Getting Started With Istio

WRITTEN BY CHRISTIAN POSTA CTO AT SOLO.IO
AND ALEX SOTO SENIOR SOFTWARE ENGINEER AT RED HAT

CONTENTS

What Is a Service Mesh and Istio?

A **service mesh** is a decentralized application infrastructure for making service-to-service communication safe, reliable, and understandable.

A service mesh uses a "service proxy" deployed with each application instance to facilitate this functionality. A service proxy understands Layer 7 requests and messages and can route, secure, observe, and apply policy to these messages consistently and independently of how the service is implemented. The proxies deployed in a single cluster domain form the "mesh."

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

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

- Intelligent routing and client-side software load balancing
- Resilience against service and network failures
- Policy enforcement between services
- Observability of your L7 communication
- Securing service to service communication

Istio Architecture

Istio follows the typical service-mesh architecture with the following logical separation:

- **Data plane** that is composed of **Envoy** service proxies deployed (as a sidecar) along with your service through which all application traffic flows
- **Control plane** that manages and configures the data plane (**Envoy** service proxies) while also managing back-end infrastructure that complements the data plane (like metrics sinks, policy engines, and security infrastructure)

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

Tigera Secure EE for Microservice Security

Deep Network Security for Microservices on Kubernetes and Istio Service Mesh

Tigera Unified Security Controls

- **L-7** Application Layer
- **L-6** Presentation Layer
- **L-5** Session Layer
- **L-4** Transport Layer
- **L-3** Network Layer

- **L-2** Data Link Layer
- **L-1** Physical Layer

Visit the Tigera Istio Resource Center for more information: www.tigera.io/istio

- Technical Papers →
- On-Demand Webinars →
- Blog Posts →
- Customer Success Stories →

A service mesh, like Istio, should be a critical component of your microservices architecture.

When moving your microservice application to production, you will need to secure your application and enforce corporate and often regulatory security controls to meet internal security team or external industry/regulatory compliance requirements.

Tigera extends traditional and cloud-native firewalls to your Kubernetes and Istio environment, enforcing and reporting on compliance while integrating your microservice application into your security team's existing tools and processes.

Tigera Secure EE gives you:

- ✓ Enforcement of internal and regulatory security controls required by auditors for compliance
- ✓ Advanced network security including threat prevention, anomaly detection, and remediation
- ✓ Tiered policies allow security to enforce their controls while providing developers more freedom

Request a Demo Today

Learn how to meet security controls & requirements for your microservices.

www.tigera.io/demo

TIGERA

Key Concepts of Istio

DESTINATIONRULE

A **DestinationRule** configures the set of rules to be applied when sending traffic to an upstream service. Some of the configurations of a **DestinationRule** govern circuit breaking, client-side load balancing, and TLS setting. **DestinationRule**s are also used to define **subsets** (named versions) of the service hosts so they can be reused in other Istio elements like traffic routing and identification in metrics.

For example, to define two different 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 service instances (platform specific; for example, Kubernetes uses labels on its Pod objects and the labels used in the subsets here would refer to those Pod labels).

VIRTUALSERVICE

A **VirtualService** describes the mapping between one or more user-addressable virtual service names to the actual services inside the service mesh.

For example, to define a single "virtual service" where the traffic is split between two deployed versions with 90 percent going to version 1 and 10 percent 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 are not automatically discovered by the Istio control plane and typically (but not strictly) live outside of the service mesh and 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 proxy 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 configure 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 a starting point **without** automatic sidecar injection, so you understand each of the steps. If your level of deployment maturity is comfortable with automatic sidecar injection, it's possible it can save some steps when deploying your services.

INSTALLING ISTIO

First you need to download Istio and register in PATH:

```
open https://github.com/istio/istio/releases/
cd istio-1.1.8
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:

```
kubectl create namespace istio-system
for i in install/kubernetes/helm/istio-init/files/
crd*yaml; do kubectl apply -f $i; done
kubectl apply -f install/kubernetes/istio-demo.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 when matching a 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 subset version-v1.

Mirroring traffic between two versions:

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

Note, for mirroring traffic, the Host or Authority header gets appended with "-shadow". 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

Istio comes with an automatic retry that retries up to two times, but you can fine tune the retry policy on a VirtualService. For example, to set retries to three when calling *recommendation* service:

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

You can also fine-tune on what errors a retry is tried with the **retryOn** option.

