
      "name": "0.0.0.0_8080",
      "address": {
        "socket_address": {
          "address": "0.0.0.0",
          "port_value": 8080
        }
      }
    }
  }
}

"filter_chains": [
{
  "filters": [
    {
      "name": "envoy.filters.network.http_connection_manager",
      "typed_config": {
        "@type": "type.googleapis.com/envoy.extensions.filters.network.http
        "stat_prefix": "outbound_0.0.0.0_8080",
        "rds": {
          "config_source": {
            "ads": {}
          },
          "initial_fetch_timeout": "0s",
          "resource_api_version": "V3"
        }
      }
    }
  ]
}
```

```

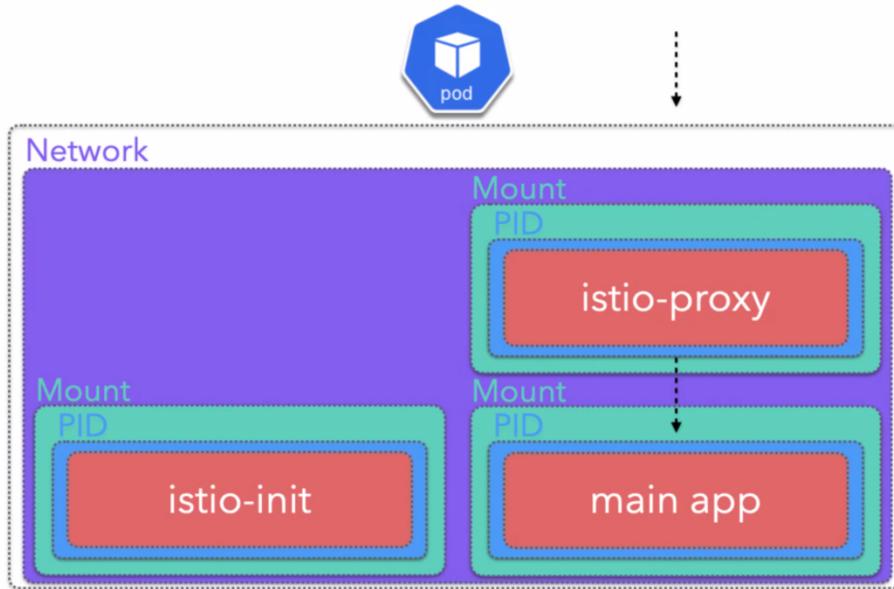
        "route_config_name": "http.8080"
    },
    "http_filters": [
        {
            "name": "envoy.filters.http.fault",
            "typed_config": {
                "@type": "type.googleapis.com/envoy.extensions.filters.http.fault.v
            }
        },
        {
            "name": "envoy.filters.http.router",
            "typed_config": {
                "@type": "type.googleapis.com/envoy.extensions.filters.http.router.
            }
        }
    ]
]

```

Istio Sidecar

Jak implementowane jest Istio w Kubernetesie?

A Pod in the mesh



W Kubernetesie możemy mieć kilka kontenerów w podzie i będą one miały ten sam network namespace, czyli ip tables, network interfaces itp. Dzięki temu Istio może stworzyć Service Mesh. W przypadku Istio mamy dodatkowy kontener `istio-proxy` - w nim jest Envoy i Pilot. Może też być dodatkowo `istio-init` container, który wstaje zawsze razem z podem, wykonuje swoją akcję i się wyłącza, dopiero gdy zostanie usunięty, inne kontenery zostaną uruchomione. Jego zadaniem jest przekierowanie całego ruchu najpierw do `istio-proxy` containera, jest to realizowane poprzez modyfikację iptables. Mogłoby to robić także `istio-proxy`, ale wtedy miałoby za dużo przywilejów sieciowych (potrzebny jest `NET_ADMIN` i `NET_RAW`), więc chciano stworzyć oddzielny kontener, który będzie usunięty po wykonaniu zadania. Mamy też `istio` CNI plugin, który zastępuje `istio-init` i jest w stanie wykonać te zadania. Może on być zdeployowany jako daemonset na każdym node i kubelet będzie go prosił o wykonanie zmian w iptables za każdym razem, gdy pod zostanie stworzony.

Port 15001 - outbound port, jeśli nasza aplikacja wysyła jakikolwiek ruch outbound, to będzie on przekierowany na ten port.

Port 15006 - inbound port, każdy ruch przychodzący będzie przekierowany na ten port, następnie pójdzie do Envoya i w nim przez Listeners, Filter chain, cluster i dopiero do endpointa.

Aby uruchomić cały proces tworzenia Istio wewnątrz poda, korzystamy z mutating webhook.

Sidecar injection per Namespace

```
kubectl label namespace default istio-injection=enabled --overwrite
```

```
apiVersion: admissionregistration.k8s.io/v1
kind: MutatingWebhookConfiguration
metadata:
  labels:
    app: sidecar-injector
    name: istio-sidecar-injector
webhooks:
- admissionReviewVersions:
  - v1beta1
  - v1
  clientConfig:
    caBundle: <REDACTED>
    service:
      name: istiod
      namespace: istio-system
      path: /inject
      port: 443
    failurePolicy: Fail
    matchPolicy: Equivalent
    name: namespace.sidecar-injector.istio.io
  namespaceSelector:
    matchExpressions:
    - key: istio-injection
      operator: In
      values:
      - enabled
  objectSelector:
    matchExpressions:
    - key: sidecar.istio.io/inject
      operator: NotIn
      values:
      - "false"
  rules:
  - apiGroups:
    - ""
      apiVersions:
      - v1
      operations:
      - CREATE
      resources:
      - pods
      scope: '*'
      sideEffects: None
      timeoutSeconds: 10
  reinvocationPolicy: Never
```

Dzięki temu za każdym razem, gdy pod jest tworzony i spełnia określony warunek np. ma określoną labelkę (tu `istio-injection=enabled`), przechodzi przez ten webhook i on tworzy kontenery istiowe. Dzięki temu nie trzeba samemu ustawiać dodatkowych kontenerów w pliku yaml. W ustawieniach mutating webhooka mamy `namespaceSelector` i tu w przykładzie jest ustawiony warunek, że gdzie jest `istio-injection` label, tam trzeba dodać `istio`. W `webhooks->clientConfig->service` mamy precyzowane, gdzie każdy request ma być wysyłany i tutaj jest to `istiod` w `istio-system` namespace. Istiod jest odpowiedzialne za dodanie dodatkowych kontenerów. Jest tu też `objectSelector`, który też pozwala precyzować w jakich podach będzie dodane Istio, tutaj tak gdzie nie ma wartości `false` dla odpowiedniej labelki.

Innym sposobem na dodanie Istio jest komenda `istioctl` (doda ona automatycznie Istio do poda tworzonego w pliku yaml):

```
istioctl kube-inject -f manifest.yaml | kubectl apply -f -
```

Lub poprzez `kubectl` dodać odpowiedni label do namespace, z flagą `overwrite`, skutkujący tym, że wszystkie pody będą miały tam Istio.

```
kubectl label namespace default istio-injection=enabled --overwrite
```

Output istio-proxy containera:

istio-proxy – Istio and Envoy

The istio-proxy container has Istio and Envoy configured. The communication between Istio and Envoy is done via UNIX socket

```
istio-proxy@centos-pod:~$ ls -l /etc/istio/proxy/
total 20
srw-rw-rw- 1 istio-proxy istio-proxy    0 Oct 22 07:34 SDS
srw-rw-rw- 1 istio-proxy istio-proxy    0 Oct 22 07:34 XDS
-rw-r--r-- 1 istio-proxy istio-proxy 17675 Oct 22 07:34 envoy-rev0.json
istio-proxy@centos-pod:~$ netstat -plnt
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address          Foreign Address        State      PID/Program name
tcp     0      0 0.0.0.0:15006            0.0.0.0:*              LISTEN     19/envoy
tcp     0      0 0.0.0.0:15021            0.0.0.0:*              LISTEN     19/envoy
tcp     0      0 0.0.0.0:15090            0.0.0.0:*              LISTEN     19/envoy
tcp     0      0 127.0.0.1:15000           0.0.0.0:*              LISTEN     19/envoy
tcp     0      0 0.0.0.0:15001            0.0.0.0:*              LISTEN     19/envoy
tcp     0      0 127.0.0.1:15004           0.0.0.0:*              LISTEN     1/pilot-agent
tcp6    0      0 ::1:15020               ::*:*                  LISTEN     1/pilot-agent
```

Mamy tu 2 procesy pilot-agent i envoy. Rozmawiają ze sobą przez UNIX sockety, tutaj XDS (wysyłanie konfiguracji do Envoya, czyli update Listeners, Clusters, Routes itp) i SDS (konfiguracja certyfikatów i przesyłanie ich do Envoya - jeśli mamy mTLS to od razu będą potrzebne te certyfikaty).

Większość routingu w Kubernetesie jest na poziomie TCP, a nie ma dużo na poziomie http (żeby to uzyskać potrzebna Ingress Controllera, albo Service Mesha - czyli Istio). Istio dodaje możliwości routingu na poziomie http. Można tu ustawać circuit breakers, timeouts, retries, canary rollouts, staged rollouts with percentage based traffic itp. Główne resourcy do tworzenia routingu to VirtualService i DestinationRule.

VirtualService

VirtualService określa jak requesty będą przekazywane do docelowego service. W sekcji spec->http ustawiamy route (może być kilka). Służą one do tego, aby opisywać, jak i gdzie będzie przekazywany ruch, dodatkowo można w nich ustawić rozdzielenie ruchu oraz filtrowanie po http (np. po headerach), co nie jest trudne do osiągnięcia w Kubernetesie. Aby ustawić filtrowanie należy dodać pole *match* w danej route. Routy są sprawdzane od góry do dołu i pierwszy z nich, w którym zostanie spełniony warunek będzie użyty, więc na końcu najlepiej ustawić jakiś ogólny/domyślny wpis. W polu host określamy docelową nazwę hosta, ale trzeba pamiętać, żeby podawać FQDN, bo jeśli podamy samą nazwę to automatycznie zostanie nam doklejona domena kubernetesowa np. jeśli tak jak tutaj, nie podaliśmy namespace to Kubernetes myśli, że będzie to dla domyślnego namespace, czyli reviews.default.svc.cluster.local.

```

apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
  - reviews
  http:
  - route:
    - destination:
        host: reviews
        subset: v2
    match:
    - headers:
        end-user:
          exact: jason
  - route:
    - destination:
        host: reviews
        subset: v3

```

W przykładzie powyżej mamy sprecyzowane dwie drogi. Pomimo tego, że mają taki sam hostname, to jednak sposób dotarcia do niego będzie inny i to definiuje *subset*, który bardziej szczegółowo jest precyzowany w DestinationRule. Pierwsza będzie użyta tylko i wyłącznie jeśli w requeście będzie header *end-user: jason*. Jeśli go nie będzie, to ruch zostanie przekierowany do drugiej drogi.

DestinationRule

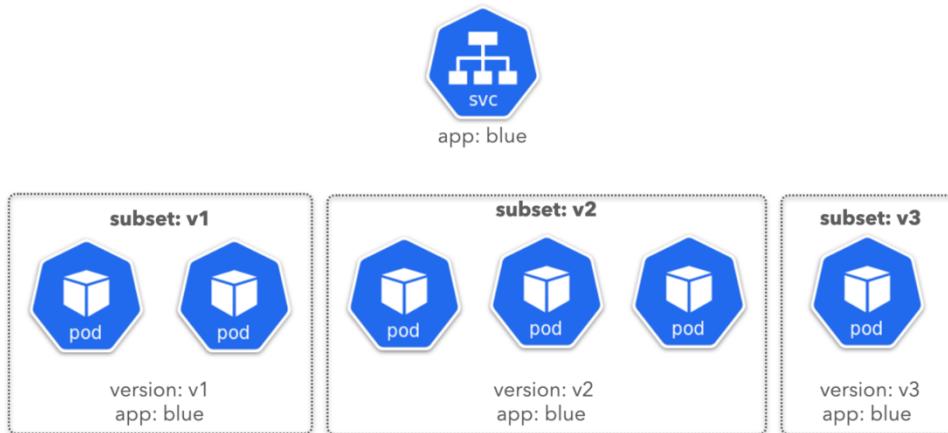
W DestinationRule podajemy policy, mówiące jak ruch będzie szedł do docelowego endpointa (clustra). Działa to na podobnej zasadzie, jak Kubernetesowe service, który jest łączony z podami poprzez labele. W przypadku DestinationRule mamy *subsets*, które dają większe możliwości rozdzielania ruchu. Ustawienia z DestinationRule są wdrażane dopiero po tym, jak nastąpi routing zrobiony przez VirtualService.

DestinationRules Example

```

apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: my-destination-rule
spec:
  host: my-svc
  trafficPolicy:
    loadBalancer:
      simple: RANDOM
  subsets:
    name: v1
    labels:
      version: v1
    name: v2
    labels:
      version: v2
    trafficPolicy:
      loadBalancer:
        simple: ROUND_ROBIN
    name: v3
    labels:
      version: v3

```



W przykładzie powyżej mamy 3 subsety i w każdym z nich dodajemy labela, na podstawie których rozdzielany i kierowany będzie ruch (tu mamy podzielenie na różne wersje aplikacji). Mamy tutaj też możliwość dodania TrafficPolicy, która daje dodatkowe możliwości modyfikowania ruchu - tutaj ustawiony jest sposób load balancingu (popularne opcje load balancingu to RANDOM, WEIGHTED - procentowy podział, LEAST_REQUEST - requesty wysypane są do endpointa, który ma ich najmniej). TrafficPolicy może być dodana bezpośrednio w sekcji spec (będzie nadana do wszystkich subsetów), lub w danym subsecie (wtedy ta dokładniejsza policy nadpisze ogólną z sekcji spec).

Należy pamiętać, że Istio używa Kubernetes service jako discovery mechanism, więc on też jest potrzebny i trzeba go stworzyć. Dopiero dzięki niemu Istio widzi pody i np. może wziąć porty do komunikacji.

ServiceEntry

Aby móc kierować ruchem wewnętrz mesha, Istio musi wiedzieć, gdzie są wszystkie endpointy i do jakich service należą. ServiceEntry jest używane do dodawania wpisów bezpośrednio do wewnętrznego rejestru Istio. Po dodaniu takiego wpisu Envoy proxy może przesyłać ruch do określonego hosta, tak samo, jakby był on wewnętrz mesha. Można tutaj dodawać zarówno zewnętrzne, jak i wewnętrzne hosty (np. mamy jakąś aplikację na EC2, która nie jest w Kubernetesie i chcemy się z nią łączyć). Poza zarządzaniem ruchem do zewnętrznych hostów ServiceEntry pozwala też na:

- przekierowywanie ruchu do zewnętrznych endpointów, takich jak API w webie lub service w legacy infrastructure.
- tworzenie policy pozwalających na zatrzymywanie, czy timeout ruchu do zewnętrznych endpointów.
- logiczne dodawanie service z innych clustrów do mesha, aby skonfigurować multicluster Istio mesh.
- dodawanie zewnętrznych workloadów z VM, który będzie traktowany jako wewnętrzny członek clustra.

The following example `MESH_EXTERNAL` service entry adds the `ext-svc` external dependency to Istio's service registry:

```

apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: svc-entry
spec:
  hosts:
  - ext-svc.example.com # FQDN of external resource or wildcard prefix
  ports:
  - number: 443
    name: https
    protocol: HTTPS
  location: MESH_EXTERNAL
  resolution: DNS

```

W przykładzie powyżej dodajemy ext-svc - zewnętrzny endpoint do rejestru Istio. Wartość parametry *location* ustawiona na MESH_EXTERNAL mówi Istio, że dodajemy jakiś zewnętrzne źródło. W *resolution* ustawiamy jak ma być odczytywany adres, tutaj wybrany jest DNS (czyli za każdym razem, żeby odszyforwać endpoint, Istio będzie używać DNS), ale można też ustawić IP.

W Envoy mamy 2 clustery: passthrough i blackhole. Domyślnie, gdy Envoy chce przesłać request do hosta, to sprawdza w rejestrze Istio, czy jest tam ten host. Jeśli jest, to wysyła request do passthrough cluster i on dalej go przesyła do hosta. Ale można też ustawić nasz mesh żeby był "registry only" i wtedy każdy nieznany endpoint będzie wysyłany do blackhole cluster i trzeba dodać ServiceEntry żeby request poszedł na zewnątrz.

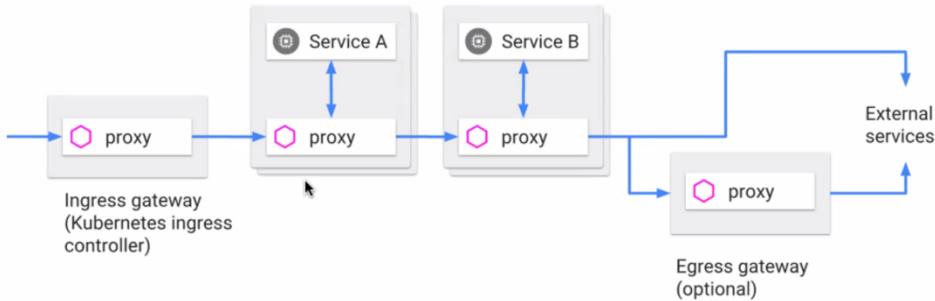
Multi cluster services

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: httpbin-bar
spec:
  hosts:
    # must be of form name.namespace.global
    - httpbin.bar.global
    # Treat remote cluster services as part of the service mesh
    # as all clusters in the service mesh share the same root of trust.
  location: MESH_INTERNAL
  ports:
    - name: http1
      number: 8000
      protocol: http
  resolution: DNS
  addresses:
    # the IP address to which httpbin.bar.global will resolve to
    # must be unique for each remote service, within a given cluster.
    # This address need not be routable. Traffic for this IP will be
    # captured by the sidecar and routed appropriately.
    - 240.0.0.2
  endpoints:
    # This is the routable address of the ingress gateway in cluster2
    # that sits in front of sleep.foo service. Traffic from the sidecar
    # will be routed to this address.
    - address: ${CLUSTER2_GW_ADDR}
      ports:
        http1: 15443
```

W przykładzie powyżej mamy wiele clustrów Istio i chcemy żeby aplikacja z clustra A rozmawiała z aplikacją w clustrze B. Tutaj wartość parametru *location* jest ustawiona na MESH_INTERNAL, a host musi być postaci *nazwa.namespace.global*.

