

ISTIO

SUCCINCTLY

BY **RAHUL RAI AND
TARUN PABBI**

Istio Succinctly

By

Rahul Rai and Tarun Pabbi

Foreword by Daniel Jebaraj



Copyright © 2020 by Syncfusion, Inc.

2501 Aerial Center Parkway

Suite 200

Morrisville, NC 27560

USA

All rights reserved.

Important licensing information. Please read.

This book is available for free download from www.syncfusion.com on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

Technical Reviewer: James McCaffrey

Copy Editor: Courtney Wright

Acquisitions Coordinator: Tres Watkins, VP of content, Syncfusion, Inc.

Proofreader: Graham High, senior content producer, Syncfusion, Inc.

Table of Contents

The Story Behind the <i>Succinctly</i> Series of Books.....	8
About the Authors	10
The ReadMe of <i>Istio Succinctly</i>.....	12
Chapter 1 Service Mesh and Istio	13
Service mesh	15
Sidecar	18
Service mesh architecture	18
Istio architecture	20
Ingress/egress gateway.....	21
Istio proxy.....	22
Pilot.....	22
Galley.....	24
Mixer	25
Citadel.....	27
Summary.....	29
Chapter 2 Installation.....	30
Prerequisites	30
Installation	32
Installing Istio with Helm	32
Summary.....	35
Chapter 3 Envoy Proxy.....	36
Micro Shake Factory	36
Deploying services	38
Ingress gateway	41

Manual sidecar injection	42
Summary	49
Chapter 4 Traffic Management: Part 1	50
Life of a network packet.....	50
Networking API.....	52
Service entry	54
Destination rule	61
Connection pool settings.....	62
TLS settings.....	64
Load balancer settings.....	66
Outlier detection.....	67
Subset	69
Summary.....	70
Chapter 5 Traffic Management: Part 2	71
Virtual service.....	71
HTTPRoute.....	74
HTTPRewrite	79
Destination.....	81
HTTPRouteDestination	82
HTTPRedirect.....	83
HTTPFaultInjection	83
Timeout	84
HTTPRetry.....	85
Mirror	86
TCPRoute.....	86
TLSRoute	87

Gateway	88
Pattern 1: Request routing	91
Pattern 2: Fault injection	93
Pattern 3: Traffic shifting	95
Pattern 4: Resilience	96
Pattern 5: Mirroring	99
Summary	99
Chapter 6 Mixer Policies	100
Policy control flow	101
Telemetry control flow	102
Adapter	102
Handler	102
Instance	103
Rule	104
QuotaSpec	104
QuotaSpecBinding	105
Testing the Mixer policy	105
Summary	107
Chapter 7 Security	108
Security architecture	109
Istio authentication	110
Transport authentication: mTLS	110
Applying transport authentication	113
Applying origin authentication	115
Authorization policy	117
Istio Authorization Policy	122

Securing ingress.....	122
Simple TLS	122
HTTPS redirect.....	125
Mutual TLS	127
TLS passthrough	128
Cleanup	130
Summary	130
Chapter 8 Observability	131
Metrics	131
Traces	136
Logs	139
Mesh visualization	141
Istioctl observability utilities	144
Summary	144
Chapter 9 Next Steps	145
Service Mesh Interface.....	145
Knative	147
Build	148
Serve	148
Events	148
Istio performance.....	149
Control plane performance	149
Data plane performance	149
Multi-cluster mesh	150
Summary	150

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!



About the Authors

Rahul Rai

Rahul is a versatile cloud architect based in Sydney, Australia, with over eleven years of experience in cloud and web technologies. He is passionate about leadership, technology, and problem-solving.

Rahul has a proven track record of applying innovative technologies and sound processes to bring efficiency to enterprise technology operations. He worked with major consulting companies, including Microsoft Consulting Services and Telstra Purple, focusing on leading delivery teams to build and maintain complex solutions on the cloud. He has thorough experience of working with clients from various industry verticals to design enterprise applications and foster innovation, collaboration, and development in teams.

Rahul has authored two books on microservices orchestrators: [Microservices with Azure](#), on Azure Service Fabric, published by Packt, and [Kubernetes Succinctly](#), on Kubernetes, published by Syncfusion. He actively blogs on the cloud and other emerging technologies on his blog, [The Cloud Blog](#).

“For my wife, parents, sister, and dear niece.
Thank you for your unconditional love and support.”



