- Additional Resources

- [The Istio documentation]
https://istio.io/latest/docs/

- [The Service Mesh Handbook (PDF)]
https://www.tetrate.io/service-mesh-handbook/

- [Istio Cheat Sheet]
https://tetr8.io/istio-cheatsheet

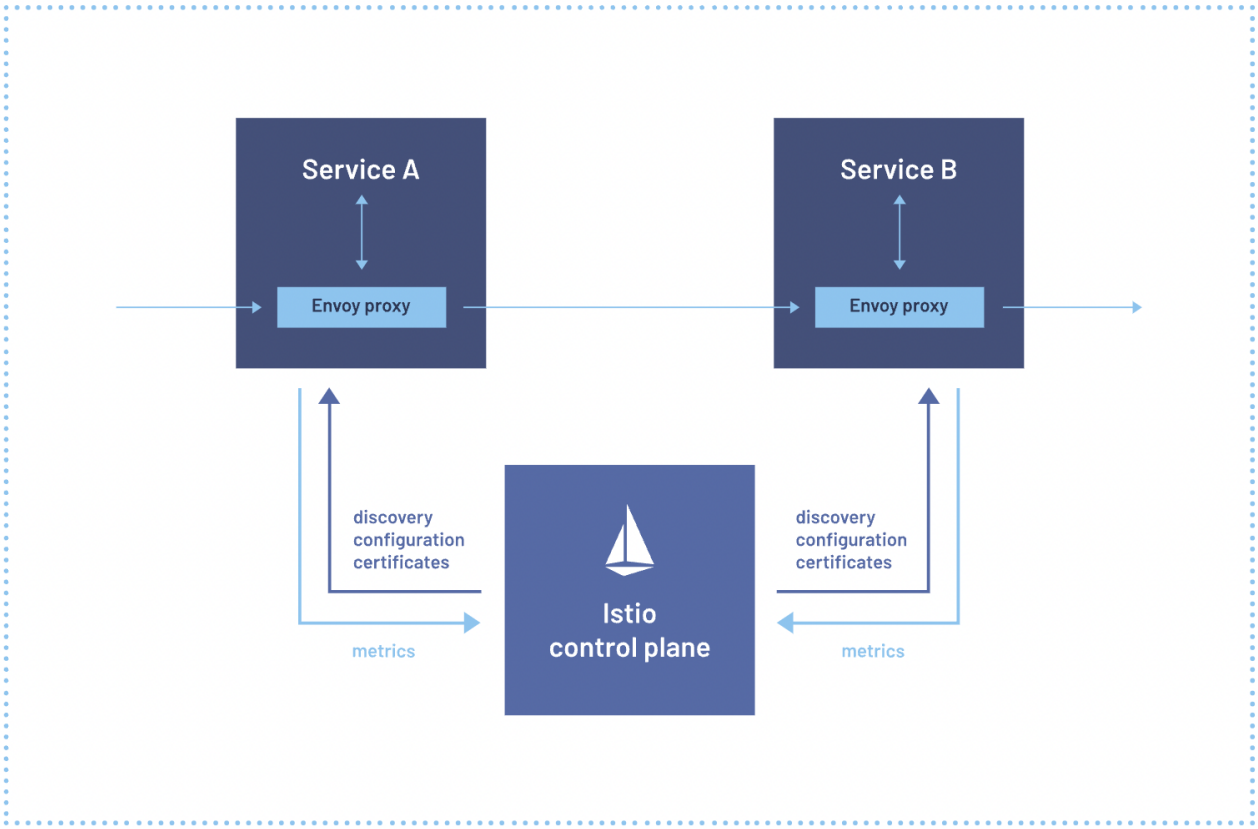
- [Collection of Istio articles and resources]
https://github.com/askmeegs/learn-istio

- [Istio weekly YouTube playlist]
https://www.youtube.com/playlist?list=PLm51GPKRAmTnMzTf9N95w_yXo7izg80Jc

- [Tetrate Tech Talks YouTube playlist]
https://www.youtube.com/playlist?list=PLm51GPKRAmTlOkjWDJBQYtjcc9WPk4E4F

- [Envoy fundamentals course]
https://academy.tetrate.io/courses/envoy-fundamentals

