

What is Service Mesh and Istio

A **service mesh** is decentralized application networking infrastructure for making service-to-service communication safe, reliable, and understandable. A service mesh uses service proxies (that live with each service) to facilitate this functionality and each application talks through its own, co-located proxy. The proxies form the "mesh".

Istio is a service mesh which allows you to connect, secure, and control the traffic for your microservices in an declarative and non-intrusive way.

Some of the features that Istio enable for cloud-native applications:

- Intelligent routing and load balancing
- Resilience against network failures
- Policy enforcement between services
- Observability of your network communication
- Securing service to service communication

Istio Architecture

Istio follows the typical service-mesh architecture with the following abstractions:

- **Data plane** which is composed of **Envoy** service proxies deployed as a sidecar along with your service through which all application traffic flows
- **Control plane** which manage and configure the data plane (**Envoy** service proxies); additionally the control plane manages back-end infrastructure that complements the data plane (like metrics, policy, and security engines).

All communication within the **service mesh** happens through each application's **Envoy** proxy. Any network logic (retries, timeouts, circuit breaking, etc) is moved from your service into the service mesh.

Key Concepts of Istio

DestinationRule

A **DestinationRule** configures the set of rules to be applied when sending traffic to a service. Some of the purposes of a **DestinationRule** are describing circuit breakers, load balancer, and TLS settings or define **subsets** (named versions) of the destination hosts so they can be reused in other Istio elements.

For example, to define two versions of a service named **recommendation** you could do:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: recommendation
  namespace: tutorial
spec:
  host: recommendation
  subsets:
  - labels:
    version: v1
    name: version-v1
  - labels:
    version: v2
    name: version-v2
```

In this example, the subsets are defined by labels on the services (platform specific; for example, Kubernetes uses labels on its Deployment objects and the labels used in the subsets here would refer to those Deployment labels).

VirtualService

A **VirtualService** describes the mapping between one or more user-addressable destinations to the actual destination inside the service mesh.

For example, to define a single "virtual service" where the traffic is split between two deployed versions with 90% going to version 1 and 10% going to version 2:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: recommendation
  namespace: tutorial
spec:
  hosts:
  - recommendation
  http:
  - route:
    - destination:
        host: recommendation
        subset: version-v1
        weight: 90
    - destination:
        host: recommendation
        subset: version-v2
        weight: 10
```

ServiceEntry

A **ServiceEntry** is used to define services that exist outside of the service mesh but need to be made available to services within the mesh. A **ServiceEntry** is a way to add the details of a service into

Istio's service registry. You can write **VirtualService** and/or **DestinationRule** against a **ServiceEntry** just as if it was a mesh-native service.

For example to configure *httpbin* external service:

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: httpbin-egress-rule
  namespace: istioegress
spec:
  hosts:
  - httpbin.org
  ports:
  - name: http-80
    number: 80
    protocol: http
```

Gateway

A **Gateway** is used to describe a load balancer operating at the edge of the mesh for incoming/outgoing HTTP/TCP connections. You can use a **VirtualService** to define routing rules (using the full power of Istio's routing capabilities) for traffic originating at the edge or destined for external services.

To configures a **Gateway** to allow external HTTPS traffic for host foo.com into the mesh:

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: foo-gateway
spec:
  servers:
  - port:
    number: 443
    name: https
    protocol: HTTPS
  hosts:
  - foo.com
  tls:
    mode: SIMPLE
    serverCertificate: /tmp/tls.crt
    privateKey: /tmp/tls.key
```

Getting started with Istio

Istio can be installed with *automatic sidecar injection* or without it. We recommend as starting point **without** *automatic sidecar injection* so you understand each of the steps.

Installing Istio

First you need to download Istio and register in PATH:

```
open https://github.com/istio/istio/releases/

cd istio-1.0.3
export ISTIO_HOME=`pwd`
export PATH=$ISTIO_HOME/bin:$PATH
```

You can install Istio into Kubernetes cluster by either using helm install or helm template. With template we can create all of the resource files explicitly and then apply them like in this example:

```
$ helm template install/kubernetes/helm/istio \
  --name istio --namespace istio-system \
  --set sidecarInjectorWebhook.enabled=false \
  > $HOME/istio.yaml

kubectl create namespace istio-system
kubectl create -f $HOME/istio.yaml
```

Wait until all pods are up and running.