TIMEOUT

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

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

OUTLIER DETECTION/CIRCUIT BREAKER

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 from the load-balancing pool to serve any other client request for a certain amount of time (outlier detection).

In the next example, we see outlier detection after five consecutive errors, ejection analysis every 15 seconds, and in the case of host ejection, the host will be ejected for 2m x (the number of ejections).

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

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 a specific destination and port:

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

If the ports field is not configured, then it applies to all ports.

FIELD	TYPE	DESCRIPTION
peers	PeerAuthenticationMethod[]	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	OriginAuthenticationMethod[]	List of authentication methods for origin auth
originIsOptional	boolean	Accept request when none of the origin authentication methods defined are satisfied
principalBinding	principalBinding	Peer or origin identity should be use for principal. USE_PEER by default

After enabling mTLS, you need to configure it on the client side by using a `DestinationRule`. You 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 certificates 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 PASSTHROUGH, SIMPLE, MUTUAL
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

END-USER AUTHENTICATION

End user authentication (origin authentication) using JWT:

```
apiVersion: "authentication.istio.io/v1alpha1"
kind: "Policy"
spec:
  targets:
    - name: customer
  origins:
    - jwt:
        issuer: "testing@secure.istio.io"
        jwksUri: https://keycloak/auth/realms/istio
        /protocol/openid-connect/certs
  principalBinding: 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 audiences 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
triggerRules	TriggerRule[]	List of trigger rules to decide if this JWT should be used to validate the request.

ISTIO RBAC

Istio's authorization feature provides access control for services in an Istio Mesh. To enable RBAC:

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

By default, Istio uses a deny by default strategy, meaning that nothing is permitted until you explicitly define access control policy to grant access to any service.

Valid modes are: `ON`, `OFF`, `ON_WITH_INCLUSION`, and `ON_WITH_EXCLUSION`. Inclusion is used when `WITH_INCLUSION` and exclusion is 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: customer
spec:
  rules:
    - services: ["*"]
      methods: ["GET"]
```

FIELD	TYPE	DESCRIPTION
services	string[]	A list of service names to apply
paths	string[]	A list of HTTP paths
methods	string[]	A 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
destiantion.ip	["10.1.2.3", "10.2.0.0/16"]
destination.port	["80", "443"]
destination.labels[ver]	["v1", "v2"]
destination.name	["productpage*"]
destiantion.namespace	["tutorial"]
destination.user	["customer-tutorial"]
request.headers[X-Tok]	["345CFA3"]

Granting to subjects with role customer (**who**) previous defined roles (**what**):

```
apiVersion: rbac.istio.io/v1alpha1
kind: ServiceRoleBinding
metadata:
  name: bind-customer
spec:
  subjects:
    - user: "*"
      properties:
        request.auth.claims[role]: "customer"
  roleRef:
    kind: ServiceRole
    name: customer
```

FIELD	TYPE	DESCRIPTION
user	string	username/ID (Service Account)
properties	map	Properties to identify the subject

Next properties are supported:

KEY EXAMPLE	VALUE EXAMPLE
source.ip	"10.1.2.3"
source.namespace	"default"
source.principal	"customer"
request.headers[User-Agent]	"Mozilla/*"
request.auth.principal	"users.tutrial.org/654654"
request.auth.audiences	"tutorial.org"
request.auth.presenter	"654654.tutorial.org"
request.auth.claims[iss]	"*@redhat.com"

The last property refers to a 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.

Written by **Christian Posta,** *CTO at solo.io*
Christian Posta is author of "Istio in Action" and "Microservices for Java Developers," and an open-source enthusiast. He is a committer at Apache and a blogger on topics related to Serverless, Cloud, Integration, Kubernetes, Docker, Istio, and Envoy.

Written by **Alex Soto,** *Senior Software Engineer at Red Hat*
Alex Soto is a senior software engineer at Red Hat. He is a Java Champion, and he is a passionate about the world of Java and software automation. He believes in the open source software model. Alex is the creator of NoSQLUnit project and a member of JSR374 (Java API for JSON Processing) Expert Group. He is an international speaker, presenting his talks at software conferences like Devoxx, JavaOne, JavaZone or JavaLand.

Devada, Inc.
600 Park Offices Drive
Suite 150
Research Triangle Park, NC

888.678.0399 919.678.0300