- Overview



Sidecar Injection

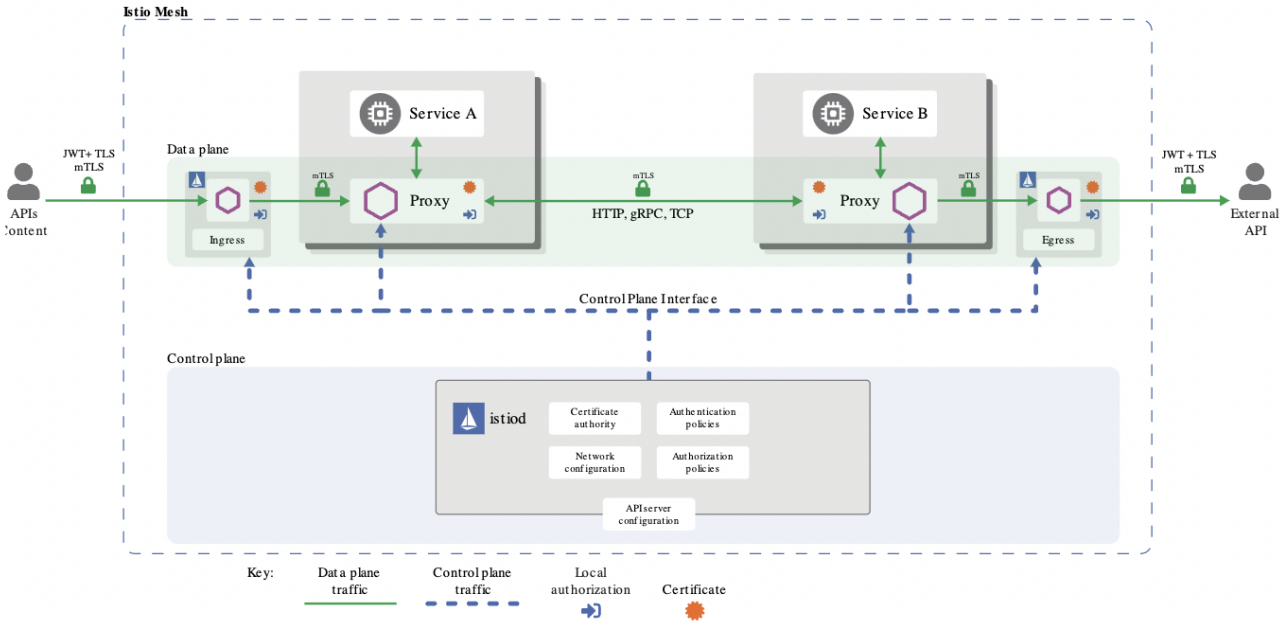
Manual sidecar injection

Istio has a command-line interface (CLI) named **`istioctl`** with the subcommand **`kube-inject`**. The subcommand processes the original deployment manifest to produce a modified manifest with the sidecar container specification added to the pod (or pod template) specification. The modified output can then be applied to a Kubernetes cluster with the **`kubectl apply -f`** command.
With manual injection, the process of altering the manifests is explicit.

Automatic sidecar injection

With automatic injection, the bundling of the sidecar is made transparent to the user.
The webhook is triggered according to a simple convention, where the application of the label **`istio-injection=enabled`** to a Kubernetes namespace governs whether the webhook should modify any deployment or pod resource applied to that namespace to include the sidecar.

In addition to the Envoy sidecar, the sidecar injection process injects a Kubernetes init container. This **`init-container`** is a process that applies these **`iptables`** rules before the Pod containers are started.



- DEMO K8s

https://istio.io/latest/docs/setup/getting-started/#download - **install Istio binary**

https://kubernetes.io/docs/tasks/tools/ - **Install kubectl**

Configuration profiles

Istio service mesh has numerous configuration settings that operators can update before installing Istio. To group the most common configuration settings into a higher-level abstraction, Istio uses the concept of configuration profiles.

The configuration profiles contain different configuration settings for the control plane as well as the data plane of Istio. The installation configuration profiles are expressed through the Istio Operator API and the IstioOperator resource.

Six configuration profiles are currently available, as shown in the list below. To get an up-to-date list of Istio configuration profiles, run the **`istioctl profile list`** command.

- **Default profile**
  The default profile is meant for production deployments and deployments of primary clusters in multi-cluster scenarios. It deploys the control plane and ingress gateway.
- **Demo profile**
  The demo profile is intended for demonstration deployments. It deploys the control plane and ingress and egress gateways and has a high level of tracing and access logging enabled.
- **Minimal profile**
  The minimal profile is equivalent to the default profile but without the ingress gateway. Itth deploys the control plane.
- **External profile**
  The external profile is used for configuring remote clusters in a multi-cluster scenario. It does not deploy any components.
- **Empty profile**
  The empty profile is used as a base for custom configuration. It does not deploy any components.
- **Preview profile**
  The preview profile contains experimental features. It deploys the control plane and ingress gateway.
-

To install Istio using the Istio CLI, we can use the **`--set`** flag and specify the profile like this:
istioctl install --set profile=demo

The Istio CLI offers convenience commands that allow us to get a full dump of Istio configuration profiles and the differences between the two profiles.

istioctl profile dump demo

***Additional info**
- The Istio Operator API and the IstioOperator resource allow us to install and configure Istio on a Kubernetes cluster.
- We can install install using helm

Installation

Install istio binary

curl -L https://istio.io/downloadIstio |ISTIO_VERSION=1.14.3 sh -
cd istio-1.14.3
export PATH=$PWD/bin:$PATH

#Deploy demo profile
istioctl install --set profile=demo

#Check status
kubectl get po -n istio-system

Enable sidecar injection

kubectl label namespace default istio-injection=enabled

#Check status
kubectl get namespace -L istio-injection

#Test
kubectl create deploy my-nginx —image=nginx

kubectl get po
kubectl describe po `k8s  get pod |grep nginx| awk '{print $1}'`
kubectl delete deployment my-nginx

