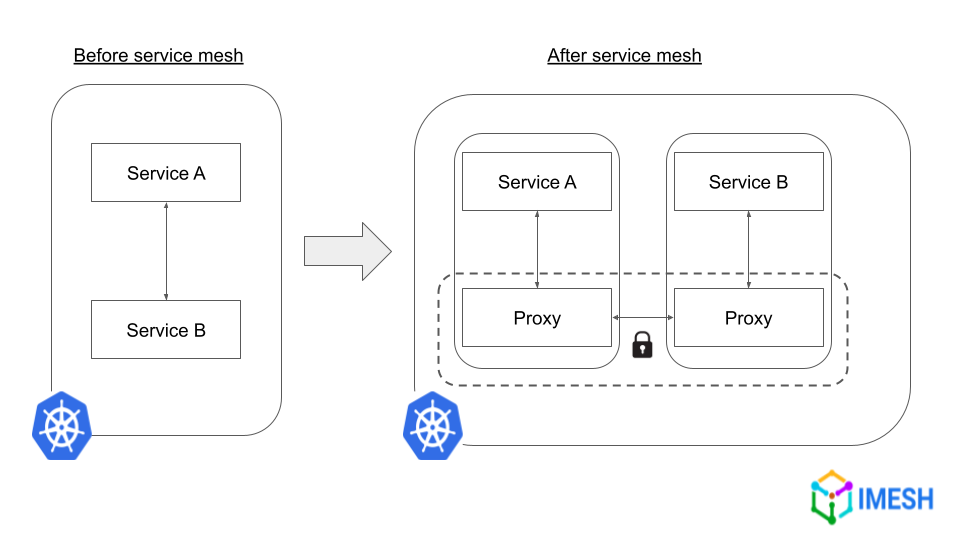
**Each microservices has to implement**

1. Own **business** logic, the core business functionality of the service.
2. **Communication configuration**, eg: making endpoints of service available to others
3. **Security Logic**, restrict list of services which can invoke the other service especially when communication is between services in different clusters.
4. **Retry logic** to make the application more robost and overcome transient exception situations.
5. Capture **metrics and tracing logic** for performance, errors and identity bottlenecks.

**What is a Service Mesh?**

A service mesh is a **dedicated infrastructure layer** for handling **service-to-service communication** within a distributed microservices based application.

It **controls** the delivery of **service requests** in an application.



It uses **proxy-based communication** where the proxies are built into an application.

The proxy authenticates messages, authorizes them, and encrypts/decrypts them using TLS, among other functions.

A service mesh contains two parts.

1. Data Plane
2. Control Plane

A diagram of a service

Description automatically generated

It provides following functions **without** **requiring** changes to the application code.

* Service discovery
* Load balancing
* TLS encryption
* Authentication and authorization
* Metrics aggregation, such as request throughput and response time
* Distributed tracing
* Rate limiting
* Routing and traffic management
* Traffic splitting
* Request retries
* Error handling

Istio

Istio is the most popular, CNCF-graduated open-source service mesh software available.