Gateway

Istio może kontrolować ruch przychodzący i wychodzący z mesha definiując ingress i egress Gateway. Konfiguracje Gatewya są przypisane do Envoy proxy, które są osobnymi obiektami (nie są wewnątrz poda jako kontener). Ingress Gateway instruuje wejściowy load balancer, który otrzymuje przychodzące http/tcp requesty. Egress gateway natomiast, kontroluje ruch wyjściowy z mesha, określając, które service mogą mieć dostęp do zewnętrznych sieci. Pozwala to na zwiększenie bezpieczeństwa w meshu.



httpbin-gateway.yaml

```

apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: httpbin-gateway
spec:
  selector:
    istio: ingressgateway # use Istio default gateway implementation
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "httpbin.example.com"
  
```

Powyżej mamy plik yaml tworzący konfigurację istniejącego już Gatewya. Należy pamiętać, że taki plik yaml, nie tworzy nowej instancji Gatewya, ale dodaje do niego konfigurację. To do którego rzeczywistego gatewya dodajemy tą konfigurację określa label selector, tutaj *istio: ingressgateway*. W tym przypadku konfiguracja zezwala połączenia przez gateway do danego URLa.

Ingress and Egress Gateway installation

```

istioctl manifest install -f <(cat <<EOF
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  profile: default
  components:
    ingressGateways:
    - enabled: true # default is true for default profile
    egressGateways:
    - enabled: true
EOF
)
  
```

Dopiero tutaj w powyższym pliku yaml mamy instalację faktycznego Gatewya jako obiektu, ustawiamy tu, czy ma być ingress i egress. Po stworzeniu obiektu Gatewya trzeba sprecyzować w VirtualService (jeśli chcemy się łączyć z zewnętrznymi hostami), którego Gatewya będziemy używać. Należy jednak pamiętać, że precyzuje tu nie rzeczywisty gateway (ten stworzony przez IstioOperator yaml file), ale konfigurację gatewya (stworzoną przez Gateway yaml file):

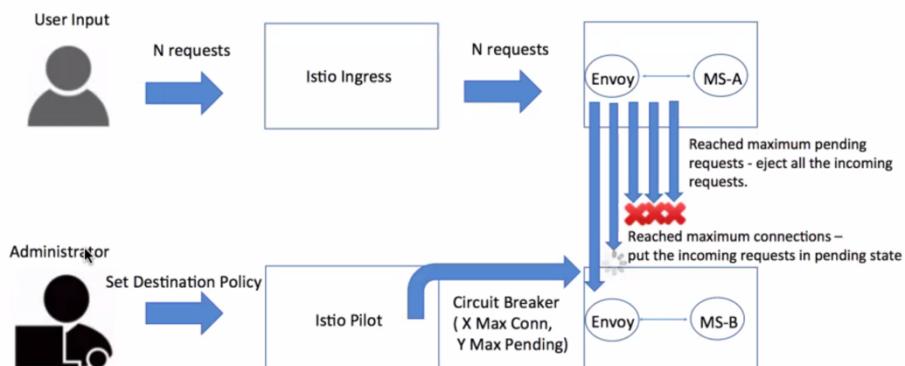
The Gateway must be bound to a VirtualService and specify routing as show below:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: httpbin
spec:
  hosts:
  - "httpbin.example.com"
  gateways:
  - httpbin-gateway
  http:
  - match:
    - uri:
        prefix: /status
    - uri:
        prefix: /delay
    route:
    - destination:
        port:
          number: 8000
        host: httpbin
```

Circuit Breaking

Jest to wbudowany w Istio mechanizm, który zapewnia, większą elastyczność i wydajność poprzez szybszy failure aplikacji, zamiast wymagania łączenia się do przeładowanego endpointa. Jeśli dołożymy kolejny request do czegoś przeładowanego to będzie jeszcze gorzej, więc lepiej do razu stworzyć szybki failure.

Circuit Breaking



<https://developer.ibm.com/code/wp-content/uploads/sites/118/2017/08/Screen-Shot-2017-08-22-at-12.50.46-AM-768x342.png>

Realizujemy to poprzez dodanie w DestinationRule opcji trafficPolicy:

Circuit Breaking

The following example limits the number of concurrent connections for the reviews service workloads of the v1 subset to 100:

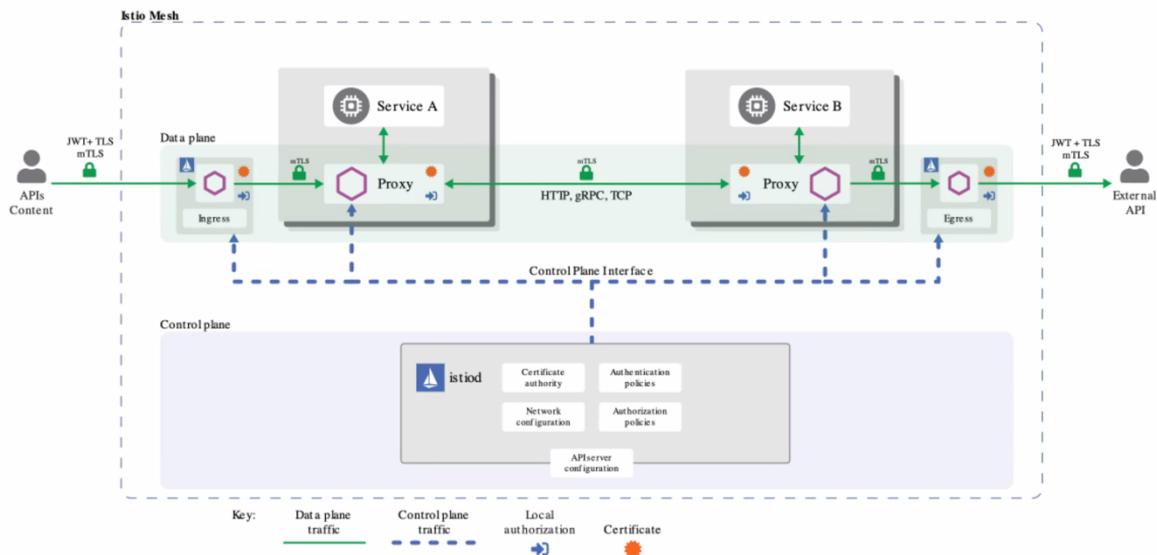
```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: reviews
spec:
  host: reviews
  subsets:
  - name: v1
    labels:
      version: v1
    trafficPolicy:
      connectionPool:
        tcp:
          maxConnections: 100
          connectTimeout: 10s
      outlierDetection:
        consecutive5xxErrors: 7
        interval: 5s
        baseEjectionTime: 3m
        maxEjectionPercent: 30
```

Powyższy przykład wprowadza limit na 100 jednoczesnych połączeń dla service *reviews* w subsecie v1 (w sekcji *connectionPool* możemy też ustawić ograniczenia na http, nie tylko tcp). Jest to też dodatkowo pasywne sprawdzanie, czy dany endpoint jest cały czas zdrowy. Są różne warunki uznawania endpointu za unhealthy: np. *consecutive5xxErrors* (ile może być pod rząd błędów 5xx przez odłączeniem hosta), *interval* (co ile jest sprawdzany health poda), *baseEjectionTime* (minimalny czas odłączenia hosta), *maxEjectionPercent* (maksymalny procent hostów z load balancing pool, które mogą być odłączone). Jest to trochę podobne do zasady działania Readiness Probe w Kubernetesie.

Security

Istio pozwala zwiększyć możliwości zarządzania ruchem sieciowym w clustrze, a dodatkowo zwiększa jego bezpieczeństwo. Między innymi Istio pozwala na obronę przeciwko man-in-the-middle attacks, gdzie potrzebujemy enkrypcji ruchu sieciowego pomiędzy servicami, używanie mTLS i fine-grained access policies, aby elastycznie kontrolować dostęp do service, korzystanie z audytów i logów, aby sprawdzić, kto co i kiedy zrobił. Głównymi założeniami Istio security są: brak potrzeby zmian w kodzie aplikacji lub w infrastrukturze na potrzeby bezpieczeństwa, integracja z obecnymi systemami bezpieczeństwa, aby zapewnić ochronę na wielu poziomach, blokowanie wszelkiego ruchu od nieznanych, groźnych źródeł. Dzięki Istio cały ruch do poda przechodzi przez Envoy proxy, które zapewnia zaszyfrowany ruch, oraz użycie mTLS.

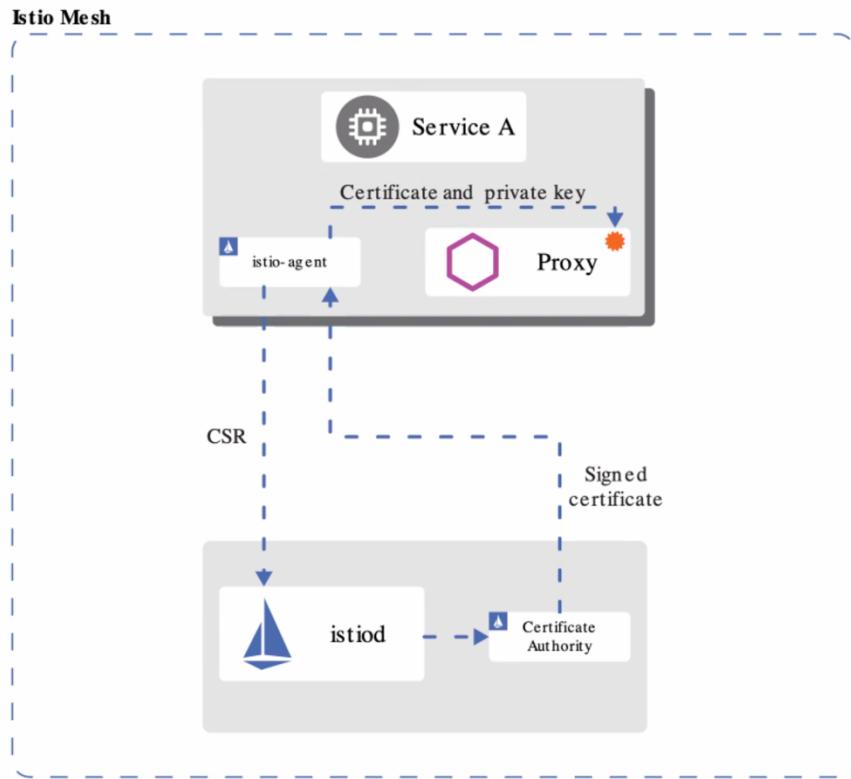
High-level architecture



Tożsamość, to podstawa w każdym zarządzaniu bezpieczeństwem systemu. Istio używa first-class service identity do określania tożsamości service. To daje duże możliwości w reprezentowaniu pojedynczego usera, service, czy grupy service.

Jeśli rozbijemy aplikacje na wiele małych to potrzebujemy identity żeby mieć pewność, że dany service chcący się połączyć jest naprawdę tym, za kogo się podaje. W Kuberentesie mamy Service Accounty (jeśli stworzymy poda bez SA to weźmie domyślne z namespace). Dobrą więc praktyką jest stworzyć SA dla każdej aplikacji i Istio będzie mogło ich używać. Jak Istio będzie tworzyć certyfikat, to będzie już tam miało wbudowane, że jest on dla danego Service Account, a co za tym idzie dla danego poda, czy service.

Istio PKI (Public Key Infrastructure) zapewnia certyfikaty X.509 dla każdego workloadu (w nich jest właśnie wpisana tożsamość danego service), aby było to zautomatyzowane, Istio uruchamia agenta obok każdego Envoy proxy, po to, aby tworzył on certyfikaty i klucze.



Zawsze, gdy pod jest tworzony w service meshu, Envoy robi request do SDS żeby otrzymać certyfikat dla nowo stworzonego ServiceAccount. Następnie zajmuje się nim istio-agent, tworzy Certificate Signing Request i wysyła on go do istiod wraz z nazwą Service Account, następnie jest on przesłany do CA wraz z ServiceAccount tokenem. Tu następuje podpisanie certyfikatu i odesłanie z powrotem do istio-agenta, a na koniec do Envoy proxy.

Mutual TLS

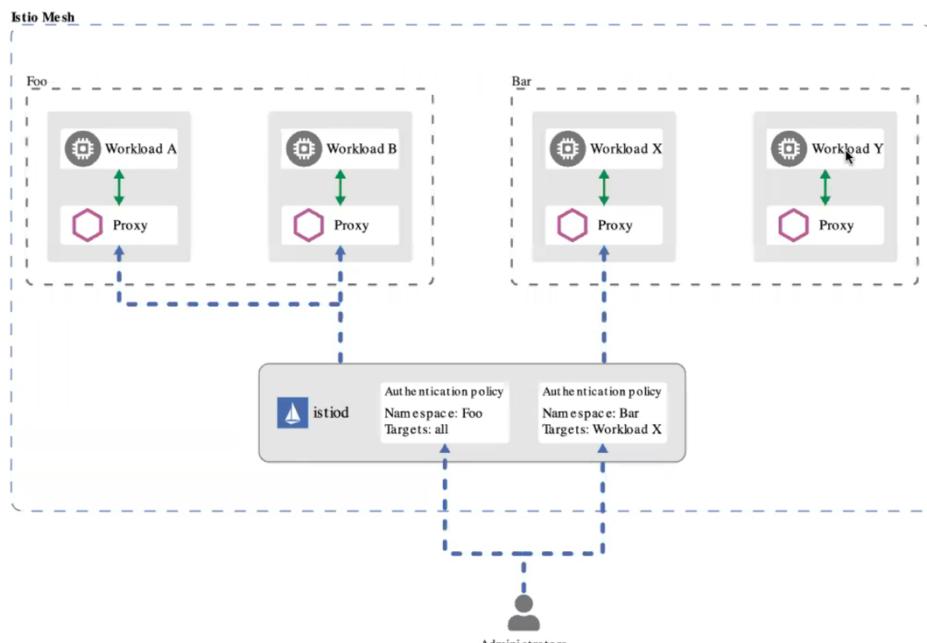
Cały ruch wewnętrz service mesha jest domyślnie zaszyfrowany przez mTLS. Cały ten ruch przechodzi przez instancje Envoy proxy w źródle i celu połączenia. Aby klient mógł wykonać call z mTLS authentication:

1. Istio przesyła wychodzący ruch od klienta do jego lokalnego Envoy proxy,
2. Envoy proxy po stronie klienta rozpoczyna mTLS handshake z Envoy proxy, po stronie docelowego serwera (sprawdzane są tu certyfikaty obu stron),
3. Następuje ustanowienie połączenia mTLS pomiędzy dwoma Envoy proxy i dopiero teraz wysyłany jest ruch z jednego proxy do drugiego,
4. Po autoryzacji, docelowy Envoy proxy przesyła ruch do docelowego service przez połączenie TCP.

Należy pamiętać, że z mTLS należy używać protokołu http, a nie https.

Authentication

Authentication architecture



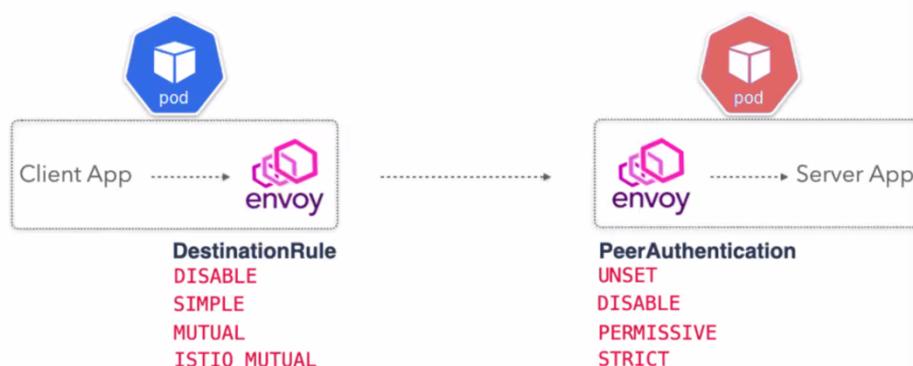
Istio zapewnia 2 rodzaje autentykacji - peer i request. Dodatkowo, są 3 poziomy na których możemy tę autentykację zastosować: workload-specific, namespace-wide i mesh-wide. Będą one sprawdzane zawsze w tej kolejności i wejdzie w życie ta najwęższa.

Peer Authentication, jest to prostsze rozwiązywanie stosowane do autentykacji połączeń pomiędzy serwisami. Mamy tu od razu mTLS bez potrzeby modyfikacji kodu aplikacji, co zapewnia tożsamość każdego service (service account), bezpieczne połączenia w clustrze i automatyczne zarządzanie kluczami i certyfikatami (automatyczne tworzenie, dystrybucję i odnawianie). W Peer Authentication mamy 3 tryby ustawiania, jaki rodzaj ruchu mTLS Envoy proxy zaakceptuje:

- PERMISSIVE (domyślny) - Envoy proxy w docelowym serwerze zaakceptuje ruch z mTLS, ale też zwykły ruch.
- STRICT - będą zaakceptowane tylko requesty z mTLS.
- DISABLE - ruch musi się odbywać bez mTLS (używane tylko do onboardingu lub testowania).

DestinationRule vs PeerAuthentication

- DestinationRule – The type of TLS traffic the sidecar will send.
- PeerAuthentication – The type of mTLS traffic the sidecar will accept.



Różnica pomiędzy DestinationRule, a PeerAuthentication jest taka, że w DR ustawiamy, jaki rodzaj ruchu z TLS będzie wysłany przez Envoy proxy, a w PeerAuthenticaiton ustawiamy, jaki rodzaj ruchu z

TLS będzie zaakceptowany. W DR jeśli mamy wszystko domyślnie to używamy ISTIO_MUTUAL i to zapewni od razu mTLS, a jak mamy trochę certyfikatów z zewnątrz, lub z naszego innego CA to używamy MUTUAL.

PeerAuthentication example