#Deploy the BookInfo sample application that is bundled with the Istio distribution
kubectl apply -f samples/bookinfo/platform/kube/bookinfo.yaml

#Finally, deploy the bundled sleep sample service:
kubectl apply -f samples/sleep/sleep.yaml
kubectl get pod

PRODUCTPAGE_POD=$(kubectl get pod -l app=productpage
-ojsonpath='{.items[0].metadata.name}')

SLEEP_POD=$(kubectl get pod -l app=sleep -ojsonpath='{.items[0].metadata.name}')

kubectl exec $SLEEP_POD -it -- curl productpage:9080/productpage | head

kubectl get pod $PRODUCTPAGE_POD -ojsonpath='{.spec.containers[*].name}'

#Use the below **`istioctl dashboard`** command to open the Envoy dashboard web page:

istioctl dashboard envoy deploy/productpage-v1.default

Observability

Deploy metrics server

- **Using minikube**
  minikube addons enable metrics-server
  minikube dashboard

Prometheus for Istio

kubectl apply -f samples/addons/prometheus.yaml
Next
With Prometheus now running and collecting metrics, send another request to the **`productpage`** service:

SLEEP_POD=$(kubectl get pod -l app=sleep -ojsonpath='{.items[0].metadata.name}')

kubectl exec $SLEEP_POD -it -- curl productpage:9080/productpage | head

Run the following command, which Istio provides as a convenience to expose the Prometheus server's dashboard locally:

istioctl dashboard prometheus

Enter the metric name **`istio_requests_total`**

istio_requests_total{response_code=“200"}

istio_requests_total{source_app=“sleep",destination_app="productpage"}
rate(istio_requests_total{destination_app=“productpage"}[5m])

while true; do kubectl exec $SLEEP_POD -it -- curl productpage:9080/productpage; sleep 1; done

Grafana Dashboards for Istio

kubectl apply -f samples/addons/grafana.yaml

Launch the Grafana UI with the following command:

istioctl dashboard grafana

Inside that folder, you will find six dashboards:

- **Mesh**: provides a high-level overview of the health of services in the mesh.
- **Service**: for monitoring a specific service. The metrics shown here are aggregated from multiple workloads.
- **Workload**: this allows you to inspect the behavior of a single workload.
- **Control Plane**: designed to monitor the health of **`istiod`**, the Control plane itself. It helps determine whether the control plane is healthy and able to synchronize the Envoy sidecars to the state of the mesh.
- **Performance**: this allows you to monitor the resource consumption of **`istiod`** and the sidecars.
- **Wasm Extension**: For monitoring the health of custom Web Assembly extensions deployed to the mesh.

Capture the name of the **`sleep`** pod:

SLEEP_POD=$(kubectl get pod -l app=sleep -ojsonpath='{.items[0].metadata.name}')

Run the following simple script to make repeated calls to the **`productpage`** endpoint:

while true; do kubectl exec $SLEEP_POD -it -- curl productpage:9080/productpage; sleep 0.3; done

Begin by navigating to the **Istio Mesh Dashboard**. This dashboard is a perfect place to get our bearings and see what services are running, inspect their health, and view global stats such as the global request volume.

Visit the **Istio Service Dashboard**, select the service named **`productpage.default.svc.cluster.local`**, and expand the *General* panel. There you will find the typical *golden signals*, including request volume, success rate (or errors), and request duration. The other two panels, *Client Workloads* and *Service Workloads,* break down incoming requests to this service by source and by destination, respectively.

Distributed Tracing

Distributed tracing is an important component of observability that complements metrics dashboards.
The idea is to provide the capability to "see" the end-to-end request-response flow through a series of microservices and to draw important information from it.
From a view of a distributed trace, developers can discover potential latency issues in their applications.
Terms
The end-to-end request-response flow is known as a *trace*. Each component of a trace, such as a single call from one service to another, is called a *span*. Traces have unique IDs, and so do spans. All spans that are part of the same trace bear the same trace ID.
The IDs are propagated across the calls between services in HTTP headers whose names begin with **`x-b3`** and are known as *B3 trace headers*

Deploy Jaeger

kubectl apply -f samples/addons/jaeger.yaml

The resulting deployment can be seen in the **`istio-system`** namespace.
As in previous labs, store the name of the **`sleep`** pod in an environment variable:

SLEEP_POD=$(kubectl get pod -l app=sleep -ojsonpath='{.items[0].metadata.name}')

while true; do kubectl exec $SLEEP_POD -it -- curl productpage:9080/productpage; sleep 1; done

istioctl dashboard jaeger

From the search form on the left-hand side of the UI, select the service **`productpage.default`**, and click on the button **Find Traces** at the bottom of the form.

Kiali Console

Kiali is an open-source graphical console specifically designed for Istio and includes numerous features.
Through alerts and warnings, it can help validate that the service mesh configuration is correct and that it does not have any problems.
With Kiali, one can view Istio custom resources, services, workloads, or applications.