Tarun Pabbi

Tarun works as a cloud consultant in Sydney, Australia, with over twelve years of work experience.

As a cloud consultant, he works on large-scale and complex IT engagements for enterprises and helps them build new applications as well as migrate existing applications to the cloud. He has also extensively worked on Kubernetes in production environments. He worked for over four years with Microsoft, where he primarily worked on building Microsoft Azure solutions. He has also worked with an augmented reality startup, Foyr.com, to build an automated interior designer product for real estate developers. His interests include DevOps, machine learning, and identity and access management. He has worked extensively on cloud computing and IoT applications and has deep experience working on major cloud platforms such as Microsoft Azure, AWS, and GCP.

Tarun is the author of [Kubernetes Succinctly](#), published by Syncfusion. You can connect with him via his blog at tarunpabbi.com.



“A big thanks to my parents, my wife, and my son for their support and understanding throughout the writing of this book.”

Acknowledgments

Many people work behind the scenes at Syncfusion to bring a *Succinctly* book to life. Our special thanks to Tres Watkins for initiating the process and being with us throughout the journey of authoring this book. We would also like to thank everyone on the editorial, production, and marketing teams at Syncfusion. Many thanks to our technical reviewer, James M, who worked with us through the multiple revisions of the draft.


In our careers, we've met people who believed in our capabilities and provided us opportunities that helped us develop into the individuals capable of writing this book. Our many thanks to Paul Glavich (Principal Consultant, Telstra Purple), Atul Malaviya (Principal Program Manager, Microsoft), and Manish Sharma (Senior Program Manager, Microsoft), who mentored us and reviewed the early drafts of this book. They spent many hours reading our cluttered documents and provided us with valuable feedback that helped ensure that this book meets your expectations.

The ReadMe of *Istio Succinctly*

Service meshes are steadily gaining momentum in cloud-native infrastructure, and they are changing the architecture of modern applications. Out of all the service meshes in the market today, Istio has the richest set of features and capabilities, and hence it overshadows the service mesh landscape. This book aims to serve as a super-condensed, practical walkthrough of Istio. Although this book is short, it packs the core concepts and guidance that you can use to deploy and operate with Istio at scale.

Although Istio does not mandate the host on which it executes, you will find that most organizations deploy Istio on Kubernetes clusters. Therefore, in this book, we will discuss Istio in the context of Kubernetes. It is essential that you have a good understanding of Kubernetes so that you can perform necessary activities, such as deploying services and viewing the logs, with ease. You can refer to our previous Syncfusion title, [Kubernetes Succinctly](#), if you need a quick refresher on Kubernetes before diving into this book.

Other resources for this book are available at the following locations.

GitHub	Code
	https://github.com/Istio-Succinctly
	Container Images
	https://cloud.docker.com/u/istiosuccinctly/

The GitHub account for this title consists of three repositories:

- **MicroShakeFactory:** This is a simple application with two microservices that can interact with each other. The microservices are REST APIs written in [TypeScript](#) using [Restify](#), which is a popular Node.js service framework used for building REST APIs. You are not required to understand the intricacies of this application, as we have already published the container images of the microservices to Docker Hub. In this book, we will only create containers with those images, and you are not required to make any changes to the application code.
- **Policies:** This is the core repository that we will reference throughout this book. It contains all the Istio specifications that we will apply to the mesh as we discuss the various concepts of Istio. In the repository, you will find that the specifications (or manifests) are segregated by chapters of this book into folders, and specifications in a folder are independent from those in other folders.



Note: Throughout our discussion, we will only specify the name of the specification/manifest that you need to apply to your cluster. You will find the specification file inside the folder of the corresponding chapter.

- **ExoticFruits:** This repository contains a single file from which we will read raw data to demonstrate how you can invoke external APIs from services within the mesh.

Welcome to *Istio Succinctly*—we hope you enjoy reading this book.

Chapter 1 Service Mesh and Istio

Organizations all over the world are in love with microservices. Teams that adopt microservices have the flexibility to choose their tools and languages, and they can iterate designs and scale quickly. However, as the number of services continues to grow, organizations face challenges that can be broadly classified into two categories:

- Orchestrate the infrastructure that the microservices are deployed on.
- Consistently implement the best practices of service-to-service communication across microservices.