```
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: "workload2-policy"
  namespace: "staging"
spec:
  selector:
    matchLabels:
      app: test-app
  portLevelMtls:
    80:
      mode: DISABLE
```

Mamy tu przykład Peer Authentication. W takiej policy można np. ustawić, że chcemy wyłączyć TLS tylko na danym porcie, tak jak tutaj na 80.

Request Authentication, jest bardziej skomplikowane, ponieważ tutaj każdy kto chce wykonać request, musi mieć JWT token.

Request Authentication jest używane do autentykacji end-usera poprzez sprawdzenie credentiali dołączonych do requesta. Istio zapewnia Request Authentication przez walidację JSON Web Token (JWT) oraz poprzez interakcję z jakimś authentication providerem lub dowolnym OpenID Connect providerem (np. Google Auth, Firebase Auth, Auth0, Keycloak itp.).

Request Authentication example

The following request authentication policy requires an end-user JWT for the ingress gateway:

```
apiVersion: "security.istio.io/v1beta1"
kind: "RequestAuthentication"
metadata:
  name: "jwt-example"
  namespace: istio-system
spec:
  selector:
    matchLabels:
      istio: ingressgateway
  jwtRules:
  - issuer: "testing@secure.istio.io"
    jwksUri: "https://raw.githubusercontent.com/istio/istio/release-1.5/
```

W powyższym przykładzie wymagany jest JWT od end-usera do połączenia z ingress gatewayem. Ważne jest, że pomimo tego Request Authentication policy, jeśli end-user nie będzie mieć JWT, to domyślnie jego request też będzie przepuszczony - trzeba skonfigurować dodatkowo policy mówiące o tym, żeby takie requesty odrzucać. Token wtedy musi być podany np. w headerze requesta.

Authorisation

Gdy mamy już tożsamość usera lub service, możemy przeprowadzić jego autoryzację i wtedy zdecydować, czy request będzie zaakceptowany, czy odrzucony. Tutaj przydaje się idea Principal - jest to określenie tego skąd bierzemy identity (czyli w zależności od sposobu autentykacji):

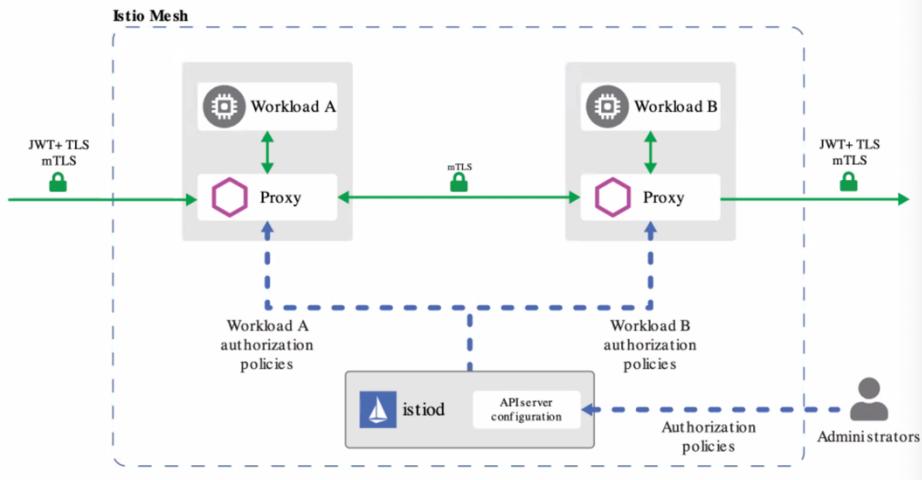
-PeerAuthentication - source.principal (ServiceAccount)

-RequestAuthentication - request.auth.principal (JWT)

Tak samo, jak w przypadku autentykacji, autoryzację w service meshu też możemy ustawić mesh-level, namespace-level i workload-level. Obejmuje to połączenia pomiędzy workloadami, a także od end-

usera do workloadu. Zapewnia to dużą swobodę, bo można ustawiać to różnie w zależności od potrzeb. Dzięki temu mamy dobry performance, bo autoryzacja też jest ustawiana na Envoy proxy i nie trzeba tym się nigdzie indziej zajmować. Jest tu też dobra kompatybilność bo wspierane są protokoły http, https i zwykły tcp. Jest to wszystko realizowane przez `AuthorizationPolicy` resource.

Authorisation architecture



Proces jest całkiem prosty. Tworzymy `AuthorizationPolicy` i tam deklarujemy nasze wymagania. Przesyłane jest to następnie do istiod, który prześle nasze prośby bezpośrednio do Envoy proxy i tam się to ustawi. Należy pamiętać, że wystarczy tylko `AuthorizationPolicy` resource i nie trzeba nic innego ustawiać, żeby wymusić kontrolę dostępu. Jeśli tego nie ustawimy, to wszystko domyślnie będzie puszczone. Jeśli natomiast mamy ustawiony `AuthorizationPolicy` to wszystko domyślnie będzie zablokowane, oprócz tego, co jest ustawione w policy.

Taka policy składa się z selectora (określa on gdzie ta policy będzie dodana), akcji (Deny lub Allow) i listy zasad mówiących, kiedy ta policy ma być brana pod uwagę:

- pole `from` w sekcji `rules` określa źródło requesta, można tu np. użyć `source namespace`.
- pole `to` w sekcji `rules` określa operację, cel requesta
- pole `when` w sekcji `rules` określa warunki, kiedy te zasady będą brane pod uwagę.

Poniżej mamy przykład `AuthorizationPolicy`, która zezwala na pełny dostęp do wszystkich workloadów w default namespace (jeśli dajemy pusty nawias {} to znaczy, że wszystko jest zezwolone):

Allow-all

The example below shows an allow-all policy which allows full access to all workload in the default namespace.

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: allow-all
  namespace: default
spec:
  rules:
  - {}
```

Poniżej natomiast mamy `AuthorizationPolicy` blokującą dostęp do wszystkich workloadów w admin namespace (tutaj różnica jest taka, że nie ma sekcji `rules` i jest sam pusty nawias {} - to znaczy, że nic nie jest dozwolone):

Deny-all

The example below shows a deny-all policy which denies access to all workloads in the admin namespace.

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: deny-all
  namespace: admin
spec:
  {}
```

Jeśli chcemy, aby nasze workloady były publicznie dostępne, to zostawiamy sekcję *source* pustą. To pozwoli wszystkim userom i workloadom (zautentykowanym i nieautentykowanym) na dostęp. Poniżej mamy to ustawione dla aplikacji *httpbin* w namespace *foo*:

Authenticated and unauthenticated identity

To make workloads publicly accessible, the *source* section is left empty.

This allows sources from all (both authenticated and unauthenticated) users and workloads.

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: httpbin
  namespace: foo
spec:
  selector:
    matchLabels:
      app: httpbin
      version: v1
  rules:
  - to:
    - operation:
      methods: [ "GET", "POST" ]
```

Ale jeśli chcielibyśmy tylko zautentykowanych userów wpuścić to trzeba ustawić sekcję *from->source-principals* i ustawić tam wartość ***, co oznacza wszystkich:

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: httpbin
  namespace: foo
spec:
  selector:
    matchLabels:
      app: httpbin
      version: v1
  rules:
  - from:
    - source:
      principals: [ "*" ]
    to:
    - operation:
      methods: [ "GET", "POST" ]
```

Należy pamiętać, że cel policy, czyli to na co jest ona nałożona jest zdeterminowany przez sekcje *metadata->namespace* i *spec->selector* (jeśli chcemy to do konkretnej aplikacji).

Policy Target

Policy scope (target) is determined by metadata/namespace and an optional selector.

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: allow-read
  namespace: default
spec:
  selector:
    matchLabels:
      app: products
  rules:
  - to:
    - operation:
        methods: [ "GET", "HEAD" ]
```

Poniżej mamy pełny przykład. AuthorizationPolicy jest nadana w namespace foo do aplikacji o labeli app: httpbin i version: v1. Policy pozwala na dostęp tylko połączonym z Service Account o nazwie sleep z namespace default, oraz obiektom z namespace dev. Tutaj jeśli w sekcji from mamy ustawione różne sourcy od myślników to jest pomiędzy nimi logiczna wartość OR, czyli jeden lub drugi. Te obiekty mogą wykonać call metodą GET do ścieżki /info*, tylko i wyłącznie wtedy gdy wartość request.auth.claims[groups] (wartość wzięta bezpośrednio z JWT) równa się group1.

Full example

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: httpbin
  namespace: foo
spec:
  selector:
    matchLabels:
      app: httpbin
      version: v1
  rules:
  - from:
    - source:
        principals: [ "cluster.local/ns/default/sa/sleep" ]
    - source:
        namespaces: [ "dev" ]
    to:
    - operation:
        methods: [ "GET" ]
        paths: [ "/info*" ]
  when:
  - key: request.auth.claims[groups]
    values: [ "group1" ]
```

Debugging Istio

Komendą `istioctl ps` sprawdzamy status proxy. Będzie tu pokazany status całego mesha. Wyświetlone zostaną wszystkie pody, które mają zainstalowane Istio. Jest tu pokazana wersja istiod, w kolumnie ISTIOD będzie określony pod istiod, który odpowiada danemu podowi z Kubernetesa. CDS, LDS, EDS, RDS to ustawienia Envoya, w outpcie tej komendy widzimy status, czy odpowiednie dane z istiod są już zsynchronizowane z Envoy proxy dla danego poda.

istioctl proxy-status

\$ istioctl ps	CDS	LDS	EDS	RDS	ISTIOD	VER
NAME						
debug-app-6476bb945-vnr76.internl	SYNCED	SYNCED	SYNCED	SYNCED	istiod-76d66d9876-bapxg	1.1
details-v1-79f774bdb9-vz9m2.bookinfo	SYNCED	SYNCED	SYNCED	SYNCED	istiod-76d66d9876-bapxg	1.1
istio-egressgateway-687f4db598-p2s9f.istio-system	SYNCED	SYNCED	SYNCED	NOT SENT	istiod-76d66d9876-bapxg	1.1
istio-ingressgateway-78f69bd5db-r66ms.istio-system	SYNCED	SYNCED	SYNCED	NOT SENT	istiod-76d66d9876-bapxg	1.1
productpage-v1-b6994bb9-trvd5.bookinfo	SYNCED	SYNCED	SYNCED	SYNCED	istiod-76d66d9876-bapxg	1.1
ratings-v1-545db7b95-jtv7h.bookinfo	SYNCED	SYNCED	SYNCED	SYNCED	istiod-76d66d9876-bapxg	1.1
reviews-v2-7b18c9648f-qw6dm.bookinfo	SYNCED	SYNCED	SYNCED	SYNCED	istiod-76d66d9876-bapxg	1.1
reviews-v3-84779c7bbc-k9rm9.bookinfo	SYNCED	SYNCED	SYNCED	SYNCED	istiod-76d66d9876-bapxg	1.1

Możliwe statusy ustawień Envoya to:

- SYNCHED - najnowsza konfiguracja od istiod jest zsynchronizowana,
- NOT SENT - istiod nie wysłało nic do Envoya, może być tak dlatego, że nie ma co wysłać i tu nie ma błędu,
- STALE - istiod wysłało dane, ale nie otrzymało potwierdzenia o ich przyjęciu. To najczęściej znaczy o problemie sieciowym pomiędzy Envoy i istiod, lub o problemie z Istio.

Inna przydatna komenda to *istioctl analyze*. Pozwala ona na wykrywanie potencjalnych problemów z konfiguracją w Istio (w żywym clustrze lub w plikach konfiguracyjnych), jest to operacja read-only, więc nic nie zepsuje. Można puścić tą komendę na cały cluster, albo tylko w określonym katalogu (najlepiej tam, gdzie mamy konfiguracje Istio).

istioctl analyze output example

```
$ istioctl analyze --recursive networking
Error [IST0101] (VirtualService test/bookinfo networking/bookinfo-gateway.yaml:39) Referenced host not found: "productpage"
Error [IST0145] (Gateway test/bookinfo-gateway networking/bookinfo-gateway.yaml:1) Conflict with gateways istio-system/cert-manager-gateway (workload selector istio=ingressgateway, port 80, hosts *).
Warning [IST0117] (Deployment test/demo-app-v1) No service associated with this deployment. Service mesh deployments must be associated with a service. Warning [IST0140] (VirtualService test/bookinfo networking/bookinfo-gateway.yaml:16) Subset in virtual service test/productpage has no effect on ingress gateway test/bookinfo requests Error: Analyzers found issues when analyzing namespace: test. See https://istio.io/v1.12/docs/reference/config/analysis for more information about causes and resolutions.
```

Inna komenda to *istioctl experimental describe*. Ta komenda pokazuje więcej informacji o danym obiekcie np. tak jak poniżej, o podzie. Pokazuje np. porty na których można się połączyć z poszczególnymi kontenerami, rodzaj Istio TLS itp.

istioctl experimental describe

experimental features may be modified or deprecated

```
$ istioctl experimental describe pod ratings-v1-f745cf57b-qrx12
Pod: ratings-v1-f745cf57b-qrx12
  Pod Ports: 9080 (ratings), 15090 (istio-proxy)
-----
Service: ratings
  Port: http 9080/HTTP
DestinationRule: ratings for "ratings"
  Matching subsets: v1
    (Non-matching subsets v2,v2-mysql,v2-mysql-vm)
  Traffic Policy TLS Mode: ISTIO_MUTUAL
```

Przydatne w sprawdzaniu i troubleshootingu połączeń sieciowych jest sprawdzanie logów z kontenera istio-proxy, który jest w danym podzie. Należy jednak pamiętać, że domyślnie te logi są wyłączone w

sposób mało szczegółowy i żeby mieć więcej informacji należy dodać *meshConfig.accessLogFile*: */dev/stdout*, który uruchomi te logi globalnie.