Intelligent Routing

Routing some percentage of traffic between two versions of recommendation service:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: recommendation
  namespace: tutorial
spec:
  hosts:
  - recommendation
  http:
  - route:
    - destination:
        host: recommendation
        subset: version-v1
      weight: 75
    - destination:
        host: recommendation
        subset: version-v2
      weight: 25
```

Routing to a specific version in case of prefixed URI and cookie with a value matching a regular expression:

```
spec:
  hosts:
  - ratings
  http:
  - match:
    - headers:
        cookie:
          regex: "^(*?;)?(user=jason)(;.*)?"
        uri:
          prefix: "/ratings/v2/"
    route:
    - destination:
        host: ratings
        subset: version-v2
```

Possible **match** options:

Field	Type	Description
uri	StringMatch	URI value to match. exact, prefix, regex
scheme	StringMatch	URI Scheme to match. exact, prefix, regex
method	StringMatch	Http Method to match. exact, prefix, regex
authority	StringMatch	Http Authority value to match. exact, prefix, regex
headers	map<string, StringMatch>	Headers key/value. exact, prefix, regex
port	int	Set port being addressed. If only one port exposed, not required
sourceLabels	map<string, string>	Caller labels to match
gateways	string[]	Names of the gateways where rule is applied to.

Sending traffic depending on caller labels:

```
- match:
  - sourceLabels:
      app: preference
      version: v2
    route:
  - destination:
      host: recommendation
      subset: version-v2
- route:
  - destination:
      host: recommendation
      subset: version-v1
```

When the calling service contains labels app=preference and version=v2, traffic is routed to **subset** version-v2. Otherwise, traffic is routed to version-v1

Mirroring traffic between two versions:

```
spec:
  hosts:
  - recommendation
  http:
  - route:
      - destination:
          host: recommendation
          subset: version-v1
    mirror:
      host: recommendation
      subset: version-v2
```

For routing purposes VirtualService also supports **redirects, rewrites, corsPolicies** or **appending** custom headers.

Apart from HTTP rules, VirtualService also supports matchers at *tcp* level.

```
spec:
  hosts:
  - postgresql
  tcp:
  - match:
      - port: 5432
        sourceSubnet: "172.17.0.0/16"
    route:
      - destination:
          host: postgresql
          port:
              number: 5555
```

Possible **match** options at *tcp* level:

Field	Type	Description
destinationSubnet	string	IPv4 or IPv6 of destination with optional subnet
port	int	Set port being addressed. If only one port exposed, not required
sourceSubnet	string	IPv4 or IPv6 of source with optional subnet
sourceLabels	map<string, string>	Caller labels to match
gateways	string[]	Names of the gateways where rule is applied to

Resilience

Retry

Retry 3 times when things go wrong before throwing the error upstream.

```
spec:
  hosts:
  - recommendation
  http:
  - retries:
      attempts: 3
      perTryTimeout: 4.000s
    route:
      -destination:
          host: recommendation
          subset: version-v1
```

Timeout

You can add timeouts to communications, for example aborting call after 1 second:

```
spec:
  hosts:
  - recommendation
  http:
  - route:
      - destination:
          host: recommendation
          timeout: 1.000s
```

Outlier detection/Circuit breaking

If the request is forwarded to a certain instance and it fails (e.g. returns a 5xx error code), then this instance of an instance/pod can be ejected to serve any other client request for an amount of time (outlier detection).

In next example we see outlier detection after 5 consecutive errors, ejection analysis every 15 seconds, and in the case of host ejection, the host will be ejected for 2 minutes.

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: recommendation
  namespace: tutorial
spec:
  host: recommendation
  trafficPolicy:
    outlierDetection:
      baseEjectionTime: 2m
      consecutiveErrors: 5
      interval: 15.000s
      maxEjectionPercent: 100
  subsets:
```

trafficPolicy can be applied at subset level to make it specific to a subset instead of all them.

You can also create connection pools at *tcp* and *http* level:

```
trafficPolicy:
  connectionPool:
    http:
      http1MaxPendingRequests: 100
      http2MaxRequests: 100
      maxRequestsPerConnection: 1
    tcp:
      maxConnections: 100
      connectTimeout: 50ms
```

Traffic Policy possible values:

Field	Type	Description
loadbalancer	LoadBalancerSettings	Controlling load blancer algorithm
connectionPool	ConnectionPoolSettings	Controlling connection pool
outlierDetection	OutlierDetection	Controlling eviction of unhealthy hosts

Field	Type	Description
tls	TLSSettings	TLS settings for connections
portLevelSettings	PortTrafficPolicy[]	Traffic policies specific to concrete ports

Policy Enforcement

Istio provides a model to enforce authorization policies in the communication between policies. You can, for example, black-list or white-list intercommunication between services or add some quota.