By adopting container orchestration solutions such as [Docker Swarm](#), [Kubernetes](#), and [Marathon](#), developers gain the ability to delegate infrastructure-centric concerns to the platform. With capabilities such as cluster management, scheduling, service discovery, application state maintenance, and host monitoring, the container orchestration platforms specialize in servicing layers 1–4 of the [Open Systems Interconnection](#) (OSI) network stack.

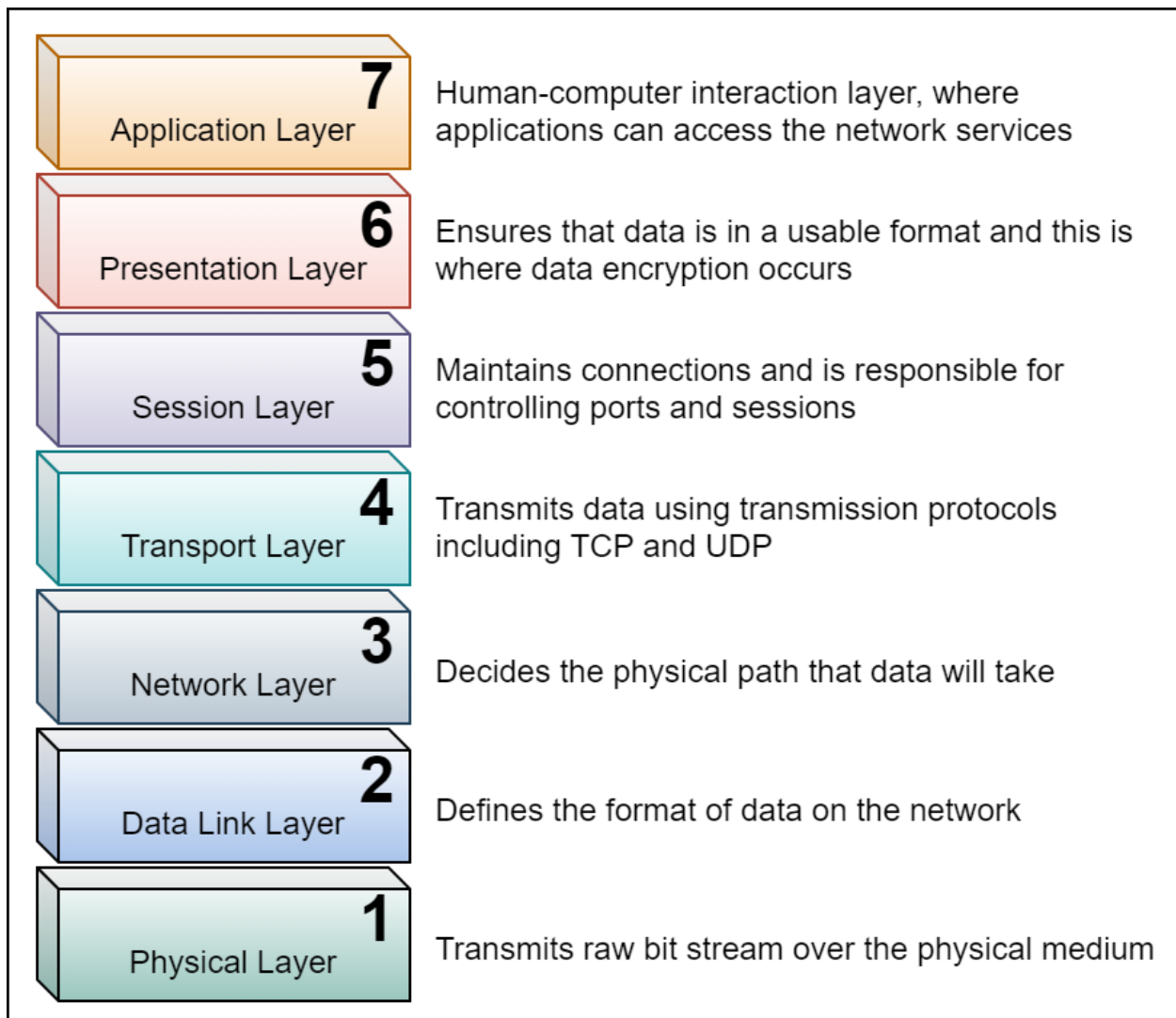


Figure 1: OSI stack ([source](#))

Almost all popular container orchestrators also provide some application life-cycle management (ALM) capabilities at layers 5–7, such as application deployment, application health monitoring, and secret management. However, often these capabilities are not enough to meet all the application-level concerns, such as rate-limiting and authentication.