Begin by deploying Kiali to your Kubernetes cluster:

kubectl apply -f samples/addons/kiali.yaml

As in previous labs, store the name of the **`sleep`** pod in an environment variable:
SLEEP_POD=$(kubectl get pod -l app=sleep -ojsonpath='{.items[0].metadata.name}')

Next, run the following command to send requests to the **`productpage`** service at a 1-2 second interval:

while true; do kubectl exec $SLEEP_POD -it -- curl productpage:9080/productpage; sleep 1; done

Finally, launch the Kiali dashboard:
istioctl dashboard kiali

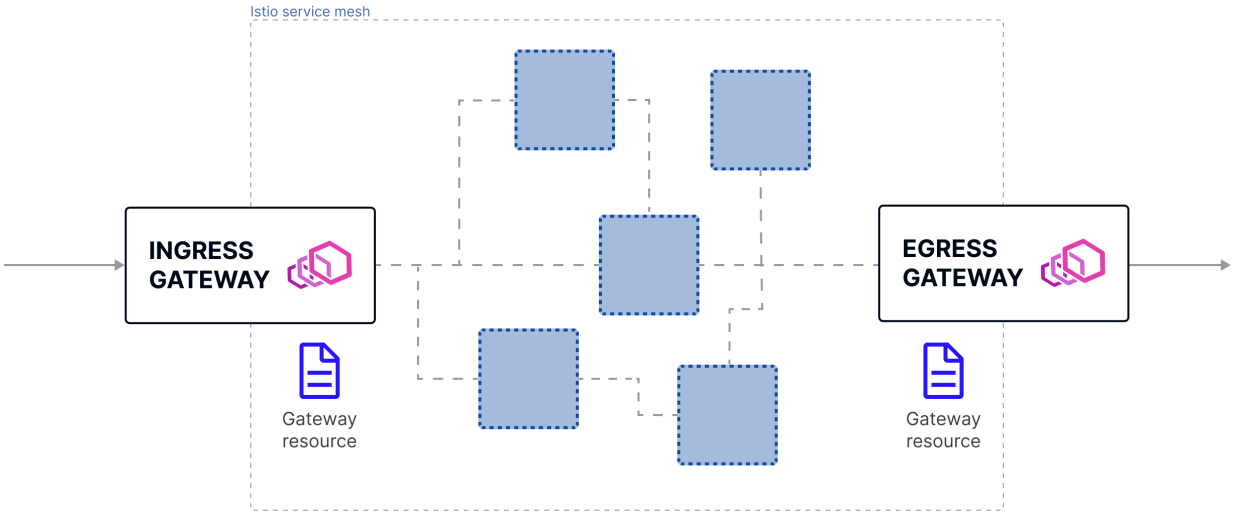
Traffic management

Gateways

Previously we installed Istio using the `demo` profile, it included the ingress and egress gateways.

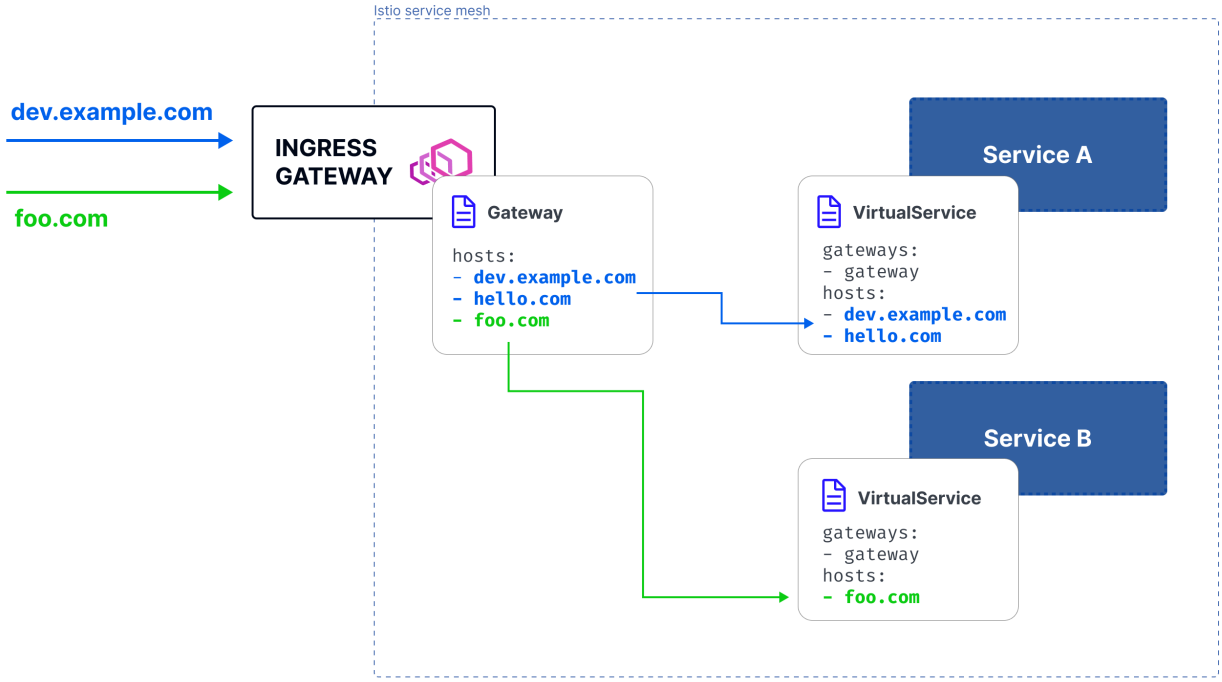
Both gateways are Kubernetes deployments that run an instance of the Envoy proxy, and they operate as load balancers at the edge of the mesh. The ingress gateway receives inbound connections, while the egress gateway receives connections going out of the cluster.