You can configure that preference service only allows requests from the recommendation service.

```
apiVersion: "config.istio.io/v1alpha2"
kind: listchecker
metadata:
  name: preferencewhitelist
spec:
  overrides: ["recommendation"]
  blacklist: false
```

```
apiVersion: "config.istio.io/v1alpha2"
kind: listentry
metadata:
  name: preferencesource
spec:
  value: source.labels["app"]
---
apiVersion: "config.istio.io/v1alpha2"
kind: rule
metadata:
  name: checkfromcustomer
spec:
  match: destination.labels["app"] == "preference"
  actions:
    - handler: preferencewhitelist.listchecker
      instances:
        - preferencesource.listentry
```

Source part is configured by using `listchecker` (to provide the list of allows hosts) and `listentry` (to configure how to get whitelist value from the request) elements. Destination part and rule is configured by using the `rule` element.

Field	Type	Description
providerUrl	string	Url where to load the list to check against, can be empty

Field	Type	Description
refreshInterval	Duration	How often provider is polled
ttl	Duration	How long keep list before discarding it
cachingInterval	Duration	How long a caller can cache an answer befoer ask again
cachingUseCount	int	Number of times a caller can use a cached answer
overrides	string[]	List of entries consulted first before <code>providerUrl</code>
entryType	ListEntryType	The kind (STRINGS, CASE_INSENSITIVE_STRINGS, IP_ADDRESSES, `REGEX) of list entry and overrides
blacklist	boolean	the list operates as a blacklist or a whitelist

Telemetry, Monitoring and Tracing

Istio comes with observability providing out-of-the-box integration with Prometheus/Grafana and Jaeger (OpenAPI Spec).

Service to Service Security

You can secure the communication between all services by enabling mutual TLS (peer authentication).

Mutual TLS

First, you need to enable mutual TLS.

You can enable it globally with **MeshPolicy**

```
apiVersion: "authentication.istio.io/v1alpha1"
kind: "MeshPolicy"
metadata:
  name: "default"
spec:
  peers:
    - mtls: {}
```

Or more fine-grained with **Policy**; in this case by namespace:

```
apiVersion: "authentication.istio.io/v1alpha1"
kind: "Policy"
metadata:
  name: "default"
  namespace: "tutorial"
spec:
  peers:
    - mtls: {}
```

Applying mTLS to specific destination and port:

```
spec:
  target:
    - name: preference
      ports:
        - number: 9000
```

Important If `ports` not set then it is applied to all ports.

Field	Type	Description
peers	PeerAuthentication Method[]	List of authentication methods for peer auth
peerIsOptional	boolean	Accept request when none of the peer authentication methods defined are satisfied
targets	TargetSelector[]	Destinations where policy should be applied on. Enabled all by default
origins	OriginAuthentication Method[]	List of authentication methods for origin auth
originIsOptional	boolean	Accept request when none of the origin authentication methods defined are satisfied

Field	Type	Description
principalBinding	PrincipalBinding	Peer or origin identity should be use for principal. USE_PEER by default

End-user Authentication

End user authentication (origin authentication) using JWT:

```
spec:
  origins:
  - jwt:
      issuer: "https://keycloak/auth/realms/istio"
      audiences:
      - "customer-tutorial"
      jwksUri: >
        https://keycloak/auth/realms/istio
          /protocol/openid-connect/certs
      principaBinding: USE_ORIGIN
```

At this time, Origins only support JWT. Possible values for JWT are:

Field	Type	Description
issuer	string	Issuer of the token
audiences	string[]	List of JWT <i>audiences</i> allowed to access
jwksUri	string	URL of the public key to validate signature
jwtParams	string[]	JWT is sent in a query parameter
jwtHeaders	string[]	JWT is sent in a request header. If empty Authorization: Bearer \$token

After enabling mTLS, you need to configure it at the client side by using a DestinationRule. Need to set which hosts communicate through mTLS using host field.

```
apiVersion: "networking.istio.io/v1alpha3"
kind: "DestinationRule"
metadata:
  name: "default"
  namespace: "tutorial"
spec:
  host: "*.tutorial.svc.cluster.local"
  trafficPolicy:
    tls:
      mode: ISTIO_MUTUAL
```

If **ISTIO_MUTUAL** is set, Istio configures client certificate, private key and CA crtificates with its internal implementation.