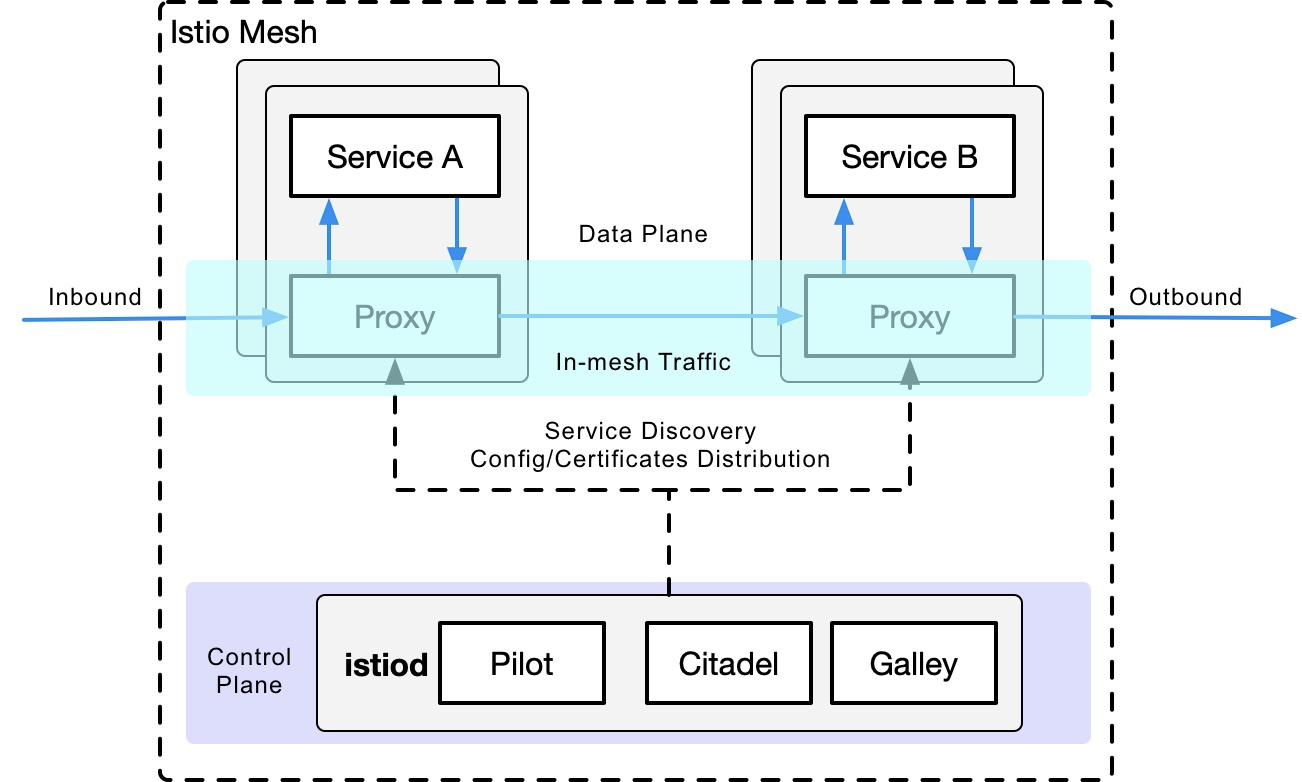
Istio service mesh provides a **control plane** to define and implement the way microservices communicate with each other.

Istio adds to every pod a **sidecar** **container** called as **Envoy Proxy**. Envoy is responsible for all service interaction in Kubernetes.

**Features of Istio Service Mesh:**

1. **Security**: All your data communicated among the services, in and outside of the cluster or data center, will be encrypted based on **mTLS protocols** provided by Istio resources. You can also ensure **authentication** of apps from internal and external users using JSON Web Tokens (JWT) provided by Istio.
2. **Traffic Management**: Istio provides features like load balancing, health checks, and deployment strategies. Istio allows load balancing based on algorithms that include round robin, random selection, weighted algorithms, etc. Istio performs constant health checks of service instances to ensure they are available before routing the traffic request.
3. **Resilience**: Istio removes the need for coding circuit breakers within an application. Istio helps platform architects to define mechanisms such as timeouts to a service, number of retries to be made and planned automatic failover of high availability (HA) systems, without the application knowing about them.
4. **Observability**: Istio keeps track of network requests and traces each call across multiple services. Istio provides telemetry (such as latency, saturation, traffic health, and errors).
5. **Advanced Deployment**: Istio helps to achieve canary and blue-green deployment by providing the capability to route specific user groups to newly deployed applications.

**Istio Components:**



1. **Envoy Proxy (Sidecar Proxy)**

* Each service in your cluster has an Envoy sidecar proxy running alongside it.
* This proxy intercepts all incoming and outgoing traffic to the service.
* Envoy in the data plane is responsible for functionalities like failure handling, health checks, load balancing and enforcing security using mTLS.