```
$ kubectl -n istio-system get cm istio -o yaml | grep accessLog  
    accessLogFile: /dev/stdout
```

Możemy też te logi oglądać w JSONie i wtedy będziemy mieć więcej szczegółów np. w punkcie *response_flag* możemy znaleźć kod błędu:

```
{  
    "protocol": "HTTP/1.1",  
    "duration": 16,  
    "upstream_local_address": "10.60.2.8:43826",  
    "response_flags": "-",  
    "response_code": 418,  
    "path": "/status/418",  
    "upstream_transport_failure_reason": null,  
    "x_forwarded_for": null,  
    "upstream_host": "10.60.1.7:80",  
    "upstream_cluster": "outbound|8000||httpbin.default.svc.cluster.local",  
    "connection_termination_details": null,  
    "start_time": "2021-07-29T08:55:33.467Z",  
    "request_id": "9c9ecb6e-c240-4d17-b8c3-7bbac329faa2",  
    "user_agent": "curl/7.78.0-DEV",  
    "upstream_service_time": "15",  
    "response_code_details": "via_upstream",  
    "requested_server_name": null,  
    "method": "GET",  
    "bytes_sent": 135,  
    "authority": "httpbin:8000",  
}
```

Na przykładzie powyżej nie ma żadnego, ale najpopularniejsze dostępne kody błędów to:

- UH - no healthy upstream, czyli 503
- UO - upstream overflow - za duży ruch przychodzący
- NR - no route configured, czyli 404
- NC - upstream cluster not found
- DT - kiedy request osiągnie max_connection_duration (czyli taki timeout)

Jest też opcja uruchomienia Debug Log, ale jest droga i bardzo obciąża system, więc nie uruchamiamy jej na całym clustrze. Można też np. zwiększyć szczegółowość niektórych rodzajów logów. Poniżej mamy przykładowe komendy do tego:

Istioctl proxy-config log <pod_name[.namespace]> - wyświetlenie informacji na temat obecnych ustawień szczegółowości logów dla danego poda w namespace

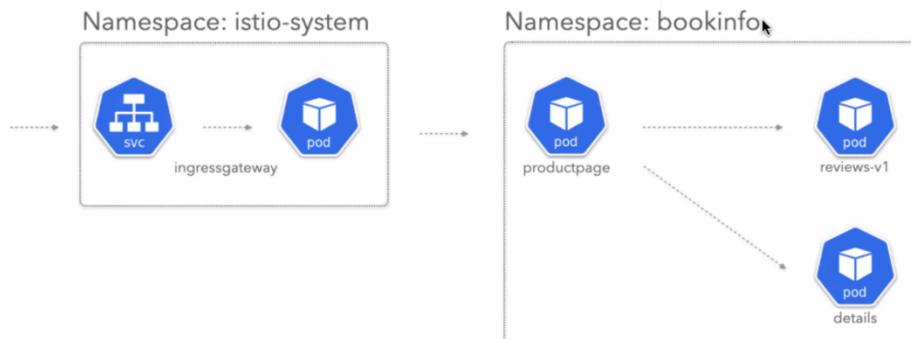
Istioctl proxy-config log <pod_name[.namespace]> --level none - zmiana wszystkich logów dla poda na *none*, czyli wyłączenie

Istioctl proxy-config log <pod_name[.namespace]> --level http:debug - zmiana określonej wartości logów tu http na debug

Istioctl proxy-config log <pod_name[.namespace]> -r - restart wszystkich wartości logów, do domyślnych wartości

Life of a Packet

The Environment



Tak wygląda nasze środowisko. Mamy w istio-system namespace ingressgateway (składający się z service i poda), do którego idzie ruch z zewnątrz, a także nasz bookinfo namespace, w którym jest pod productpage, który dalej będzie rozdzielał ruch pomiędzy reviews i details.

Na początku tworzymy obiekt Gateway (czyli konfigurację ingressgatewaya), w którym ustawiamy ingressgateway jako selector i zezwalamy w nim na dostęp do hosta bookinfo.com na porcie 80.

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: bookinfo-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "bookinfo.com"
```

Na obrazku poniżej wyświetlamy listenery ingressgatewaya. Listener to port, na którym ingressgateway zaakceptuje request:

The Ingress Gateway

```
$ ./istioctl -n istio-system proxy-config listener istio-ingressgateway-78f69bd5db-rmsq4
ADDRESS PORT MATCH DESTINATION
0.0.0.0 8080 ALL Route: http.8080
0.0.0.0 15021 ALL Inline Route: /healthz/ready*
0.0.0.0 15090 ALL Inline Route: /stats/prometheus*
```

Pomimo, że ustaliliśmy port 80 tu jest 8080. Dzieje się tak dlatego, że przed podem ingressgatewaya mamy jeszcze jego Service, który zbiera ruch z zewnątrz i przesyła go do poda. Poniżej mamy jego domyślną konfigurację i widzimy, że w rulach zewnętrzny port to 80 i potem ten ruch jest przemieniany wewnętrzny na 8080:

The Ingress Gateway Service

```
apiVersion: v1
kind: Service
metadata:
  labels:
    istio: ingressgateway
    name: istio-ingressgateway
    namespace: istio-system
spec:
  ports:
  - name: http2
    nodePort: 31364
    port: 80
    targetPort: 8080
  - name: https
    nodePort: 32414
    port: 443
    targetPort: 8443
  selector:
    app: istio-ingressgateway
    istio: ingressgateway
    type: LoadBalancer
  status:
    loadBalancer:
      ingress:
      - ip: 34.105.169.239
```

W konfiguracji zewnętrznego service widzimy, że jest on typu LoadBalancer i że ma nadany adres IP od providera (tu gdzie jest zainstalowany Kubernetes). Aby móc się bezpośrednio łączyć do naszej aplikacji musimy połączyć domenę bookinfo.com w serwerze DNS z tym adresem IP. Jak teraz zrobimy test to wyjdzie 404, bo Istio nie wie, jak przekazać dalej request:

```
$ curl -svo /dev/null http://34.105.169.239/productpage -H "Host: bookinfo.com"
*   Trying 34.105.169.239:80...
*   Connected to 34.105.169.239 (34.105.169.239) port 80 (#0)
> GET /productpage HTTP/1.1
> Host: bookinfo.com
> User-Agent: curl/7.77.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 404 Not Found
< date: Thu, 27 Jan 2022 15:41:28 GMT
< server: istio-envoy
< content-length: 0
<
* Connection #0 to host 34.105.169.239 left intact
```

Aby ustawić dalsze kroki (routing) tworzymy Virtual Service oraz Destination Rule. W VS mamy ustawione różne ścieżki do różnych aplikacji które będą zaakceptowane i następnie routowane.

Dodatkowo jako następny cel ruchu dla tych ścieżek ustawiamy hosta productpage (czyli nasz frontendowy pod).

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: bookinfo
  namespace: bookinfo
spec:
  hosts:
  - "bookinfo.com"
  gateways:
  - bookinfo-gateway
  http:
  - match:
    - uri:
        exact: /productpage
    - uri:
        prefix: /static
    - uri:
        exact: /login
    - uri:
        exact: /logout
    - uri:
        prefix: /api/v1/products
  route:
  - destination:
      host: productpage
      port:
        number: 9080
```

Teraz możemy sprawdzić routy dla ingressgatewaya i tu mamy te wszystkie ścieżki przekazane do VS:

```
$ ./istioctl -n istio-system proxy-config route istio-ingressgateway-78f69bd5db-rmsq4
NAME      DOMAINS      MATCH          VIRTUAL SERVICE
http.8080 bookinfo.com  /productpage  bookinfo.bookinfo
http.8080 bookinfo.com  /static*       bookinfo.bookinfo
http.8080 bookinfo.com  /login         bookinfo.bookinfo
http.8080 bookinfo.com  /logout        bookinfo.bookinfo
http.8080 bookinfo.com  /api/v1/products* bookinfo.bookinfo
*           *             /stats/prometheus*
*           *             /healthz/ready*
```

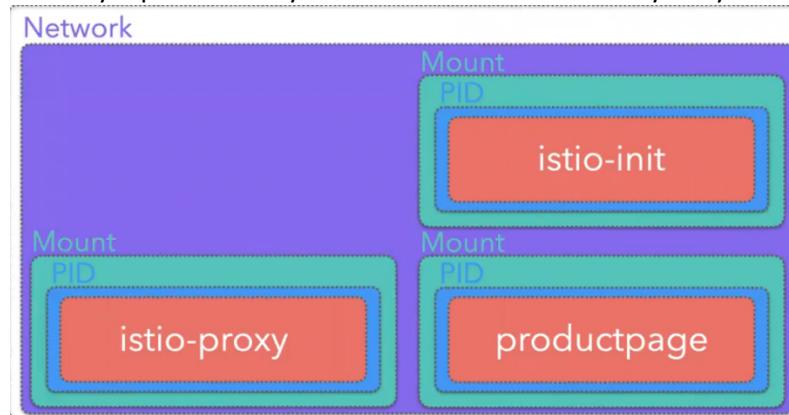
Można też otrzymać dokładniejszy output dodając -o json:

```
$ ./istioctl -n istio-system proxy-config route istio-ingressgateway-78f69bd5db-rmsq4 --name http.8080 -o
[{"name": "http.8080", "virtualHosts": [{"name": "bookinfo.com:80", "domains": ["bookinfo.com", "bookinfo.com:""], "routes": [{"match": {"path": "/productpage", "caseSensitive": true}, "route": {"cluster": "outbound|9080||productpage.bookinfo.svc.cluster.local", "weight": 100}}]}]
```

Jeśli sprawdzimy endpointy ingressgatwuya, to zobaczymy naszą nową stworzoną rule, czyli przekazanie do productpage.booking.svc.cluster.local:

```
$ ./istioctl -n istio-system proxy-config endpoint istio-ingressgateway-78f69bd5db-rmsq4 \
--cluster "outbound|9080||productpage.bookinfo.svc.cluster.local"
ENDPOINT STATUS OUTLIER CHECK CLUSTER
10.104.3.13:9080 HEALTHY OK outbound|9080||productpage.bookinfo.svc.cluster.local
```

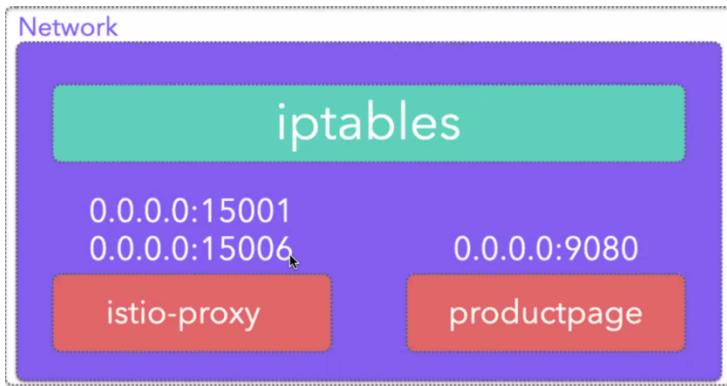
Teraz musimy ustawić ruch wewnętrz aplikacyjnego namespace, w którym mamy już sidecar pattern. W każdym podzie mamy dodatkowe istiowe kontenery. Przykładowo poniżej nasz productpage pod:



Istio-init służy tylko do przechwycenia ruchu do poda i modyfikacji iptables, poniżej mamy jego parametry:

```
initContainers:
- args:
  - istio-iptables
  - -p
  - -z
  - -15001
  - -15006
  - -u
  - -1337
  - -m
  - REDIRECT
  - -i
  - '*'
  - -X
  - ""
  - -b
  - '*'
  - -d
  - 15090,15021,15020
  image: docker.io/istio/proxyv2:1.12.1
```

Iptables będzie wyglądać w ten sposób. Mamy dwa porty, bo jeden jest do przechwycania ruchu inbound, a drugi do outbound:



Jak pakiet dostanie się do iptables, to pierwszy chain (rodzaj routowania), który dostanie to PREROUTING RULE. Jest to domyślne i tam mamy takie ustawienie, które sprawia, że każdy ruch przychodzący i wychodzący wrzucamy do tablicy ISTIO_INBOUND:

```

Chain PREROUTING (policy ACCEPT)
target      prot opt source          destination
ISTIO_INBOUND  tcp  --  anywhere       anywhere

```

Tak wygląda ISTIO_INBOUND. W tej tabeli, wszystko co ma wartość RETURN nie będzie przepuszczone, czyli port 15008, ssh itp. (my akurat mamy port 9080 do poda, więc pójdzie dalej). Na końcu jest rula mówiąca, że reszta dalej będzie przekazana do ISTIO_IN_REDIRECT:

```

Chain ISTIO_INBOUND (1 references)
target      prot opt source          destination
RETURN     tcp  --  anywhere       anywhere           tcp  dpt:15008
RETURN     tcp  --  anywhere       anywhere           tcp  dpt:ssh
RETURN     tcp  --  anywhere       anywhere           tcp  dpt:15090
RETURN     tcp  --  anywhere       anywhere           tcp  dpt:15021
RETURN     tcp  --  anywhere       anywhere           tcp  dpt:15020
ISTIO_IN_REDIRECT  tcp  --  anywhere       anywhere

```

W ISTIO_IN_REDIRECT mamy jedną rulę mówiącą, że każdy ruch będzie przekazany na port 15006:

```

Chain ISTIO_IN_REDIRECT (3 references)
target      prot opt source          destination
REDIRECT   tcp  --  anywhere       anywhere           redir ports 15006

```

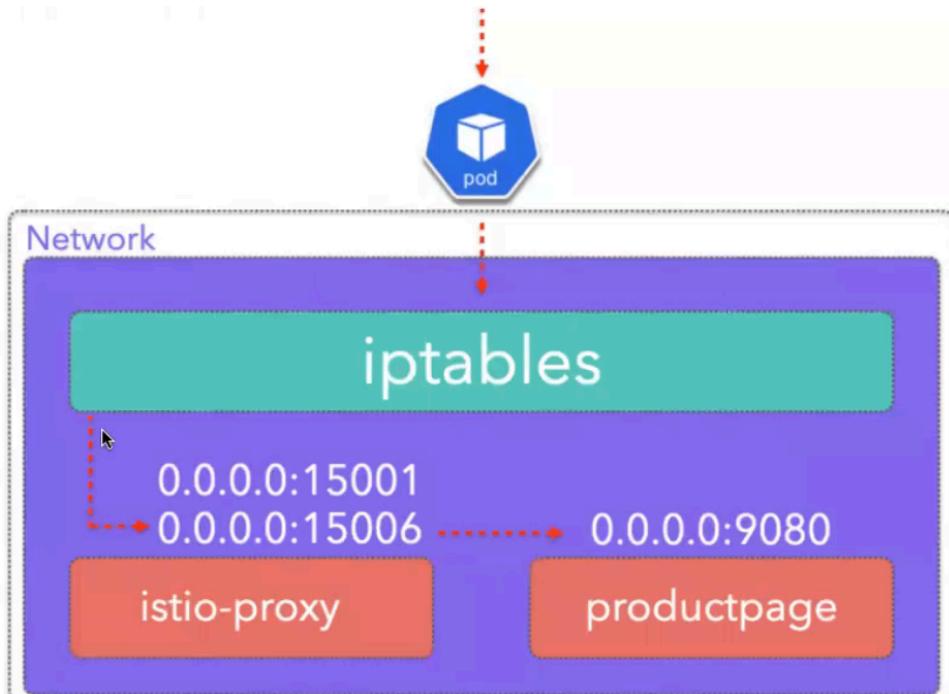
W ten sposób Envoy przechwytuje ruch. Poniżej mamy netstat output z contenera istio-proxy. Mamy tu właśnie port 15006 przypisany do Envoya:

```

istio-proxy@debug-pod:/ $ netstat -plnt
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address          Foreign Address        State      PID/Program name
tcp        0      0 127.0.0.1:15004        0.0.0.0:*
tcp        0      0 0.0.0.0:15006          0.0.0.0:*
tcp        0      0 0.0.0.0:15006          0.0.0.0:*
tcp        0      0 0.0.0.0:15021          0.0.0.0:*
tcp        0      0 0.0.0.0:15021          0.0.0.0:*
tcp        0      0 0.0.0.0:15090          0.0.0.0:*
tcp        0      0 0.0.0.0:15090          0.0.0.0:*
tcp        0      0 127.0.0.1:15000        0.0.0.0:*
tcp        0      0 0.0.0.0:15001          0.0.0.0:*
tcp        0      0 0.0.0.0:15001          0.0.0.0:*
tcp6       0      0 :::15020             :::*                  LISTEN     1/pilot-agent

```

Te ostatnie przechwycenia możemy przedstawić na tym schemacie:



Request mógłby być zablokowany w istio-proxy, gdybyśmy nadali jakiś Authorization Policy, ale tutaj jej nie ma, więc idzie dalej. Poniżej mamy parametry Envoy, które ma w trakcie sprawdzania naszego requesta. PID Envoya to 19, mamy tu sprecyzowaną binarkę Envoya i plik konfiguracyjny `envoy-rev0.json`:

Envoy Config

```
istio-proxy@debug-pod:/# cat /proc/19/cmdline
/usr/local/bin/envoy-cetc/istio/proxy/envoy-rev0.json--restart-epoch0--drain-time-s45-
-drain-strategyimmediate--parent-shutdown-time-s60--local-address-ip-versionv4--file-f
lush-interval-msec1000--disable-hot-restart--log-format%Y-%m-%dT%T.%FZ      %l      env
oy %n      %v-lwarning--componen
```

Poniżej sprawdzamy clustery, które jest w stanie rozpoznać `productpage` pod i w tym przypadku konkretnie oglądamy `xds-grpc`. Jest to statyczny cluster i aby móc rozpoznać jego tożsamość trzeba sprawdzić plik `./etc/istio/proxy/XDS`. Ta lokacja mówi nam, że jest to konfiguracja Envoya zarządzana przez pilot-agent i stąd Envoya wie, jak się komunikować z Istio.

Istioctl Inspection

```
$ ./istioctl proxy-config cluster productpage-v1-6b746f74dc-96z9w --fqdn xds-grpc -o json | jq
[
  {
    "name": "xds-grpc",
    "type": "STATIC",
    "connectTimeout": "1s",
    "loadAssignment": {
      "clusterName": "xds-grpc",
      "endpoints": [
        {
          "lbEndpoints": [
            {
              "endpoint": {
                "address": {
                  "pipe": [
                    "path": "./etc/istio/proxy/XDS"

```

Poniżej mamy poda z kontenerem `istio-proxy` i widzimy, że jest w nim tworzony mount do `/etc/istio/proxy`. To właśnie ten sam katalog, w którym też jest XDS i konfiguracja Envoya. Właśnie tam są wszystkie ustawienia dla proxy.