Using the ingress gateway, we can apply route rules to the inbound traffic entering the cluster. As part of the ingress gateway, a Kubernetes service of type LoadBalancer is deployed, giving us an external IP address.



Here's an example of a Gateway resource:

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: my-gateway
  namespace: default
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - dev.example.com
    - test.example.com
```



kubectl get svc -n istio-system

NOTE: How the LoadBalancer Kubernetes service type works depends on how and where we run the Kubernetes cluster. For a cloud-managed cluster (GCP, AWS, Azure, etc.), a load balancer resource gets provisioned in your cloud account, and the Kubernetes LoadBalancer service will get an external IP address assigned to it. Suppose we are using Minikube or Docker Desktop. In that case, the external IP address will either be set to **`localhost`** (Docker Desktop) or, if we are using Minikube, it will remain pending, and we will have to use the **`minikube tunnel`** command to get an IP address.

If we wanted to access the ingress gateway through a domain name, we could set the hosts' value to a domain name (e.g., **`example.com`**) and add the external IP address as an A record in the domain's DNS settings.

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

kubectl apply -f gateway.yaml

`minikube tunnel` (See **NOTE** below)

If we try to access the ingress gateway's external IP address, we will get back an HTTP 404 because there aren't any VirtualServices bound to the Gateway. The ingress proxy doesn't know where to route the traffic as we have not defined any routes yet.

To get the ingress gateway's external IP address, run the command below and look at the **`EXTERNAL-IP`** column value:

kubectl get svc -l=istio=ingressgateway -n istio-system

export GATEWAY_IP=$(kubectl get svc -n istio-system istio-ingressgateway -ojsonpath='{.status.loadBalancer.ingress[0].ip}')

#Create deployment and service “hello world”.

kubectl apply -f hello-world.yaml

kubectl get po,svc -l=app=hello-world

#Create VirtualService

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: hello-world
spec:
  hosts:
    - "*"
  gateways:
    - gateway
  http:
    - route:
        - destination:
            host: hello-world.default.svc.cluster.local
            port:
              number: 80
```

We use the **`*`** in the **`hosts`** field, just like in the Gateway resource. We have also added the Gateway resource we created earlier (**`gateway`**) to the **`gateways`** array. We say that we have attached the Gateway to the VirtualService.
Finally, we specify a single route with a destination that points to the Kubernetes service **`hello-world.default.svc.cluster.local`**.

kubectl apply -f vs-hello-world.yaml

kubectl get vs

curl -v http://$GATEWAY_IP/

Also, notice the **`server`** header set to **`istio-envoy`**, indicating that the request went through the sidecar proxy.

#Clean up.

kubectl delete deploy hello-world
kubectl delete service hello-world
kubectl delete vs hello-world
kubectl delete gateway gateway

Traffic Routing

We can use the VirtualService resource to configure routing rules for services within the Istio service mesh.

For example, in the VirtualService resource, we match the incoming traffic based on the request properties and then route the traffic to one or more destinations. For example, once we match the traffic, we can split it by weight, inject failures and/or delays, mirror the traffic, and etc.

The DestinationRule resource contains the rules applied after routing decisions (from the VirtualService) have already been made. With the DestinationRule, we can configure how to reach the target service. For example, we can configure outlier detection, load balancer settings, connection pool settings, and TLS settings for the destination service.

The last resource we should mention is the ServiceEntry. This resource allows us to take an external service or an API and make it appear as part of the mesh. The resource adds the external service to the internal service registry, allowing us to use Istio features such as traffic routing, failure injection, and others against external services.

!!!!!Before the Envoy proxy can decide where to route the requests, we need a way to describe what our system and services look like!!!!!!

To describe the different service versions, we use the concept of labels in Kubernetes. The pods created from the two customer deployments have the labels **`version: v1`** and **`version: v2`** set.

How does Istio know or distinguish between the different versions of the service?

We can set different labels in the pod spec template in each versioned deployment and then use these labels to make Istio aware of the two distinct versions or destinations for traffic. The labels are used in a construct called **subset** that can be defined in the DestinationRule.