1. **Pilot (Traffic Management)**

* Pilot manages service discovery and traffic routing rules for the proxies.
* It allows you to configure routing rules such as canary releases, retries, or fault injection.
* It ensures that Envoy proxies are updated with the latest routing and load balancing policies.

1. **Citadel (Security):**

* Citadel provides security services like issuing and managing service identities, certificates, and enforcing mutual TLS (mTLS) between services.

1. **Mixer (Telemetry & Policy):**

* Mixer is responsible for collecting telemetry data (like logs, metrics, and traces) from Envoy proxies and enforcing policies (such as rate limiting or authorization).

1. **Gallery (Configuration):**

* Galley validates and processes the configuration data for Istio. It ensures that configuration applied to the system is correct and consistent.

**Install ISTIO**

[**https://istio.io/latest/docs/setup/getting-started/**](https://istio.io/latest/docs/setup/getting-started/)

Traffic Management

Traffic management in a service mesh involves controlling the flow of requests between services. This includes features such as load balancing, routing, traffic shaping, and fault tolerance mechanisms like retries and circuit breaking.

**This task shows you how to route requests dynamically to multiple versions of a microservice.**

**#Create Namespaces**

kubectl create ns istio-ns1

kubectl create ns istio-ns2

kubectl create ns not-istio-ns

**#demo.yaml**

apiVersion: v1

kind: Pod

metadata:

name: **mypod-v1**

namespace: istio-ns1

labels:

app: my-app

**version: v1**

spec:

containers:

- name: my-con

image: **nginx**

ports:

- containerPort: 80

---

apiVersion: v1

kind: Pod

metadata:

name: **mypod-v2**

namespace: istio-ns1

labels:

app: my-app

**version: v2**

spec:

containers:

- name: my-con

image: **httpd**

ports:

- containerPort: 80

---

apiVersion: v1

kind: Service

metadata:

name: **myservice**

namespace: istio-ns1

spec:

type: ClusterIP

selector:

app: my-app

ports:

- protocol: TCP

port: 8080

targetPort: 80

---

apiVersion: v1

kind: Pod

metadata:

name: pod1

namespace: istio-ns2

labels:

app: web

spec:

containers:

- name: my-con

image: nginx

ports:

- containerPort: 80

**#Create/Update Deployment, Service and Test POD**

kubectl apply -f demo.yaml

kubectl get all -n istio-ns1

kubectl get all -n istio-ns2

kubectl get all -n not-istio-ns

**#Enable ISTIO for istio-ns1, istio-ns2 namespaces**

kubectl label namespace istio-ns1 istio-injection=enabled

kubectl label namespace istio-ns2 istio-injection=enabled

kubectl delete -f demo.yaml

kubectl apply -f demo.yaml

**#Note that the Pods in istio-ns1 and istio-ns2 namespace have two containers including the side-car container added by ISTIO.**

kubectl get all -n istio-ns1

kubectl get all -n istio-ns2

kubectl get all -n not-istio-ns

**Destination Rules**

Destination rules in Istio configure the traffic policies applied to the traffic destined for a particular service version or subset. They define things like load balancing algorithms, circuit breaking settings, and TLS settings for communication between services.

**Virtual Services**

Virtual services in Istio define how incoming requests to a service should be routed to different versions or subsets of that service. They enable sophisticated traffic management strategies like A/B testing and canary deployments.