```
apiVersion: v1
kind: Pod
...
spec:
  containers:
    - name: istio-proxy
      image: docker.io/istio/proxyv2:1.12.1
      volumeMounts:
        - mountPath: /etc/istio/proxy
          name: istio-envoy
  volumes:
    - emptyDir:
        medium: Memory
        name: istio-envoy
...

```

Teraz wracając do naszej aplikacji trzeba stworzyć VirtualService, aby ruch z productpage mógł iść do reviews. Ustawiamy tu odpowiedni subset i host. Czyli, jak jakiś ruch przyjdzie na hosta reviews to wysyłamy do subsetu v1, który jest ustawiony w DestinationRule poniżej:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
    - reviews
  http:
    - route:
        - destination:
            host: reviews
            subset: v1

```

Tutaj precyzujemy, że w subsecie v1 będą pody z labelką version:v1

```

apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: reviews
spec:
  host: reviews
  subsets:
  - name: v1
    labels:
      version: v1

```

Teraz możemy łączyć się do reviews. Poniżej mamy wycinek z Access Loga w komunikacji z productpage do naszego poda reviews:

```
$ kubectl -n bookinfo logs productpage-v1-6b746f74dc-5pk24 -c istio-proxy | grep reviews:9080
[2022-01-28T15:08:40.589Z] "GET /reviews/0 HTTP/1.1" 200 - via_upstream "-" 0 295 1045 1043 "-" "Mozilla
/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/97.0.4692.71 Safari
/537.36" "84f89a42-e6f2-95bd-900c-79943a2f5cf9" "reviews:9080" "10.244.0.160:9080" outbound|9080|v1|review
s.bookinfo.svc.cluster.local 10.244.0.165:60718 10.244.0.160:9080 10.244.0.165:60716 - -
```

Pamiętamy, że Istio przechwytuje cały ruch wejściowy i wyjściowy, więc, tak jak mieliśmy wcześniej, to połączenie przejdzie też przez wszystkie tabele w output chain:

Chain OUTPUT (policy ACCEPT)	target	prot	opt	source	destination
	ISTIO_OUTPUT	tcp	--	anywhere	anywhere

Chain ISTIO_OUTPUT (1 references)	target	prot	opt	source	destination
	RETURN	all	--	127.0.0.6	anywhere
	ISTIO_IN_REDIRECT	all	--	anywhere	!localhost
	RETURN	all	--	anywhere	anywhere
	RETURN	all	--	anywhere	anywhere
	ISTIO_IN_REDIRECT	all	--	anywhere	!localhost
	RETURN	all	--	anywhere	anywhere
	RETURN	all	--	anywhere	anywhere
	RETURN	all	--	anywhere	localhost
	ISTIO_REDIRECT	all	--	anywhere	anywhere

Chain ISTIO_REDIRECT (1 references)	target	prot	opt	source	destination
	REDIRECT	tcp	--	anywhere	anywhere
					redir ports 15001

1337 to UID pilot-agenta i ustawione jest tak, aby cały ruch był przechwytywany, oprócz pilot-agenta. Na końcu wyjście do Envoy na porcie 15001.

Poniżej sprawdzamy, jakie listenery zna pod productpage i mamy tutaj nasz dodany listener do poda reviews po porcie 9080:

```
$ ./istioctl proxy-config listener productpage-v1-6b746f74dc-5pk24 --port 9080
ADDRESS PORT MATCH                               DESTINATION
0.0.0.0 9080 Trans: raw_buffer; App: HTTP Route: 9080
0.0.0.0 9080 ALL                                PassthroughCluster
```

Jak spojrzymy na znajome mu routy to zobaczymy kilka możliwych z różnymi domenami i jest tu nasz reviews:

```
$ ./istioctl proxy-config route productpage-v1-6b746f74dc-5pk24 --name 9080
NAME      DOMAINS          MATCH      VIRTUAL SERVICE
9080      details, details.bookinfo + 1 more...  /*        details.bookinfo
9080      productpage, productpage.bookinfo + 1 more... /*        productpage.bookinfo
9080      reviews, reviews.bookinfo + 1 more...   /*        reviews.bookinfo
```

Poniżej mamy wszystkie domeny, które zadziaiają w przypadku tego połączenia i wszystko to zostanie przerzucone do reviews.bookinfo.svc.cluster.local tak jak mamy na dole w route:

```

  "name": "reviews.bookinfo.svc.cluster.local:9080",
  "domains": [
    "reviews.bookinfo.svc.cluster.local",
    "reviews.bookinfo.svc.cluster.local:9080",
    "reviews",
    "reviews:9080",
    "reviews.bookinfo.svc",
    "reviews.bookinfo.svc:9080",
    "reviews.bookinfo",
    "reviews.bookinfo:9080",
    "10.105.82.232",
    "10.105.82.232:9080"
  ],
  "routes": [
    {
      "match": {
        "prefix": "/"
      },
      "route": {
        "cluster": "outbound|9080|v1|reviews.bookinfo.svc.cluster.local",
        ...
      }
    }
  ]
}

```

Tutaj widzimy endpointy dla poda productpage i tutaj już jest ustawione IP poda końcowego:

```

$ ./istioctl proxy-config endpoint productpage-v1-65b75f6885-cgf7l \
--cluster "outbound|9080|v1|reviews.bookinfo.svc.cluster.local"
ENDPOINT           STATUS     OUTLIER CHECK   CLUSTER
10.244.0.190:9080   HEALTHY     OK          outbound|9080|v1|reviews.bookinfo.svc.cluster.local

```

Istio Performance

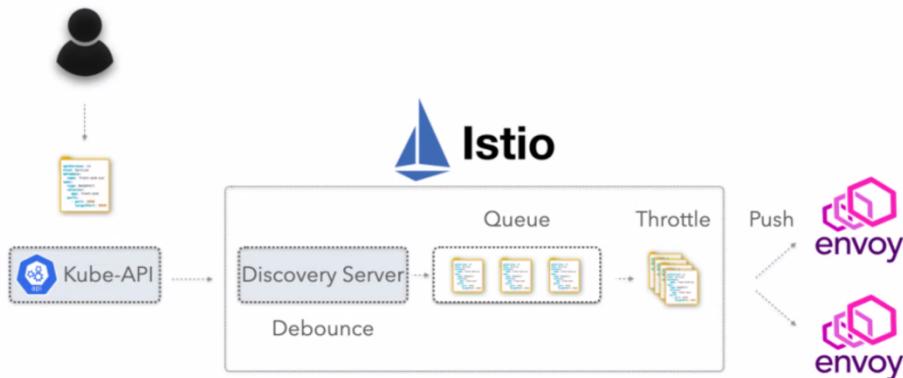
Średnie wyniki testów na performance Istio to:

- Envoy proxy używa 0.35 vCPU i 40MB pamięci na 1000 requestów przechodzących w sekundę przez proxy
- Istio używa 1vCPU i 1.5GB pamięci
- Envoy proxy dodaje około 2.56 ms do opóźnienia połączenia

Performance data plane (czyli proxy poinstalowanych w podach) zależy od ilości przychodzących requestów, rozmiaru requestów i odpowiedzi, liczby proxy pracujących w clustrze, protokołu, ilości rdzeni CPU i ilości zastosowanych filtrów w proxy. Na wiele z tych rzeczy nie mamy wpływu, jeśli chcemy polepszyć performance to można uprościć trochę filtry i dodać więcej instancji proxy.

Performance control plane zależy od tego jak często występują zmiany w deploymentach (Istio ciągle monitoruje pody, service itp.) i konfiguracjach (im więcej rzeczy w clustrze tym więcej konfiguracji), ilości proxy, które łączą się z Istiod i od rozmiaru konfiguracji Envoya, które muszą być przesłane do proxy. Im więcej, tym wzrasta zużycie CPU i pamięci.

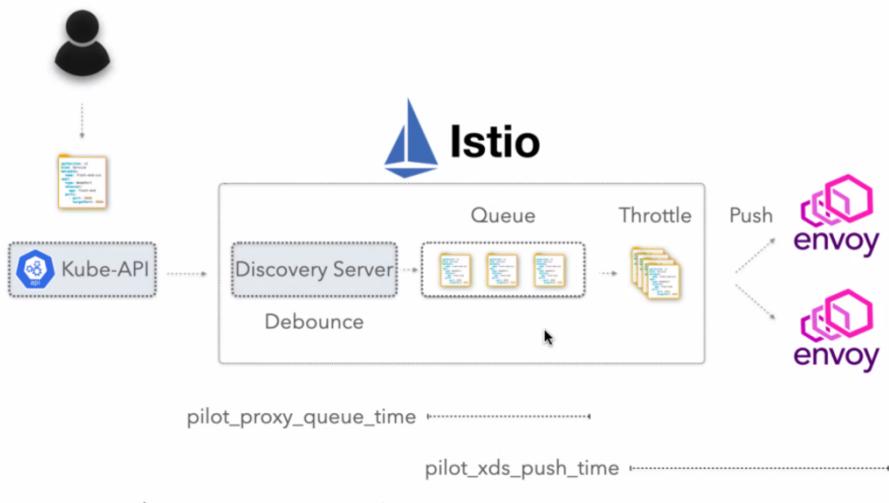
Eventual Consistency



W Istiod jest Discover Server, jest to kontroler, który sprawdza zmiany w Kubernetes clustrze. Wszystkie wykrywane zmiany trafiają najpierw do Debounce, jest to wstępna faza, w której zmiany są sprawdzane i jeśli się nakładają na siebie to jest z nich tworzona pojedyncza uwzględniająca wszystko. Jeśli natomiast są jakieś konflikty w zmianach, to w życie wejdzie tylko najnowszy request. Następnie takie przetworzone zmiany trafiają do Queue. Debounce jest po to, żeby nie przeciążyć Istiod. Z kolejki zmian następnie są wypychane bezpośrednio do konfiguracji Envoya, ale tak, by nie przeciążyć systemu. Procesowanie zmian jedna po drugiej jest szybsze, ale nie jest to dobre dla performance. Lepiej jedną większą paczkę, ale rzadziej.

Control Plane Metrics

Control Plane Latency



Na opóźnienie w Control Plane składają się następujące czasy:

Pilot_proxy_convergence_time - czas od momentu wejścia w kolejkę do dotarcia konfiguracji do Envoy proxy.

Pilot_proxy_queue_time - czas, w którym requesty czekają w kolejce.

Pilot_xds_push_time - czas na wypchnięcie konfiguracji do Envoy proxy.

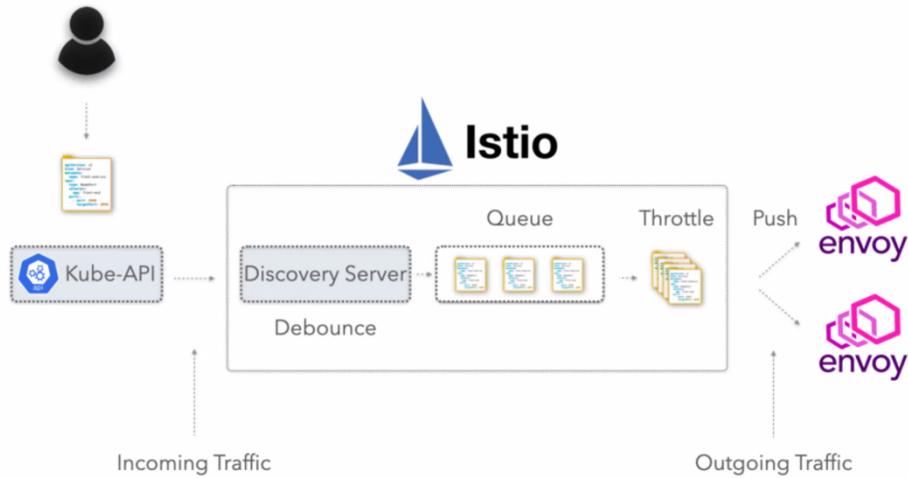
Należy pamiętać, że te metryki nie biorą pod uwagę momentu Debounce.

Istio jest bardzo wrażliwe i pobiera dużo CPU, zwłaszcza gdy dużo dzieje się na clustrze. Jeśli chodzi o saturację mamy rozróżniamy:

Container_cpu_usage_seconds_total - mierzy użycie CPU, tak jak to rapportuje Kubernetes container.

Process_cpu_seconds_total - mierzy użycie CPU, tak jak to rapportuje Istiod.

Control Plane Traffic



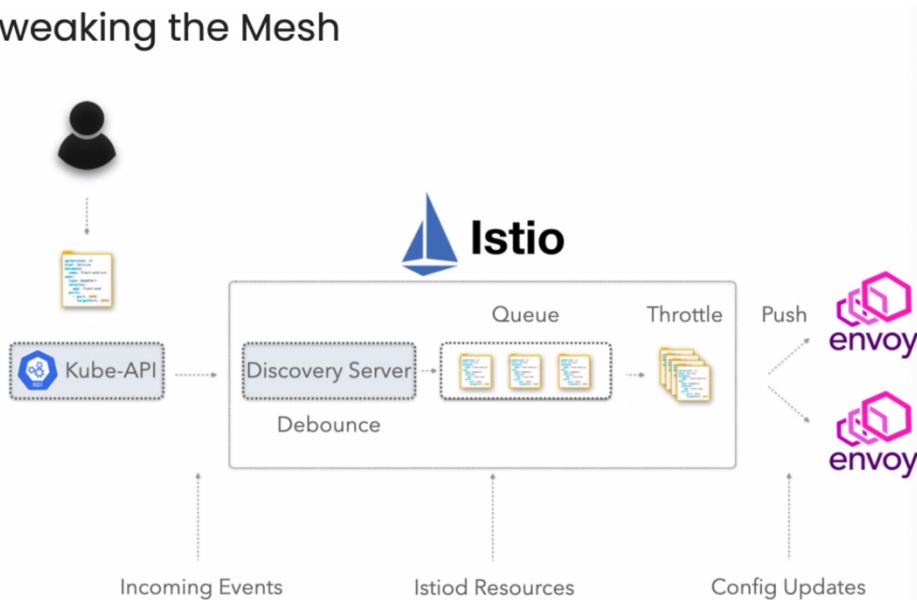
Biorąc pod uwagę ruch przychodzący i wychodzący z Istio mamy:

- incoming: pilot_inbound_updates (ilość zmian konfiguracji otrzymanych na instancję istiod), pilot_push_triggers (łączna liczba eventów, które wywołały request np. tu liczone są service, endpointy, configi), pilot_services (liczba service znana pilotowi. Żeby zmniejszyć tą wartość można zmniejszyć ilość service znanych dla pilota).
- outbound: pilot_xds_pushes (mierzy wszystkie rodzaje requestów i zmian np. listenery, routes, clusters, endpoint updates), pilot_xds_metric (łączna ilość połączeń do workloadów na instancję pilota), envoy_cluster_upstream_cx_tx_bytes_total (rozmiar konfiguracji w bajtach, który jest przesyłany przez sieć).