Field	Type	Description
httpsRedirect	boolean	Send 301 redirect when communication is using HTTP asking to use HTTPS
mode	TLSmode	How TLS is enforced. Values <i>PASSTHROUGH, SIMPLE, MUTUAL</i>
serverCertificate	string	The location to the file of the server-side TLS certificate
privateKey	string	The location to the file of the server's private key
caCertificates	string	The location to the file of the certificate authority certificates
subjectAltNames	string[]	Alternate names to verify the subject identity

Istio RBAC

Istio’s authorization feature provides access control for services in an Istio Mesh.

To enable RBAC:

```
apiVersion: "rbac.istio.io/v1alpha1"
kind: RbacConfig
metadata:
  name: default
spec:
  mode: 'ON_WITH_INCLUSION'
  inclusion:
    namespaces: ["tutorial"]
```

Valid modes are: ON, OFF, ON_WITH_INCLUSION, ON_WITH_EXCLUSION. inclusion is used when WITH_INCLUSION and exclusion used when WITH_EXCLUSION. They support the next properties:

Field	Type	Description
services	string[]	A list of services
namespaces	string[]	A list of namespaces

Granting access (**what**) to all services, when using the GET method and given destination services.

```
apiVersion: "rbac.istio.io/v1alpha1"
kind: ServiceRole
metadata:
  name: service-viewer
  namespace: tutorial
spec:
  rules:
  - services: ["*"]
    methods: ["GET"]
    constraints:
      - key: "destination.labels[app]"
        values: ["customer", "recommendation", "preference"]
```

Field	Type	Description
services	string[]	List of service names to apply.
paths	string[]	List of HTTP paths
methods	string[]	List of HTTP methods
constraints	Constraint[]	Extra constraints

And the `Constraint` is an array of pairs `key (string)` and `values (string[])`. Valid keys are:

Key Example	Value Example
<code>destination.ip</code>	<code>["10.1.2.3", "10.2.0.0/16"]</code>
<code>destination.port</code>	<code>["80", "443"]</code>
<code>destination.labels[version]</code>	<code>["v1", "v2"]</code>
<code>destination.name</code>	<code>["productpage*"]</code>
<code>destination.namespace</code>	<code>["tutorial"]</code>
<code>destination.user</code>	<code>["customer-tutorial"]</code>
<code>request.headers[X-Custom-Token]</code>	<code>["345CFA3"]</code>

Granting to all subjects (**who**) previous defined roles (**what**).

```
apiVersion: "rbac.istio.io/v1alpha1"
kind: ServiceRoleBinding
metadata:
  name: bind-service-viewer
  namespace: tutorial
spec:
  subjects:
  - user: "*"
  roleRef:
    kind: ServiceRole
    name: "service-viewer"
```

Field	Type	Description
<code>user</code>	string	username/ID (Service Account).
<code>properties</code>	map	Properties to identify the subject

Next properties are supported:



Key Example	Value Example
<code>source.ip</code>	<code>"10.1.2.3"</code>
<code>source.namespace</code>	<code>"default"</code>
<code>source.principal</code>	<code>"customer"</code>
<code>request.headers[User-Agent]</code>	<code>"Mozilla/*"</code>
<code>request.auth.principal</code>	<code>"users.tutrial.org/654654"</code>
<code>request.auth.audiences</code>	<code>"tutorial.org"</code>
<code>request.auth.presenter</code>	<code>"654654.tutorial.org"</code>
<code>request.auth.claims[iss]</code>	<code>"*@redhat.com"</code>

Last property refers to JWT claim named `iss`. Obviously, you can use any other claim for this purpose. Usually, you might use `group` claim to allow access to users under a specific group.

Resources

- Istio In Action book
 - <https://www.manning.com/books/istio-in-action>
- Introducing Istio Service Mesh for Microservices
 - <https://developers.redhat.com/books/introducing-istio-service-mesh-microservices/>
- Red Hat Developer Istio-Tutorial
 - <https://redhat-developer-demos.github.io/istio-tutorial/>

Authors :

-  **@alexsotob**
Java Champion and SW Engineer at Red Hat
-  **@christianposta**
Author, Speaker, Chief Architect at Red Hat

v1.0.0