**Create Destination Rule**

apiVersion: networking.istio.io/v1alpha3

kind: **DestinationRule**

metadata:

name: myservice-dr

**namespace: istio-ns1**

spec:

host: myservice.istio-ns1.svc.cluster.local

subsets:

- name: v1

labels:

version: v1

- name: v2

labels:

version: v2

**Create Virtual Service Routing all traffic to Version1**

apiVersion: networking.istio.io/v1alpha3

kind: **VirtualService**

metadata:

name: myservice-vs

**namespace: istio-ns1**

spec:

hosts:

- myservice.istio-ns1.svc.cluster.local

http:

- route:

- destination:

host: myservice.istio-ns1.svc.cluster.local

subset: v1

**Following will always get o/p from Version1**

k exec -it pod1 -n istio-ns2 – bash

k exec -it pod1 -n istio-ns2 -- curl <http://myservice.istio-ns1:8080>

**Distributing traffic by Weight**

apiVersion: networking.istio.io/v1alpha3

kind: VirtualService

metadata:

name: myservice-vs

namespace: istio-ns1

spec:

hosts:

- myservice.istio-ns1.svc.cluster.local

http:

- route:

- destination:

host: myservice.istio-ns1.svc.cluster.local

subset: v1

**weight: 1**

- destination:

host: myservice.istio-ns1.svc.cluster.local

subset: v2

**weight: 1**

mTLS Authentication

**What is TLS?**

Transport Layer Security (TLS) is an encryption protocol in wide use on the Internet. TLS, which was formerly called SSL, authenticates the server in a client-server connection and encrypts communications between client and server so that external parties cannot spy on the communications.

There are three important things to understand about how TLS works:

**1. Public key and private key**

TLS works using a technique called public key cryptography, which relies on a pair of keys — a public key and a private key. Anything encrypted with the public key can be decrypted only with the *private* key.

Therefore, a server that decrypts a message that was encrypted with the public key proves that it possesses the private key. Anyone can view the public key by looking at the domain's or server's TLS certificate.

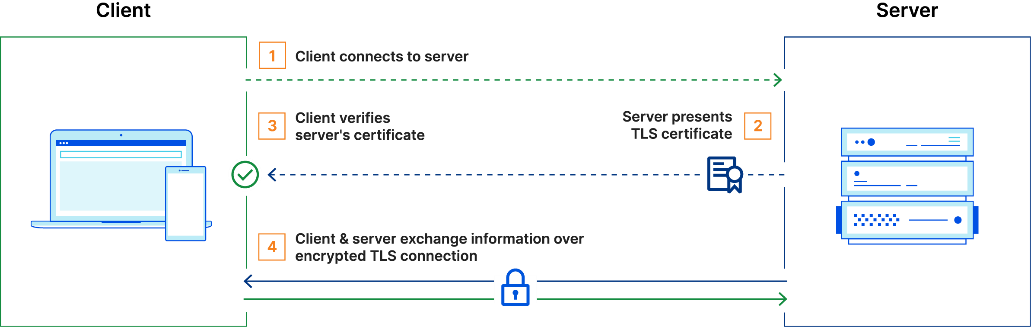
**2. TLS certificate**

A TLS certificate is a data file that contains important information for verifying a server's or device's identity, including the public key, a statement of who issued the certificate (TLS certificates are issued by a certificate authority), and the certificate's expiration date.

**3. TLS handshake**

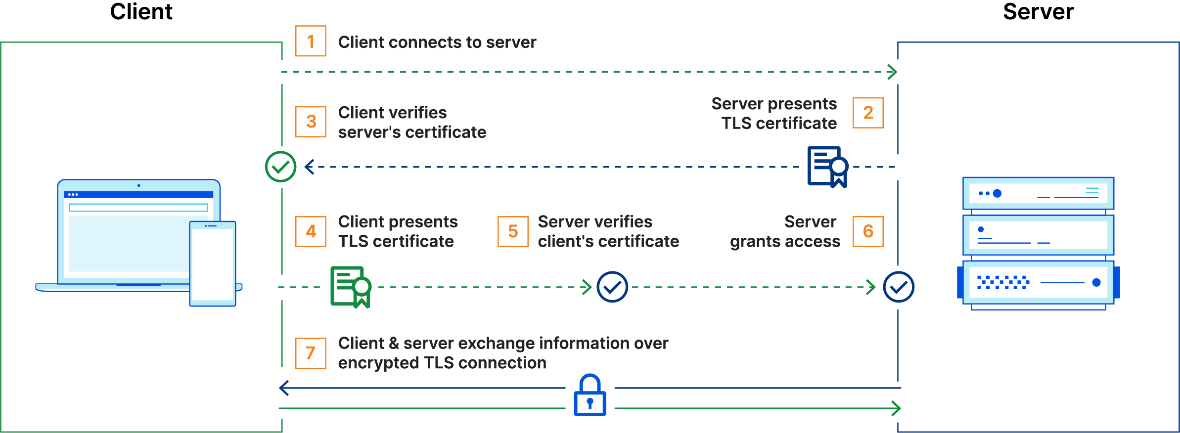
The TLS handshake is the process for verifying the TLS certificate and the server's possession of the private key. The TLS handshake also establishes how encryption will take place once the handshake is finished.