Poniżej mamy błędy Envoya występujące np. gdy nowe konfiguracje będą odrzucane. Mamy tu kilka metryk, od bardziej ogólnych, do takich, które mówią, czy był to reject np. endpointu, listenera, route, clustra itp:

- pilot_total_xds_rejects - Rejected configuration pushes
- pilot_xds_eds_reject
- pilot_xds_lds_reject
- pilot_xds_rds_reject
- pilot_xds_cds_reject
- pilot_xds_write_timeout - The sum of errors and timeouts when initiating a push
- pilot_xds_push_context_errors -The count of Istio Pilot errors while generating the envoy configuration, usually bugs in the Istio Pilot

Tweaking the Mesh



Jak już mamy metryki to można się zastanowić nad tym co możemy samemu zmienić aby poprawić performance. No i możemy np. zwiększyć Istiod Resources, czyli zwiększyć RAM itp. Możemy też ograniczyć ruch przychodzący i wychodzący z Istio. Należy pamiętać, że domyślnie Istiod odkrywa wszystkie Kubernetesowe endpointy, co może skutkować w ogromnych konfiguracjach wysyłanych do Envoya. Do kontrolowania tego co jest wysyłane do Envoya możemy użyć sidecar resource. Może on być dodany na poziomie mesh-wide, namespace lub samego workloadu. Pozwala on np. w danym namespace kontrolować tylko zmiany w wybranych namespaces i nie przejmować się tym, co dzieje się w innych.

Poniżej mamy przykład mesh-wide sidecar resource:

```
apiVersion: networking.istio.io/v1beta1
kind: Sidecar
metadata:
  name: default
  namespace: istio-system
spec:
  egress:
  - hosts:
    - "istio-system/*"
  outboundTrafficPolicy:
    mode: REGISTRY_ONLY
```

W tym przypadku w spec->egress->hosts mamy tylko istio-system namespace, to znaczy, że obiekty z tym sidecarem będą tylko śledzić zmiany w tym konkretnym namespace, a inne już ich nie będą obchodzić, co ograniczy ruch i zużycie CPU. Mode REGISTRY_ONLY mówi o monitorowaniu tylko ruchu tutaj ustawionego.

Należy też pamiętać, że domyślnie Istio sprawdza wszystkie eventy dzierżające się na obiektach Kubernetesa, takich jak pody czy service. Tu z pomocą przychodzi nam Namespace Discovery Selector, który pomoże w filtrowaniu tylko tego, co chcemy. Poniżej mamy przykład takiego selectora, który mówi o monitorowaniu tylko namespacesów z labelą mesh:include.

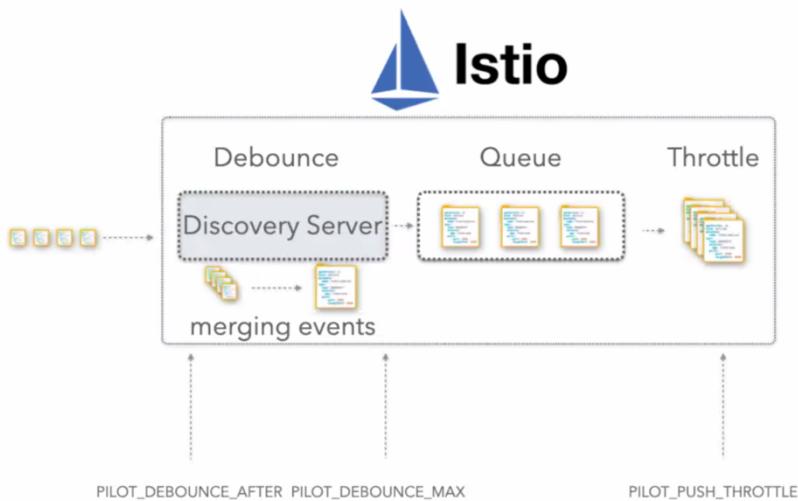
```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
metadata:
  namespace: istio-system
spec:
  meshConfig:
    discoverySelectors:
      - matchLabels:
          mesh: "include"
```

Poniżej natomiast mamy warunek na to samo, ale skonfigurowane w inny sposób:

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
metadata:
  namespace: istio-system
spec:
  meshConfig:
    discoverySelectors:
      - matchExpressions:
          - key: mesh
            operator: NotIn
            values:
              - "exclude"
```

DiscoverSelector i Sidecar są podobnymi resourcami. Pierwszy z nich umożliwia ustawienie, które namespacy będą w meshu, a drugi kontroluje to, które konfiguracje sidecarów będą widoczne w Envoy proxy. Można to tak podzielić, że DiscoverSelector sprawdza incoming traffic a sidecar outgoing. Jeśli cos jest wyłączone w Discovery Selectorze, a chcemy to w sidecarze to i tak tego nie będzie, bo Istio nie będzie tego widzieć i żeby było monitorowane to w tych obu musi być zatwierdzone.

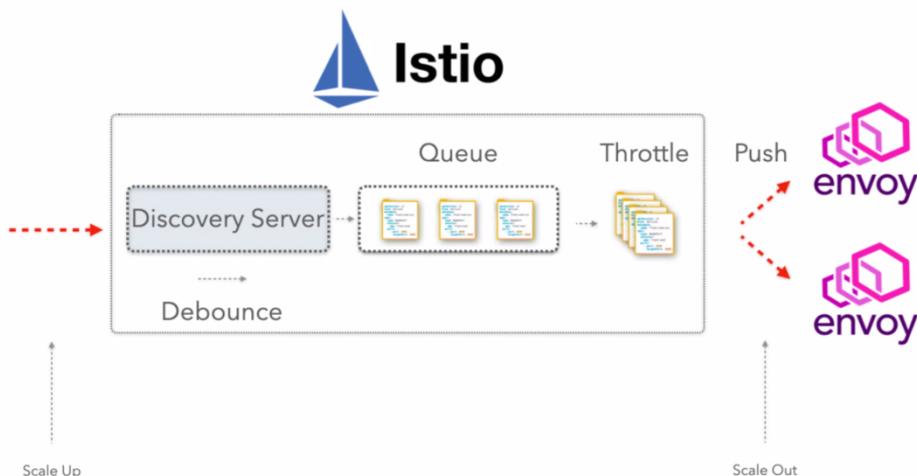
Debounce and Push



Jeśli chodzi o Debounce to nie mamy tu metryk, ale mamy opcje, których możemy użyć:

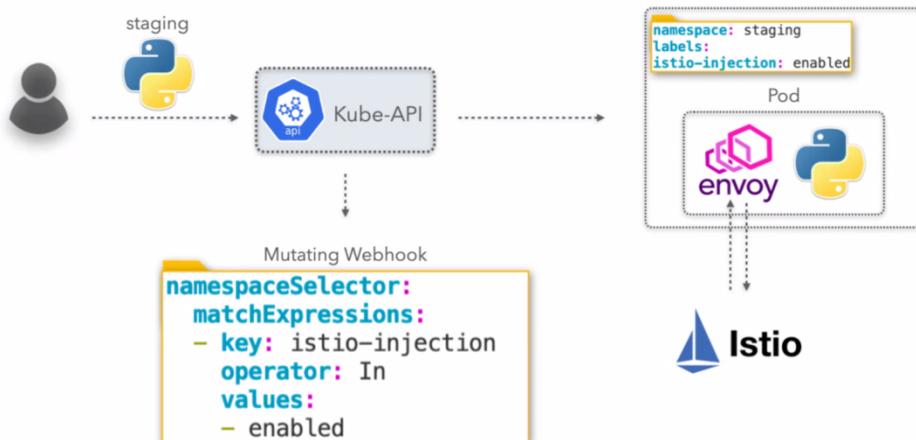
- `pilot_debounce_after` - ile czasu request będzie w fazie debounce przed wysłaniem do queue. Domyślnie to 100ms.
- `pilot_debounce_max` - maksymalny czas, w którym debounce jest zezwolone. Domyślnie to 10s.
- `pilot_push_throttle` - ile requestów Istio może procesować na raz. Domyślnie to 100.
- `pilot_enable_eds_debounce` - kontroluje, czy update endpointu może ominąć debounce. Domyślnie to True, to znaczy, że nie omija Debounce. Jeśli chcemy omijać, to możemy w to miejsce dać RDS.

Scale Up or Out?

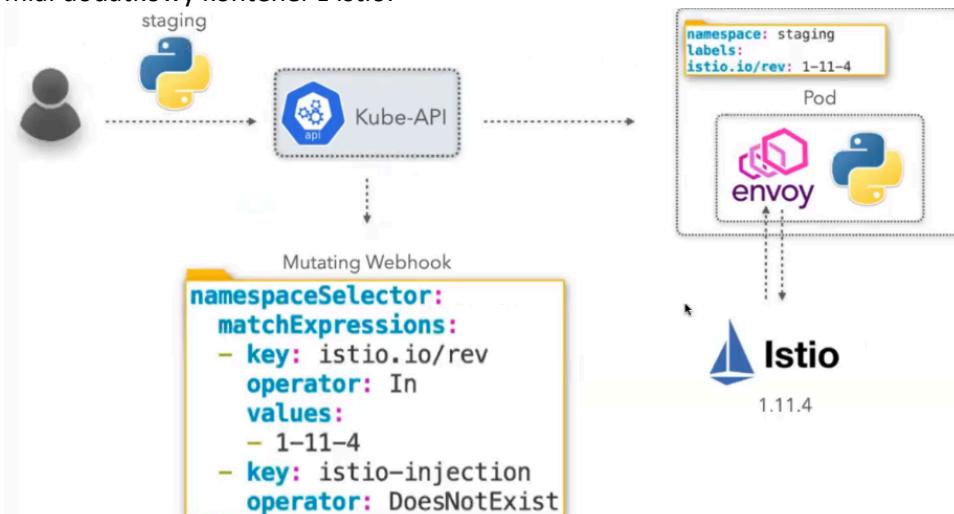


Mogimy też skalować instancje Istio. Scale Up to dodanie CPU, memory itp., a Scale Out to dodanie większej ilości instancji istiod. Jeśli bottle neck jest przy wchodzącym ruchu to dobrze rozważyć Scale Up, a jak przy wychodzącym ruchu i np. już ograniczyliśmy go to Sidecarom, a nie pomogło to dobrze rozważyć Scale Out.

Canary Upgrade Sidecar Injection



Cieźko jest zrobić Canary Upgrade samego istiod, ale już zwykłych aplikacji jest łatwiej. Powyżej mamy tworzenie aplikacji, więc request jest przesyłany przez Kube-api do mutating webhooka na podstawie labeli `istio-injection:enabled`, którą nasz staging namespace ma. Na tej podstawie nowy pod będzie miał dodatkowy kontener z Istio.



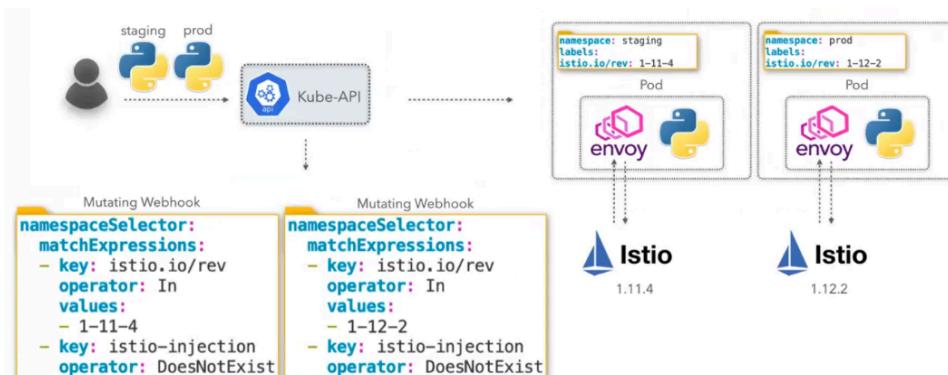
W nowszych wersjach Istio mamy wprowadzony koncept Revisions. Pozwala to oznaczać nasze deploymenty wersjami, tak jak tu `1-11-4`. Tutaj widzimy też, że nasz namespace ma labelę `istio.io/rev:1-11-4` i to pozwala na to żeby zaszedł ten sam proces. Poniżej jest przykład wprowadzenia Revision do IstioOperatora:

```

apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  profile: default
  revision: 1-11-4
  components:
    ingressGateways:
      - name: istio-ingressgateway
        enabled: false
  meshConfig:
    accessLogFile: "/dev/stdout"

```

Dzięki tej możliwości możemy zrobić Canary Upgrade, bo możemy stworzyć kilka wersji Istio nadając różny Revision dla tych instalacji. W efekcie będziemy mieć różne mutating webhooki i będą one sprawdzać różne labelle. Dzięki temu różne pody mogą dostać różne wersje sidecar proxy.



Aby jeszcze uprościć ten proces Istio wprowadziło tagi. Dzięki temu można otagować określony Revision. Tagujemy tutaj określony Revision np. 1-11-4 ma taga prod-stable. Teraz, aby uruchomić ten mutating webhook w danym namespace wystarczy dodać labelę `istio.io/rev=prod-stable` i to zadziała w ten sam sposób co wyżej.

Introducing Tags

```

$ istioctl tag set prod-stable --revision 1-11-4
Revision tag "prod-stable" created, referencing control plane revision
"1-11-4".
To enable injection using this revision tag, use
'kubectl label namespace <NAMESPACE> istio.io/rev=prod-stable'

$ istioctl tag set prod-canary --revision 1-12-2
Revision tag "prod-canary" created, referencing control plane revision
"1-12-2".
To enable injection using this revision tag, use
'kubectl label namespace <NAMESPACE> istio.io/rev=prod-canary'

```

```

namespaceSelector:
  matchExpressions:
    - key: istio.io/rev
      operator: In
      values:
        - prod-stable
    - key: istio-injection
      operator: DoesNotExist

```

```

namespaceSelector:
  matchExpressions:
    - key: istio.io/rev
      operator: In
      values:
        - prod-canary
    - key: istio-injection
      operator: DoesNotExist

```

Jeśli zrobimy tag z nazwą default, to będzie on uznawany za domyślny Revision. Ma on wtedy dodatkowe funkcje:

- dodaje Istio sidecars w każdym namespace, gdzie mamy labelę istio-injection=enabled, sidebar.istio.io/inject=true object selector lub istio.io/rev=default selector.

- dodatkowo waliduje on obiekty Istio.

Jeśli robimy Canary Upgrade to dobrym rozwiązaniem jest oddzielenie Ingress Gatewya od Istiod. Dalej to większą kontrolę w przypadku upgradu samego Ingress Gatewya, ponieważ jak nie oddzielimy, to będzie on upgradowany razem z Istiod.

Poniżej mamy przykład deploymentu Ingress Gatewya osobno od Istiod. Używamy tutaj empty profile i mówimy tylko w komponentach żeby stworzyć sam ingressgateway (podajemy jego namespace label). Ostatnia rzecz to injectionTemplate: gateway, pozwala to na automatyczne wprowadzanie nowych wersji gatewya w życie, gdy będą dostępne.