To describe the two versions of the service, we would create a DestinationRule that looks like this:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: customers
spec:
  host: customers.default.svc.cluster.local
  subsets:
    - name: v1
      labels:
        version: v1
    - name: v2
      labels:
        version: v2
```

The Envoy clusters are named by concatenating the traffic direction, port, subset name, and service hostname.
For example:

outbound|80|v1|customers.default.svc.cluster.local
outbound|80|v2|customers.default.svc.cluster.local

We have multiple options when deciding on how we want the traffic to be routed:

- Route based on weights
- Match and route the traffic
- Redirect the traffic (HTTP 301)
- Mirror the traffic to another destination

The above options of routing the traffic can be applied and used individually or together within the same VirtualService resource.

Additionally, we can add, set or remove request and response headers, and configure CORS settings, timeouts, retries, and fault injection.

Examples:

1. Weight-based routing

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: customers-route
spec:
  hosts:
  - customers.default.svc.cluster.local
  http:
  - name: customers-v1-routes
    route:
    - destination:
        host: customers.default.svc.cluster.local
        subset: v1
      weight: 70
  - name: customers-v2-routes
    route:
    - destination:
        host: customers.default.svc.cluster.local
        subset: v2
      weight: 30
```

In this example, we split the traffic based on weight to two subsets of the same service, where 70% goes to subset **`v1`** and 30% to subset **`v2`**.

2. Match and route the traffic

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: customers-route
spec:
  hosts:
  - customers.default.svc.cluster.local
  http:
  - match:
    - headers:
        user-agent:
          regex: ".*Firefox.*"
    route:
```

```
    - destination:
        host: customers.default.svc.cluster.local
        subset: v1
  - route:
    - destination:
        host: customers.default.svc.cluster.local
        subset: v2
```

In this example, we provide a regular expression and try to match the **`User-Agent`** header value. If the header value matches, we route the traffic to subset **`v1`**. Otherwise, if the **`User-Agent`** header value doesn't match, we route the traffic to subset **`v2`**.

3. Redirect the traffic

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: customers-route
spec:
  hosts:
  - customers.default.svc.cluster.local
  http:
  - match:
    - uri:
        exact: /api/v1/helloWorld
    redirect:
      uri: /v1/hello
      authority: hello-world.default.svc.cluster.local
```

In this example, we combine the matching on the URI and then redirect the traffic to a different URI and a different service.

4. Traffic mirroring

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: customers-route
spec:
  hosts:
    - customers.default.svc.cluster.local
  http:
  - route:
    - destination:
        host: customers.default.svc.cluster.local
        subset: v1
      weight: 100
    mirror:
      host: customers.default.svc.cluster.local
      subset: v2
    mirrorPercentage:
      value: 100.0
```

In this example, we mirror 100% of the traffic to the **`v2`** subset. Mirroring takes the same request sent to subset **`v1`** and "mirrors" it to the **`v2`** subset. The request is "fire and forget". Mirroring can be used for testing and debugging the requests by mirroring the production traffic and sending it to the service version of our choice.

#Weight-Based Traffic Routing

#Deploy GW
kubectl apply -f gateway.yaml

#Deploy **`web-frontend`** and the **`customers`** service deployments and corresponding Kubernetes services

* Notice we are setting an environment variable called **`CUSTOMER_SERVICE_URL`** that points to the **`customers`** service we will deploy next. The **`web-frontend`** uses that URL to make a call to the **`customers`** service.

```
env:
        - name: CUSTOMER_SERVICE_URL
          value: 'http://customers.default.svc.cluster.local'
```

kubectl apply -f web-frontend.yaml

kubectl apply -f customers-v1.yaml

kubectl get po

#Create a VirtualService for the **`web-frontend`** and bind it to the Gateway resource:

kubectl apply -f web-frontend-vs.yaml

Next, we can set the environment variable with the **`GATEWAY_IP`** address like this:

export GATEWAY_IP=$(kubectl get svc -n istio-system istio-ingressgateway -ojsonpath='{.status.loadBalancer.ingress[0].ip}')

And open the `GATEWAY_IP` in the browser

If we deployed the **`customers`** service version **`v2`**, the responses we would get back when calling the **`http://customers.default.svc.cluster.local`** would be random. They would either come from the **`v2`** or **`v1`** version of the **`customers`** service. That is because the selector label in the Kubernetes service does not have the version label set.

#DestinationRule for the **`customers`** service and define the two subsets representing **`v1`** and **`v2`** versions.
kubectl apply -f customers-dr.yaml

#Create the VirtualService and specify the **`v1`** subset in the destination:
kubectl apply -f customers-vs.yaml

#Deploy the v2 version of the customers service.
kubectl apply -f customers-v2.yaml

#Create a second **`destination`** with the same hostname but a different subset. We will also add the **`weight: 50`** to both destinations to split the traffic between the versions equally.

kubectl apply -f customers-50-50.yaml

Open the **`GATEWAY_IP`** in the browser and refresh the page several times to see the different responses. The figure below shows the response from the **`customers-v2`**.

#Clean up

kubectl delete deploy web-frontend customers-{v1,v2}
kubectl delete svc customers web-frontend
kubectl delete vs customers web-frontend
kubectl delete dr customers
kubectl delete gateway gateway

Advanced Traffic Routing

#Deploy GW
kubectl apply -f gateway.yaml

#Deploy the web-frontend, customers-v1, customers-v2, and the corresponding VirtualServices and DestinationRule.

kubectl apply -f web-frontend.yaml

kubectl apply -f customers.yaml

set the environment variable **`GATEWAY_IP`** like this:

export GATEWAY_IP=$(kubectl get svc -n istio-system istio-ingressgateway -ojsonpath='{.status.loadBalancer.ingress[0].ip}')

#Update the **`customers`** VirtualService so that the traffic is routed between two versions of the **`customers`** service.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: customers
spec:
  hosts:
    - 'customers.default.svc.cluster.local'
  http:
  - match:
    - headers:
        user:
          exact: debug
    route:
    - destination:
        host: customers.default.svc.cluster.local
        port:
```

```
        number: 80
      subset: v2
  - route:
    - destination:
        host: customers.default.svc.cluster.local
        port:
          number: 80
        subset: v1
```