**How TLS Works:**



**How does mTLS Works**

In mTLS, however, both the client and server have a certificate, and both sides authenticate using their public/private key pair.



**Further Reading**: <https://www.cloudflare.com/learning/access-management/what-is-mutual-tls/>

In Istio, when a client/service sends a request to another service/server using mutual TLS authentication, the request is handled as follows:

1. Istio re-routes the outbound traffic from a client to the client’s local sidecar Envoy.
2. The client side Envoy starts a mutual TLS handshake with the server side Envoy. During the handshake, the client side Envoy also does a secure naming check to verify that the service account presented in the server certificate is authorized to run the target service.
3. The client side Envoy and the server side Envoy establish a mutual TLS connection, and Istio forwards the traffic from the client side Envoy to the server side Envoy.
4. The server side Envoy authorizes the request. If authorized, it forwards the traffic to the backend service through local TCP connections.

Istio mutual TLS has a **permissive mode**, which allows a service to accept both plaintext traffic and mutual TLS traffic at the same time.

By default, Istio configures the destination workloads using **PERMISSIVE** mode. In order to only allow mutual TLS traffic, the configuration needs to be changed to **STRICT** mode.

**Append the following to demo.yaml**

apiVersion: v1

kind: Pod

metadata:

name: pod1

**namespace: not-istio-ns**

labels:

app: web

spec:

containers:

- name: my-con

image: nginx

ports:

- containerPort: 80

**#Confirm that Client in both namespaces, ISTIO enable and ISTIO not enabled are able to access services in ISTIO**

kubectl exec -it pod1 -n istio-ns2 -c nginx-con -- curl http://my-service-1.istio-ns1:8080

kubectl exec -it pod1 -n not-istio-ns -c nginx-con -- curl http://my-service-1.istio-ns1:8080

**Create mTLS-enable.yaml**

apiVersion: security.istio.io/v1

kind: **PeerAuthentication**

metadata:

name: istio-ns1

namespace: **istio-ns1**

spec:

mtls:

mode: **STRICT**

kubectl apply -f mTLS-enable.yaml

**#Following will work**

kubectl exec -it pod1 -n istio-ns2 -c nginx-con -- curl <http://my-service-1.istio-ns1:8080>

**#Following won’t work as the pod is not managed by ISTIO**

kubectl exec -it pod1 -n not-istio-ns -c nginx-con -- curl http://my-service-1.istio-ns1:8080

Admission Controllers

Admission controllers in Kubernetes enforce policies on objects during their creation or modification. They intercept requests to the Kubernetes API server before objects are persisted and can modify or deny requests based on predefined rules.

**Minikube ssh**

**sudo cat /etc/Kubernetes/manifest/kube-api.yaml 🡪 look for –enable-admission-plugins**

**Mutation Example:**

**pvc.yaml**

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: myclaim

spec:

accessModes:

- ReadWriteOnce

volumeMode: Filesystem

resources:

requests:

storage: 8Gi

selector:

matchLabels:

release: "stable"

matchExpressions:

- {key: environment, operator: In, values: [dev]}

kubectl apply -f pvc.yaml

kubectl edit PVC myclaim

Note that it has StorageClass property.