```

apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
metadata:
  name: ingress
spec:
  profile: empty # Do not install CRDs or the control plane
  components:
    ingressGateways:
      - name: ingressgateway
        namespace: istio-ingress
        enabled: true
        label:
          # Set a unique label for the gateway.
          istio: ingressgateway
  values:
    gateways:
      istio-ingressgateway:
        # Enable gateway injection
        injectionTemplate: gateway

```

Poniżej mamy przykład istniejącego deploymentu używającego gateway injection. Image jest tutaj ustawione jako auto, czyli będzie samemu tworzony przez mutating webhook:

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: ingressgateway
...
template:
  metadata:
    labels:
      sidecar.istio.io/inject: "true" # Triggers the Mutating Webhook
      image: auto # Image populated by the Mutating Webhook
      imagePullPolicy: Always
    name: istio-proxy

```

Ćwiczenie na VS i DR

Konfigurowanie dostępu z zewnątrz do aplikacji przy pomocy VirtualService i DestinationRule

W tym zadaniu będziemy manipulować ruchem przy pomocy VirtualService i DestinationRule. Jak wiemy Istio używa sidecarów, który działa przez stworzenie dodatkowego kontenera w naszym podzie, który przejmuje kontrolę nad całym ruchem przychodzącym do poda. Jednym ze sposobów na włączenie Istio jest dodanie "istio-injection=enabled" labeli do naszego namespace i wtedy każdy nowy pod stworzony w tym namespace będzie miał automatycznie uruchamiane Istio i Envoy proxy.

```
kubectl label namespace default istio-injection=enabled --overwrite
```

Aplikacja jest tak zbudowana, że na początku nic z zewnątrz nie ma do niej dostępu (jest service typu ClusterIP). Aby się upewnić, czy połaczenie z zewnątrz nie działa, możemy użyć komendy:

```
kubectl port-forward $(kubectl get pod -l app=productpage -o jsonpath='{.items[0].metadata.name}') 9080:9080
```

Używając istioctl możemy zobaczyć status konfiguracji, którą Istiod synchronizuje z każdym sidecarem, a także do której instancji Istiod (może być kilka) jest podłączony.

```
$ISTIO_INSTALL_DIR/bin/istioctl proxy-status
```

Aby wystawić naszą aplikację na zewnątrz, tworzymy Istio Gateway przy pomocy pliku yaml:

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: bookinfo-gateway
spec:
  selector:
    istio: ingressgateway - najlepiej użyć domyślnych istio controllerservers
  - port:
      number: 80
      name: http
      protocol: HTTP
  hosts:
    - "*"
```

Dodamy też kilka przydatnych zmiennych środowiskowych:

```
export INGRESS_HOST=$(kubectl -n istio-system get service istio-ingressgateway -o
jsonpath='{.status.loadBalancer.ingress[0].ip}')
export INGRESS_PORT=$(kubectl -n istio-system get service istio-ingressgateway -o
jsonpath='{.spec.ports[?(@.name=="http2")].port}')
export GATEWAY_URL=$INGRESS_HOST:$INGRESS_PORT
```

Należy pamiętać, że przy pomocy powyższego pliku yaml nie tworzymy nowego obiektu gatewaya, ale dodajemy nową konfigurację do istniejącego już gatewaya, w naszym przypadku jest to pod *istio-ingressgateway*. Linkiem pomiędzy tą konfiguracją, a prawdziwym gatewayem jest labela *istio: ingressgateway*, którą ma też pod. W tym przypadku pod gatewayowy ma też service typu LoadBalancer przed sobą, który będzie punktem wejściowym do naszej aplikacji. Service przy pomocy labelki będzie przekierowywał ruch do *ingressgateway* poda, jednakże w tym momencie ten pod nie ma pojęcia dokąd dalej przekazać ruch dlatego potrzebujemy stworzyć VirtualService:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
```

```

metadata:
  name: bookinfo
spec:
  hosts:
  - "*"
  gateways:
  - bookinfo-gateway
  http:
  - match:
    - uri:
        exact: /productpage
    - uri:
        prefix: /static
    - uri:
        exact: /login
    - uri:
        exact: /logout
        prefix: /api/v1/products
  route:
  - destination:
    host: productpage
    port:
      number: 9080

```

Przy pomocy tego obiektu instruujemy Istio, że każdy request przychodzący z gatewya nazwanego *bookinfo-gateway* powinien iść do service o nazwie *productpage*, przy założeniu, że URL requesta zgadza się z przynajmniej jednym wymaganiem określonym w spec->http->match->uri. W tym momencie ruch z zewnątrz będzie dopuszczony do naszej aplikacji, ale nie będzie jeszcze dostępny żaden routing. W naszej aplikacji mamy 3 pody z różnymi jej wersjami, ale każda ma te same labelle, żeby być podpięta pod ten sam service. Kubernetes nie daje nam kontroli nad rozdzielaniem ruchu do nich. Musimy skonfigurować różne wersje aplikacji jako subsets w DestinationRule.

Za service o nazwie *reviews* mamy 3 pody o wersjach v1, v2 i v3. Gdybyśmy teraz stworzyli następujący VirtualService, który będzie próbował przesyłać ruch tylko do v1 to będzie błąd:

```

apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
  - reviews
  http:
  - route:
    - destination:
      host: reviews
      subset: v1

```

Wiedząc, że w naszej aplikacji *productpage* service łączy się z service *reviews* sprawdzamy logi *productpage* podów:

```
kubectl logs $(kubectl get pods -l=app=productpage -o name) -c istio-proxy
```

Będzie tu błąd o treści *cluster_not_found*. Dzieje się tak ponieważ VirtualService nie wie jeszcze co to jest subset: v1 i aby to skonfigurować potrzebujemy to ustawić w DestinationRule:

```
apiVersion: networking.istio.io/v1alpha3
```

```
kind: DestinationRule
metadata:
  name: reviews
spec:
  host: reviews
  subsets:
    - name: v1
      labels:
        version: v1
```

Teraz patrząc na te same logi zobaczymy błąd: `no_healthy_upstream`. Podsumowując do tej pory stworzyliśmy nowy cluster z DestinationRule

powyżej outbound|9080|v1|reviews.default.svc.cluster.local, ale Istio mówi, że nie ma w nim żadnych endpointów, do których może się połączyć. Dzieje się tak dlatego, że definicja subsetu bazyuje na labelach przypisanych do podów, więc jeśli w DestinationRule ustawiliśmy subset v1 to pod musi mieć labelę: `version: v1`. Po dodaniu tej labeli ruch do pierwszej wersji aplikacji będzie już działał. Pamiętając jednak, że mamy 3 wersje aplikacji, dodając odpowiednie labele chcemy, aby ruch był rozdzielany do wszystkich wersji, więc trzeba stworzyć następujący DestinationRule:

```
apiVersion: networking.istio.io/v1alpha3
```

```
kind: DestinationRule
metadata:
  name: reviews
spec:
  host: reviews
  subsets:
    - name: v1
      labels:
        version: v1
    - name: v2
      labels:
        version: v2
    - name: v3
      labels:
        version: v3
```

Mając te ustawienia możemy rozdzielić ruch np. procentowo pomiędzy pierwszą i drugą wersję przy pomocy DestinationRule:

```
apiVersion: networking.istio.io/v1alpha3
```

```
kind: VirtualService
```

```
metadata:
  name: reviews
spec:
  hosts:
    - reviews
  http:
    - route:
        - destination:
            host: reviews
            subset: v1
            weight: 50
        - destination:
            host: reviews
            subset: v2
```

weight: 50

Innym sposobem na routing ruchu jest np. routing na podstawie headerów. Można go np. użyć, gdy chcielibyśmy, aby tylko wybrani użytkownicy mieli dostęp do danej wersji aplikacji. Takie routowanie jest możliwe dzięki temu, że productpage service dodaje custom end-user header do wszystkich wychodzących http requestów do reviews service, a my mówimy Istio, gdzie przekierować ten request. Poniższy VirtualService umożliwia nam ten routing:

```
apiVersion: networking.istio.io/v1alpha3
```

```
kind: VirtualService
```

```
metadata:
```

```
  name: reviews
```

```
spec:
```

```
  hosts:
```

```
    - reviews
```

```
  http:
```

```
    - match:
```

```
      - headers:
```

```
        end-user:
```

```
          exact: Jason
```

```
  route:
```

```
    - destination:
```

```
      host: reviews
```

```
      subset: v3
```

```
    - route:
```

```
      - destination:
```

```
        host: reviews
```

```
        subset: v2
```

Ćwiczenie na ServiceEntry i Gateway

Ćwiczenie na ServiceEntry i Gateway

Najprawdopodobniej część aplikacji będzie hostowana na zewnątrz Kuberntesa i żeby się do niej łączyć należy dodać ServiceEntry, które wpisze danego hosta do wewnętrznego rejestru Istio i zezwoli na połaczenie.

W tym przykładzie deployujemy aplikację i najpierw ustawiamy labele na namespace, tak by Istio było uruchomione w naszych podach.

```
kubectl label namespace default istio-injection=enabled --overwrite
```

```
kubectl apply -f ${ISTIO_INSTALL_DIR}/samples/sleep/sleep.yaml
```

Ustawiamy też zmienną środowiskową z nazwą naszego poda:

```
export SOURCE_POD=$(kubectl get pod -l app=sleep -o jsonpath={.items..metadata.name})
```

Teraz tworzymy rzeczywisty gateway (IstioOperator) i ma on opcję *meshConfig.outboundTrafficPolicy.mode*, która mówi, jak Envoy ma traktować nieznane serwisy (czyli takie, których nie ma w wewnętrznym rejestrze Istio). Pierwsza opcja to ALLOW_ANY (ezwala na wszystkie połączenia), a druga to REGISTRY_ONLY (wszystkie nieznane są blokowane).

Aby sprawdzić obecne ustawienie możemy użyć komendy:

```
kubectl get configmap istio -n istio-system -o yaml | grep mode -m1 -B1
```

Jeśli rezultatem tej komendy jest pusty wiesz, to znaczy, że jest ALLOW_ANY, bo jest domyślny i nie trzeba go specjalnie precyzować. Tworzenie gatewya robimy komendą:

```
${ISTIO_INSTALL_DIR}/bin/istioctl manifest install -f <(cat <<EOF
```

```
apiVersion: install.istio.io/v1alpha1
```

```
kind: IstioOperator
```

```
spec:
```

```
  profile: default
```

```
EOF
```

```
)
```

Gdybyśmy teraz próbowały z naszego poda połączyć się gdzieś na zewnątrz to dostaniemy 200, bo jest ALLOW_ANY:

```
kubectl exec-it $SOURCE_POD-c sleep -- curl -I https://www.google.com
```

Należy jednak pamiętać, że jak zezwolimy w ten sposób na dostęp do zewnętrznych serwisów, to możliwości Istio nie będą zastosowane w tych połączeniach. Teraz będzie pokazane, jak skonfigurować dostęp do zewnętrznych serwisów http i https - httpbin.org, www.google.com bez tracenia funkcjonalności Istio.

Ustawiamy więc REGISTRY_ONLY na naszym gatewayu, oraz włączamy logi.

```
${ISTIO_INSTALL_DIR}/bin/istioctl manifest install -f <(cat <<EOF
```

```
apiVersion: install.istio.io/v1alpha1
```

```
kind: IstioOperator
```

```
spec:
```

```
  profile: default
```

```
  values:
```

```
    meshConfig:
```

```
      accessLogFile: /dev/stdout
```

```
      outboundTrafficPolicy:
```

```
        mode: REGISTRY_ONLY
```

```
EOF
```

```
)
```

Gdybyśmy teraz próbowali zrobić tego samego curla co wyżej to dostaniemy błąd i będzie on w logach naszego poda w kontenerze istio-proxy:

```
kubectl logs $SOURCE_POD-c istio-proxy -f
```

Teraz trzeba dodać dwa obiekty ServiceEntry, aby zezwolić na dostęp do naszych dwóch zewnętrznych domen:

```
kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: httpbin-ext
spec:
  hosts:
    - httpbin.org
  ports:
    - number: 80
      name: http
      protocol: HTTP
  resolution: DNS
  location: MESH_EXTERNAL
EOF
```

```
kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: google
spec:
  hosts:
    - www.google.com
  ports:
    - number: 443
      name: https
      protocol: HTTPS
  resolution: DNS
  location: MESH_EXTERNAL
EOF
```

Teraz robiąc curla połączenie już będzie udane, a w headerze http requesta zobaczymy informacje dodane przez Istio sidecar proxy.

W tym momencie wszystkie zewnętrzne requesty wychodzą bezpośrednio z podów, ale można je zcentralizować, tak by mieć nad nimi większą kontrolę. Trzeba do tego dodatkowo skonfigurować nasz egress gateway:

```
 ${ISTIO_INSTALL_DIR}/bin/istioctl manifest install -f <(cat <<EOF
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  profile: default
  components:
    egressGateways:
      - enabled: true
        name: istio-egressgateway
EOF
```

```
values:  
  meshConfig:  
    accessLogFile: /dev/stdout  
    outboundTrafficPolicy:  
      mode: REGISTRY_ONLY  
EOF  
)
```

Poniższą komendą potwierdzimy, że gateway istnieje:

```
kubectl get deploy -n istio-system -l app=istio-egressgateway
```

Teraz dodamy kolejne zewnętrzne połączenie:

```
kubectl apply -f - <<EOF  
apiVersion: networking.istio.io/v1alpha3  
kind: ServiceEntry  
metadata:  
  name: cnn  
spec:  
  hosts:  
  - edition.cnn.com  
  ports:  
  - number: 443  
    name: tls  
    protocol: TLS  
  resolution: DNS  
EOF
```

Teraz dodatkowo musimy zmodyfikować konfigurację istniejącego już rzeczywistego gatewya przez stworzenie obiektu Gateway pointującego przez label selector do *istio: egressgateway*.

```
kubectl apply -f - <<EOF  
apiVersion: networking.istio.io/v1alpha3  
kind: Gateway  
metadata:  
  name: istio-egressgateway  
spec:  
  selector:  
    istio: egressgateway  
  servers:  
  - port:  
    number: 443  
    name: tls  
    protocol: TLS  
  hosts:  
  - edition.cnn.com  
  tls:  
    mode: PASSTHROUGH # tls terminated on edition.cnn.com  
EOF
```

Teraz tworzymy VirtualService i DestinationRule, aby przekazać ruch przez egress gateway i z egress gatewya do external service.

```
kubectl apply -f - <<EOF
```

```

apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: direct-cnn-through-egress-gateway
spec:
  hosts:
    - edition.cnn.com
  gateways:
    - mesh # applies to all the sidecars in the mesh- istio-egressgateway
  tls:
    - match:
        - gateways:
            - mesh
          port: 443
        sniHosts:
          - edition.cnn.com
      route:      # traffic from mesh routed via egress gateway- destination:
        host: istio-egressgateway.istio-system.svc.cluster.local
        subset: cnn
        port:
          number: 443
    - match:
        - gateways:
            - istio-egressgateway
          port: 443
        sniHosts:
          - edition.cnn.com
      route:      # traffic from egress gateway routed to edition.cnn.com- destination:
        host: edition.cnn.com
        port:
          number: 443
        weight: 100
EOF

```

Powyższy VirtualService instruuje Istio do wysyłania requestów idących do *edition.cnn.com* przez egress gateway stworzony tu. Widzimy, że ruch musi iść do subsetu *cnn*, który stworzyliśmy tutaj. Teraz musimy tylko stworzyć DestinationRule, aby sprecyzować subset:

```

kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: egressgateway-for-cnn
spec:
  host: istio-egressgateway.istio-system.svc.cluster.local
  subsets:
    - name: cnn
EOF

```

Teraz, gdy wyślemy curla do endpointa, to znajdziemy nasz request w logach gatewayowego poda *istio-egressgateway*.

```
kubectl exec-it $SOURCE_POD -c sleep -- curl -sL -o /dev/null -D - https://edition.cnn.com/politics
```

Dodatkową opcją jest stworzenie network policy, która będzie blokowała omijanie egress gatewya. W tym celu tworzymy dodatkowy namespace i deployujemy tam aplikacje, która wysyła request do external service.

```
kubectl create namespace test-egress
```

```
kubectl apply -n test-egress -f ${ISTIO_INSTALL_DIR}/samples/sleep/sleep.yaml
```

Ten stworzony pod ma tylko pojedynczy container, bo w nowym namespace nie określiliśmy tworzenia Istio:

```
kubectl get pod $(kubectl get pod -n test-egress -l app=sleep -o jsonpath={.items..metadata.name}) -n test-egress
```

Teraz, jak wyślemy request do, to będzie on udany, bo nie mamy tutaj żadnych blokujących policy <https://edition.cnn.com/politics>

Dodamy teraz trochę labeli, aby móc je użyć w Network Policy. Dodajemy labele do namespace, w którym mamy Istio controlplane i gatewaye:

```
kubectl label namespace istio-system istio=system --overwrite
```

Potem kube-system.

```
kubectl label ns kube-system kube-system=true --overwrite
```

Definiujemy NetworkPolicy, do limitowania ruchu wyjściowego z test-egress namespace i zezwalamy tylko na ruch do istio-system i kube-system DNS service (port 53):

```
cat <<EOF | kubectl apply -n test-egress -f -
```

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

```
kind: NetworkPolicy
```

```
metadata:
```

```
  name: allow-egress-to-istio-system-and-kube-dns
```

```
spec:
```

```
  podSelector: {}
```

```
  policyTypes:
```

```
    - Egress
```

```
  egress:
```

```
    - to:
```

```
      - namespaceSelector:
```

```
        matchLabels:
```

```
          kube-system: "true"
```

```
      ports:
```

```
        - protocol: UDP
```

```
          port: 53
```

```
    - to:
```

```
      - namespaceSelector:
```

```
        matchLabels:
```

```
          istio: system
```

```
EOF
```

Teraz, jak wyślemy curla do endpointu, to nie zadziała, bo ruch jest blokowany przez Network Policy. Nasz pod nie może ominąć istio-egressgatewaya. Jedyny sposób na dostanie się do edition.cnn.com to poprzez Istio sidecar proxy i przez wysłanie ruchu do istio-egressgateway. To ustawienie pokazuje, że nawet jak będzie złośliwy pod i będzie chciał ominąć proxy to nie da rady.