Save the above YAML to **`customers-vs-headers.yaml`** and update the VirtualService with the following command:

kubectl apply -f customers-vs-headers.yaml

NOTE: The destinations in the VirtualService would also work if we did not provide the port number. That's because the service has a single port defined.

If we open the **`GATEWAY_IP`**, we should still get the response from **`customers-v1`**. If we add the header **`user: debug`** to the request, you will notice that the response comes from **`customers-v2`**.

curl -H "user: debug" http://$GATEWAY_IP/ | grep -E 'CITY|NAME'

#Cleanup

kubectl delete deploy web-frontend customers-{v1,v2}
kubectl delete svc customers web-frontend
kubectl delete vs customers web-frontend
kubectl delete dr customers
kubectl delete gateway gateway

Observing Failure Injection and Delays

Service Resiliency & Failure Injection

Resiliency is the ability to provide and maintain an acceptable level of service in the face of faults and challenges to regular operation. It's not about avoiding failures. It's responding to them, so there's no downtime or data loss. The goal of resiliency is to return the service to a fully functioning state after a failure occurs.

Timeouts

A crucial element in making services available is using **timeouts** and **retry policies** when making service requests. We can configure both in the VirtualService resource.

Using the **`timeout`** field, we can define a timeout for HTTP requests. If the request takes longer than the value specified in the **`timeout`** field, the Envoy proxy will drop the request and mark it as timed out (return an **`HTTP 408`** to the application). The connections remain open unless outlier detection is triggered.

Here's an example of setting a timeout for a route:

```
...
- route:
  - destination:
      host: customers.default.svc.cluster.local
      subset: v1
  timeout: 10s
```

Circuit breaking.
It allows us to write services to limit the impact of failures, latency spikes, and other network issues.

Outlier detection is an implementation of a circuit breaker, and it's a form of passive health checking. It's called passive because Envoy isn't actively sending any requests to determine the health of the endpoints. Instead, Envoy observes the performance of different pods to determine if they are healthy or not. If the pods are deemed unhealthy, they are removed or ejected from the healthy load balancing pool.
Outlier detection in Istio is configured in the DestinationRule resource. Here's a snippet that configures outlier detection:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: customers
spec:
  host: customers
  trafficPolicy:
    connectionPool:
      tcp:
        maxConnections: 1
      http:
        http1MaxPendingRequests: 1
        maxRequestsPerConnection: 1
    outlierDetection:
      consecutive5xxErrors: 1
      interval: 1s
      baseEjectionTime: 3m
      maxEjectionPercent: 100
```

Fault injection.
We can apply the fault injection policies on HTTP traffic and specify one or more faults to inject when forwarding the request to the destination.
There are two types of fault injection. We can **delay** the requests before forwarding and emulate a slow network or overloaded service, and we can **abort** the HTTP request and return a specific HTTP error code to the caller. With the abort, we can simulate a faulty upstream service.

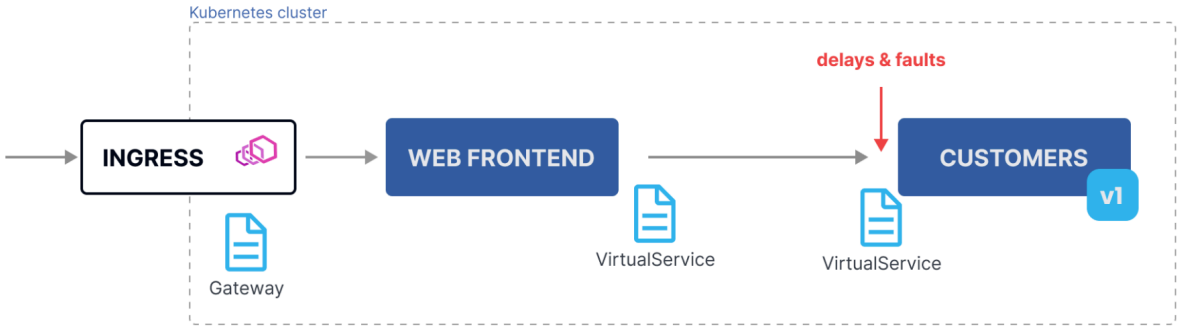
Here's an example that aborts HTTP requests and returns HTTP 404, for 30% of incoming requests:

```
- route:
    - destination:
        host: customers.default.svc.cluster.local
        subset: v1
    fault:
      abort:
        percentage:
          value: 30
        httpStatus: 404
```

Similarly, we can apply an optional delay to the requests using the **`fixedDelay`** field:

```
- route:
    - destination:
        host: customers.default.svc.cluster.local
        subset: v1
    fault:
      delay:
        percentage:
          value: 5
        fixedDelay: 3s
```

Observation in Grafana, Jaeger, Kiali



#Deploy GW
kubectl apply -f gateway.yaml

#Deploy web-frontend with corresponding Kubernetes Service, and VirtualService
kubectl apply -f web-frontend.yaml

#Deploy the `customers-v1` and related resources
kubectl apply -f customers.yaml

##########Delay##########

#Inject a 5-second delay to the `customers-v1` service for 50% of all requests.
kubectl apply -f customers-delay.yaml

#Grafana
export GATEWAY_IP=$(kubectl get svc -n istio-system istio-ingressgateway -ojsonpath='{.status.loadBalancer.ingress[0].ip}')

while true; do curl http://$GATEWAY_IP/; done

istioctl dash grafana

When Grafana opens, click **Home**, **istio** folder, and the **Istio Service Dashboards**.

On the dashboard, select the customers.default.svc.cluster.local in the Service dropdown and source in the Reporter dropdown.

#Jaeger
istioctl dash jaeger

On the main screen, select the `web-frontend.default` from the **Service** dropdown. Then, enter `5s` in the **Min Duration** text box. Click the **Find Traces** button to find the traces. Because we entered the minimum duration of traces, all traces in the list took at least 5 seconds. Click any of the traces to open the details. On the details page, we will notice the total duration is 5 seconds.
The single trace has four spans - expand the third span (one with the 5s duration) that represents the request made from the `web-frontend` to the `customers` service.

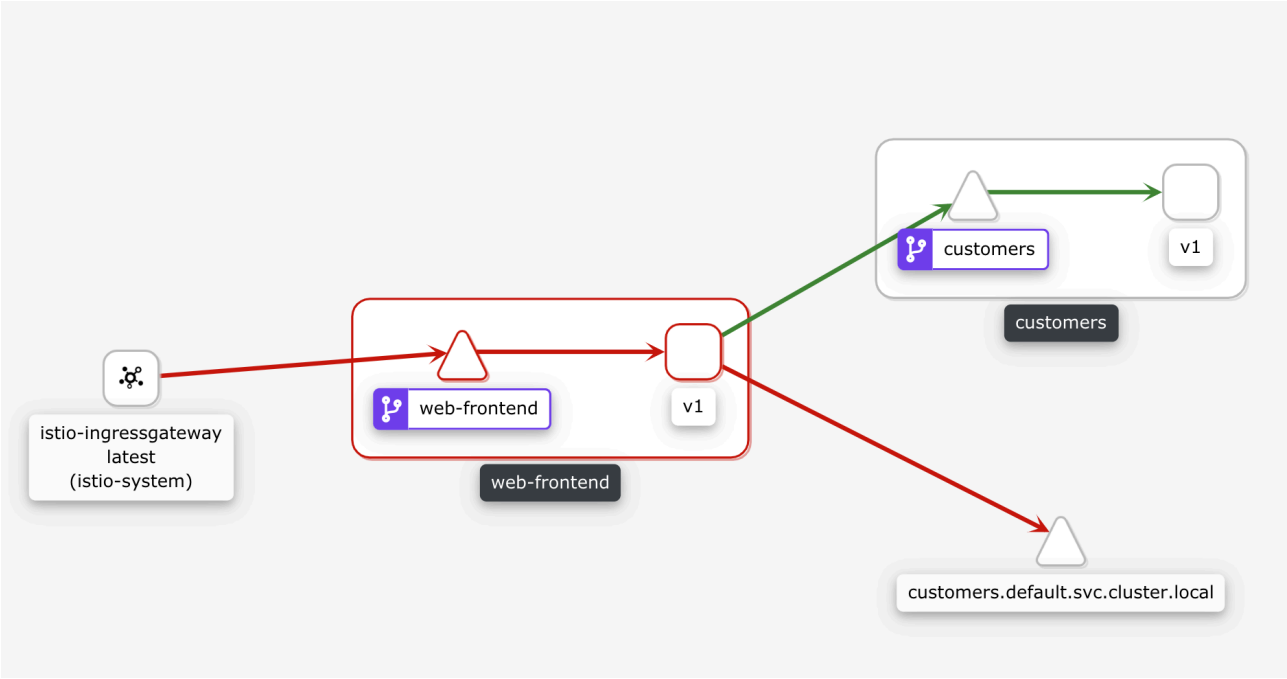
##########Fault##########
#update the VirtualService and inject a fault and return HTTP 500 for 50% of the requests.
kubectl apply -f customers-fault.yaml

#Grafana
Go back to Grafana and open the **Istio Service Dashboard** (make sure you select **`source`** from the **Reporter** dropdown), we will notice the client success rate dropping and the increase in the 500 responses on the **Incoming Requests by Source and Response Code** graph, as shown in the figure.

#Jaeger
Search for traces again (we can remove the min duration), we will notice the traces with errors.

#Kiali
istioctl dash kiali

Look at the service graph by clicking the **Graph** item. You will notice how the **`web-frontend`** service has a red border, as shown below.



- DEMO OCP