Teraz trzeba uruchomić Istio w naszym nowym namespace. Najpierw dodajemy labele i potem usuwamy poda, aby stworzył się na nowo.:

```
kubectl label namespace test-egress istio-injection=enabled
```

Po rekreacji będziemy już mieć drugi kontener istio-proxy. Teraz trzeba stworzyć taki sam DestinationRule, jak w poprzednim przypadku, aby kierować ruch do istio-egressgateway i przypisać tą Rule do naszego poda labelką:

```
kubectl apply -n test-egress -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: egressgateway-for-cnn
spec:
  host: istio-egressgateway.istio-system.svc.cluster.local
  subsets:
  - name: cnn
EOF
```

Tu można zobaczyć, dlaczego nie trzeba tu też tworzyć VirtualService, takiego, jaki jest w default namespace: [Istio traffic management best practices documentation](#)

Teraz nasze połączenie już zadziała, a wpis o tym zobaczymy w logach istio-egressgateway poda.

Ćwiczenie z Circuit Breaking

Ćwiczenie z Circuit Breaking

W Circuit Breaking ustawiamy limity, które pozwalają nam nie przeciągać przeładowanego już endpointa, w praktyce chodzi o to, że jak już endpoint ma maksymalną ilość requestów w kolejce to następne będą od razu failowane, zamiast dodawać je do kolejki. Gdy połączymy to z pasywnymi sprawdzianiami stanu poda (outlier detection), to można stworzyć bezpieczne środowisko.

Tworzymy aplikację:

```
kubectl apply -f ${ISTIO_INSTALL_DIR}/samples/httpbin/httpbin.yaml
```

Tworzymy teraz DestinationRule, w której aplikujemy Circuit Breaking w przypadku połączeń do `httpbin` service. Jeśli mamy skonfigurowany mutual TLS authentication to trzeba dodać TLS traffic policy mode: `ISTIO_MUTUAL`, bo inaczej będą błędy:

```
kubectl apply -f - <<EOF
```

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: httpbin
spec:
  host: httpbin
  trafficPolicy:
    connectionPool:
      tcp:
        maxConnections: 1
      http:
        http1MaxPendingRequests: 1
        maxRequestsPerConnection: 1
  outlierDetection:
    consecutiveErrors: 1
    interval: 1s
    baseEjectionTime: 3m
    maxEjectionPercent: 100
EOF
```

Sekcja `outlierDetection` ma za zadanie śledzić status endpointa i zadecydować, czy jest zdrowy, czy nie. W tym przypadku dowolny host, gdy będzie miał przynajmniej 1 błąd w ciągu 1 sekundy będzie uznany za `unhealthy` na przynajmniej 3 minuty.

Jeśli zrobimy teraz curla z poda do endpointu, to wszystko zadziała, bo nie będzie on przeładowany, jeśli natomiast używalibyśmy np. pętli z dużą ilością requestów to będą one odrzucone przez Circuit Breaking.

Ćwiczenie na Peer Authentication

Ćwiczenie na Peer Authenticaiton

Najpierw deployujemy apki. Będą tu dwa namespacy *foo* i *bar* z dwoma service *httpbin* i *sleep*. W obu będzie uruchomione Istio. Będzie też druga instancja naszych aplikacji bez Istio w *legacy* namespace.

kubectl create ns foo

```
kubectl apply -f <${ISTIO_INSTALL_DIR}/bin/istioctl kube-inject -f ${ISTIO_INSTALL_DIR}/samples/httpbin/httpbin.yaml> -n foookubectl apply -f <${ISTIO_INSTALL_DIR}/bin/istioctl kube-inject -f ${ISTIO_INSTALL_DIR}/samples/sleep/sleep.yaml> -n foo
```

kubectl create ns bar

```
kubectl apply -f <${ISTIO_INSTALL_DIR}/bin/istioctl kube-inject -f ${ISTIO_INSTALL_DIR}/samples/httpbin/httpbin.yaml> -n barkubectl apply -f <${ISTIO_INSTALL_DIR}/bin/istioctl kube-inject -f ${ISTIO_INSTALL_DIR}/samples/sleep/sleep.yaml> -n bar
```

kubectl create ns legacy

```
kubectl apply -f ${ISTIO_INSTALL_DIR}/samples/httpbin/httpbin.yaml -n legacy
```

```
kubectl apply -f ${ISTIO_INSTALL_DIR}/samples/sleep/sleep.yaml -n legacy
```

Sprawdzenie, czy są jakieś Peer Authentication w naszym clustrze:

kubectl get peerauthentication --all-namespaces

Wiemy, że domyślnie mTLS jest uruchomiony, ale w trybie zezwalającym też na połączenia niezaszyfrowany. Aby to zmienić i wymusić używanie mTLS w całym meshu, ustawimy mesh-wide policy z Peer Authentication z trybem STRICT. Jeśli chcemy ustawić policy na poziomie całego mesha, to nie ustawiamy w pliku yaml selectora, bo musi to być przypisane do *istio-system* namespace:

kubectl apply -f - <<EOF

```
apiVersion: "security.istio.io/v1beta1"
```

```
kind: "PeerAuthentication"
```

```
metadata:
```

```
  name: "default" namespace: "istio-system"
```

```
spec:
```

```
  mtls:
```

```
    mode: STRICT
```

```
EOF
```

Teraz, jak będziemy się łączyć pomiędzy aplikacjami curlem, to zobaczymy, że połączenia z *legacy* namespace nie zadziałają, bo nie ma tam Istio - nie ma mTLS.

Jeśli chcemy usunąć poprzednią Peer Authentication policy używamy:

kubectl delete peerauthentication -n istio-system default

Teraz dodamy taką samą policy, ale na poziomie namespace, więc ustawiamy konkretny namespace w pliku yaml:

kubectl apply -f - <<EOF

```
apiVersion: "security.istio.io/v1beta1"
```

```
kind: "PeerAuthentication"
```

```
metadata:
```

```
  name: "default"
```

```
  namespace: "foo"
```

```
spec:  
  mtls:  
    mode: STRICT  
EOF
```

Teraz, jak użyjemy pętli do sprawdzania połączeń, to nie będą działać tylko połączenia z legacy namespace do foo.

```
forfrom in"foo""bar""legacy"; do  
  forto in"foo""bar""legacy"; do  
    kubectl exec$(kubectl get pod -l app=sleep -n ${from}-o jsonpath={.items..metadata.name}) -c sleep -n ${from}-- curl "http://httpbin.${to}:8000/ip"-s -o /dev/null -w "sleep.${from}to httpbin.${to}: ${http_code}\n";  
  done;  
done
```

Done

Jeśli chcemy ustawić Peer Authentication tylko dla danego service, to w pliku yaml musimy dodać label selector, aby wybrać ten konkretny Service, który nas interesuje. Należy jednak pamiętać, że Istio nie umie wymuszać tych policy do wychodzącego ruchu z service - do tego trzeba skonfigurować DestinationRule. Poniżej mamy przykładowe ustawienie mTLS dla pojedynczego service (*httpbin.bar*), oraz DestinationRule, która będzie wymagać mTLS na wyjściu:

```
cat <<EOF | kubectl apply -n bar -f-  
apiVersion: "security.istio.io/v1beta1"  
kind: "PeerAuthentication"  
metadata:  
  name: "httpbin"  
  namespace: "bar"  
spec:  
  selector:  
    matchLabels:  
      app: httpbin  
  mtls:  
    mode: STRICT  
EOF
```

```
cat <<EOF | kubectl apply -n bar -f-  
apiVersion: "networking.istio.io/v1alpha3"  
kind: "DestinationRule"  
metadata:  
  name: "httpbin"  
  namespace: "bar"  
spec:  
  host: "httpbin.bar.svc.cluster.local"  
  trafficPolicy:  
    tls:  
      mode: ISTIO_MUTUAL  
EOF
```

Teraz po sprawdzeniu, znowu requesty z legacy namespace do *httpbin.bar* nie będą działać. Aby wymusić mTLS na połączeniu po danym porcie, trzeba ustawić sekcję *portLevelMtls*. Np. tutaj wymagamy mTLS wszędzie oprócz portu 80:

```
cat <<EOF | kubectl apply -n bar -f-  
apiVersion: "security.istio.io/v1beta1"  
kind: "PeerAuthentication"
```

```
metadata:  
  name: "httpbin"  
  namespace: "bar"  
spec:  
  selector:  
    matchLabels:  
      app: httpbin  
  mtls:  
    mode: STRICT  
  portLevelMtls:  
    80:  
      mode: DISABLE  
EOF
```

Tak jak poprzednio będziemy też potrzebować DestinationRule do ruchu wychodzącego:

```
cat <<EOF | kubectl apply -n bar -f-  
apiVersion: "networking.istio.io/v1alpha3"  
kind: "DestinationRule"  
metadata:  
  name: "httpbin"  
spec:  
  host: httpbin.bar.svc.cluster.local  
  trafficPolicy:  
    tls:  
      mode: ISTIO_MUTUAL  
  portLevelSettings:  
    - port:  
        number: 8000  
        tls:  
          mode: DISABLE  
EOF
```

Ustawienie portu w Peer Authentication to port kontenera, a port w DestinationRule to port service. Można tylko użyć opcji *portLevelMtls* jeśli dany port jest podpięty pod ten service, inaczej sekcja ta będzie ignorowana.

Ćwiczenie na Request Authentication

Ćwiczenie na Request Authentication

Najpierw deployujemy aplikacje w dwóch namespace foo i bar i w każdej aplikacji jest Istio:

```
kubectl create ns foo  
kubectl apply -f <(${ISTIO_INSTALL_DIR}/bin/istioctl kube-inject -f  
${ISTIO_INSTALL_DIR}/samples/httpbin/httpbin.yaml) -n foo  
kubectl apply -f <(${ISTIO_INSTALL_DIR}/bin/istioctl kube-inject -f  
${ISTIO_INSTALL_DIR}/samples/sleep/sleep.yaml) -n foo  
kubectl create ns bar  
kubectl apply -f <(${ISTIO_INSTALL_DIR}/bin/istioctl kube-inject -f  
${ISTIO_INSTALL_DIR}/samples/httpbin/httpbin.yaml) -n bar  
kubectl apply -f <(${ISTIO_INSTALL_DIR}/bin/istioctl kube-inject -f  
${ISTIO_INSTALL_DIR}/samples/sleep/sleep.yaml) -n bar
```

Wystawiamy teraz aplikację `httpbin.foo` na zewnątrz przez ingressgateway:

```
kubectl apply -f - <<EOF  
apiVersion: networking.istio.io/v1alpha3  
kind: Gateway  
metadata:  
  name: httpbin-gateway  
  namespace: foo  
spec:  
  selector:  
    istio: ingressgateway # use Istio default gateway implementation  
  servers:  
  - port:  
      number: 80  
      name: http  
      protocol: HTTP  
      hosts:  
      - "*"---  
apiVersion: networking.istio.io/v1alpha3  
kind: VirtualService  
metadata:  
  name: httpbin  
  namespace: foo  
spec:  
  hosts:  
  - "*"gateways:  
  - httpbin-gateway  
  http:  
  - route:  
    - destination:  
      port:  
        number: 8000  
      host: httpbin.foo.svc.cluster.local  
EOF
```

Ustawiamy IP ingressgatewaya w zmiennej:

```
export INGRESS_HOST=$(kubectl -n istio-system get service istio-ingressgateway -o jsonpath='{.status.loadBalancer.ingress[0].ip}'")
```

Teraz robiąc curla do ingressgatewaya połączenie będzie udane:
`curl $INGRESS_HOST/headers -s -o /dev/null -w "%{http_code}\n"200`

Teraz będziemy ustawiać autentykację end-usera i potrzebujemy do tego JWT. Musi on odnosić się do JWKS endpointu, którego będziemy używać w zadaniu. Będą tu użyte: [JWT test](#) i [JWKS endpoint](#) z bazy kodów Istio.Teraz dodamy RequestAuthentication policy wymagającą tokena od end-usera do autentykacji w ingressgatewayu:

```
kubectl apply -f - <<EOF
apiVersion: "security.istio.io/v1beta1"
kind: "RequestAuthentication"
metadata:
  name: "jwt-example"
  namespace: istio-system
spec:
  selector:
    matchLabels:
      istio: ingressgateway
  jwtRules:
    - issuer: "testing@secure.istio.io"jwksUri: "https://raw.githubusercontent.com/istio/istio/release-1.12/security/tools/jwt/samples/jwks.json"EOF
```

Powyższa policy będzie nadana w istio-system anmespace. Jeśli podamy token, to Istio go będzie walidować przy pomocy publicznego klucza. W tym jednak momencie requesty bez klucza też zadziałały, nie zadziała tylko ja będzie zły token.

Poniżej testy:

No token:

```
curl $INGRESS_HOST/headers -s -o /dev/null -w "%{http_code}\n"200
```

Bad token:

```
curl --header "Authorization: Bearer deadbeef"$INGRESS_HOST/headers -s -o /dev/null -w "%{http_code}\n"401
```

Valid token:

```
TOKEN=$(curl https://raw.githubusercontent.com/istio/istio/release-1.12/security/tools/jwt/samples/demo.jwt -s)
```

```
curl --header "Authorization: Bearer $TOKEN"$INGRESS_HOST/headers -s -o /dev/null -w "%{http_code}\n"200
```

Teraz zmodyfikujemy policy, tak aby odrzucać wszystkie requesty bez tokena:

```
kubectl apply -f - <<EOF
apiVersion: "security.istio.io/v1beta1"
kind: "AuthorizationPolicy"
metadata:
  name: "ingressgateway"
  namespace: istio-system
spec:
  selector:
    matchLabels:
      istio: ingressgateway
  action: DENY
```

```
rules:  
- from:  
  - source:  
    notRequestPrincipals: ["*"]
```

EOF

Jest to uzyskane dzięki ostatniemu wpisowi w sekcji spec->rules.

Ćwiczenie – Stress

Zadanie 1

Mamy ważną aplikację i użytkownicy zaczynają się skarżyć, że są timeouty w połączeniach do niej. Sprawdziliśmy, że aplikacja nie ma żadnego issue.

Rozwiążanie:

Tutaj problemem był daemonSet, który pobierał całe CPU z worker nodów. Performance można sprawdzić przy pomocy komendy *k top nodes*. Mamy wiele rozwiązań tego problemu np. dodanie sekcji requests w naszym deploymencie:

```
...
spec:
  containers:
    - name: goldengoose
      image: dippynark/goldengoose::wargaming
      resources:
        requests:
          cpu: 200m
          memory: 200Mi
```

Można też byłoby dodać ResourceQuota w namespace, w którym był deamonSet i to ograniczyłoby jego działanie. Np. tutaj nie może być poda, który ma klasę BestEffort (zużywa najwięcej i przekracza limity) i będzie on usuwany:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: no-best-effort-pods
  namespace: developer
spec:
  hard:
    pods: "0"
  scopes:
    - BestEffort
```

Inna opcja to dodanie LimitRange w namespace i to ograniczy maksymalne zużycie jednostek przez pody w tym namespace:

```
apiVersion: v1
kind: LimitRange
metadata:
  name: cpu-limit-range
  namespace: developer
spec:
  limits:
  - max:
      cpu: 800m
    min:
      cpu: 200m
    type: Container
  - maxLimitRequestRatio:
      cpu: 2
    type: Pod
```

Inną możliwością jest zmniejszenie priorytetu podów, które pobierają za dużo zasobów. To sprawi, że będą one miały mniejszy priorytet przy schedulowaniu. Im mniejsza liczba, tym mniejszy priorytet. Aby to zrealizować trzeba stworzyć obiekt PriorityClass i dodać go w sekcji spec poda:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  priorityClassName: high-priority
```

```
apiVersion: scheduling.k8s.io/v1alpha1
kind: PriorityClass
metadata:
  name: high-priority
value: 1000000
globalDefault: false
```