Application gateways such as Ambassador, Kong, and Traefik solve some of the service-to-service communication concerns. However, application gateways are predominantly concerned with managing the traffic that passes through the data center to network infrastructure, also known as north-south network traffic. Application gateways lack the capabilities to manage the communication between microservices within the network, also known as east-west network traffic. Thus, application gateways complement (but do not replace the need for a solution that can manage) the service-to-service communication. Figure 2 is a high-level design diagram that presents where application gateways fit within the architecture of a system.

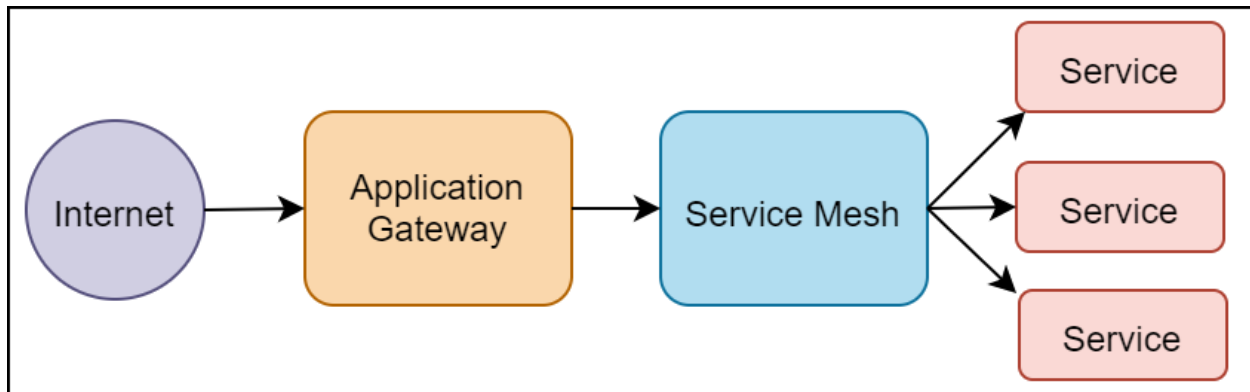


Figure 2: Application gateway and service mesh

Finally, in an unrealistically utopian scenario, an organization can enforce all the individual microservices to implement standard service-to-service communication concerns such as service discovery, circuit breaking, and rate-limiting. However, such a solution immediately starts facing issues with enforcement of consistency. Different microservices, constrained by variables such as programming language, availability of off-the-shelf packages, and developer proficiency, implement the same features differently. Even if the organization succeeds in implementing this strategy, maintaining the services, and repeated investments required to keep the policies up to date, become a significant challenge.

Service mesh

Due to these issues, there is a need for a platform-native component that can take care of network-related concerns so that application developers can focus on building features that deliver business value. The service mesh is a dedicated layer for managing service-to-service communication and mostly manages layer 5 through layer 1 of the OSI network stack. The service mesh provides a policy-based services-first network that addresses east-west (service-to-service) networking concerns such as security, resiliency, observability, and control, so that services can communicate with each other while offloading the network-related concerns to the platform. The value of using service meshes with microservices scales linearly with the number of microservices. The higher the number of microservices, the more value organizations can reap from it.



Note: The service mesh is named so because of the physical network topology that it uses to operate in a cluster. Each node in a mesh topology is connected directly to several other nodes. A mesh network generally configures itself and dynamically distributes the workload. The mesh is, therefore, the design of choice for service mesh.

Following are some of the key responsibilities that a service mesh offloads from microservices:

- **Observability:** The ability to extract metrics from executing services without instrumenting the services themselves. Service meshes provide consistent metrics across all the services that make up the application, so that developers can trace every operation as it flows across services. In service meshes, the capability to collect traces and metrics is independent of the metrics backend provider. Therefore, organizations can choose the metrics backend according to their needs. Observability is composed of the following three features:
 - **Logging:** The service mesh enforces baseline visibility of operations to all microservices. What this means is that even if a microservice does not log anything, the service mesh records information such as source and destination, request protocol, response, and response status code.
 - **Metrics:** Without any instrumentation, the service mesh emits telemetry such as overall request volume, success rate, and source. This information is useful for automating operations such as autoscaling.
 - **Tracing:** Tracing helps track operations across services and dependencies. To enable tracing, microservices are required to forward context headers, and the rest of the configuration, such as the generation of span IDs, is handled by the service mesh. Traces are used to visualize information such as dependencies, request volume, and failure rates.
- **Traffic control:** There are three key traffic control features that are required by microservices and provided by service meshes. The first feature is traffic splitting based on information available at layer 7, such as cookies and session identifiers. Applications usually use this feature for A/B testing to validate a new release. The second key feature in this category is traffic steering, with which a service mesh can look into the contents of a request and route it to a specific set of instances of a microservice. Finally, microservices can use a service mesh gateway to apply access rules, such as a whitelist or blacklist created by the administrator to route the ingress (incoming) and egress (outgoing) traffic.
- **Resiliency:** To counter an unstable network, microservices are required to implement resiliency measures such as timeouts and retries when trying to access out-of-process resources and other microservices. In addition to automatic retries, service meshes support some of the common resiliency design patterns such as circuit breaker, health checks, and many others, which help microservices gain control over the chaotic network that they operate.
- **Efficiency:** Service meshes do not significantly degrade the performance of applications in return for the flexibility they offer. One of the primary goals of service meshes is to apply a minimal resource overhead and scale flexibly with microservices.
- **Security:** For service meshes, security includes three distinct capabilities. The first one is authentication. Service meshes support several authentication options such as mTLS, [JWT](#) validation, and even a combination of the two. Service meshes also provide service-to-service and user-to-service authorization capabilities. The two types of authorizations that are supported by service meshes are role-based access control

(RBAC) and attribute-based access control (ABAC). We will discuss these policies later in this book. Finally, service meshes can enforce a zero-trust network by assigning trust based on identity as well as context and circumstances. With service meshes, you can enforce policies such as mTLS, RBAC, and certificate rotation, which help create a zero-trust network.

- **Policy:** Microservices require enforcement of policies such as rate limiting and access restrictions, among others, for addressing security and non-functional requirements. Service meshes allow you to configure custom policies to enforce rules at runtimes such as rate limiting, denials, and whitelists to restrict access to service, header rewrites, and redirects.

The service mesh helps decouple operational concerns from development so that both the operations and development teams can iterate independently. For example, with a service mesh managing the session layer (layer 5) of the microservice hosting platform, operations need not depend on developers to apply a consistent resiliency strategy such as retries on all microservices. As another example, customer teams do not need to depend on the development or operations team to enforce quotas based on pricing plans. For organizations that depend on many microservices for their operations, the service mesh is a vital component that helps them compose microservices and allow their various teams, such as development, operations, and customer teams, to work independently of each other.

[Istio](#) is one of the open-source implementations of a service mesh. It was initially built by Lyft, Google, and IBM, but it is now supported and developed by organizations such as Red Hat, and many individual contributors from the community. Istio offloads horizontal concerns of applications such as security, policy management, and observability to the platform. Istio addresses the application-networking concerns through a proxy that lives outside the application. This approach helps applications that use Istio stay unaware of its presence, and thus requires little to no modification to existing code. Although Istio works best for microservices or SOA architectures, it helps organizations that have several legacy applications reap its benefits because of its nature of operating out of band (sidecar pattern) from the underlying application.



Note: Many open-source projects within the Kubernetes ecosystem have nautical Greek terms as names. Kubernetes is the Greek name for “helmsman.” Istio is a Greek term that means “sail.”

Another popular service mesh available today is [Linkerd](#) (pronounced *linker-dee*), which is different from Istio in that its data plane (responsible for translating, forwarding, and observing every network packet that flows to and from a service instance) and control plane (responsible for providing policy and configuration for all of the running data planes in the mesh) are included in a single package. It is an open-source service written in Scala, and it supports services running in container orchestrators like Kubernetes as well as virtual and physical machines. Both Istio and Linkerd share the same model of data plane deployment known as the sidecar pattern. Let’s discuss the pattern in detail.

Sidecar

Let us briefly discuss the sidecar pattern, since this pattern is the deployment model used in Istio to intercept the traffic coming in or going out of services on the mesh. A sidecar is a component that is co-located with the primary application, but runs in its own process or container, providing a network interface to connect to it. While the core business functionalities are driven from the main application, the other common crosscutting features, such as network policies, are facilitated from the sidecar. Usually, all the inbound and outbound communication of the application with other applications takes place through the sidecar proxy. The service mesh proxy is always deployed as a sidecar to the services on the mesh. By introducing service mesh, the communication between the various services on the mesh happens via the service mesh proxy.

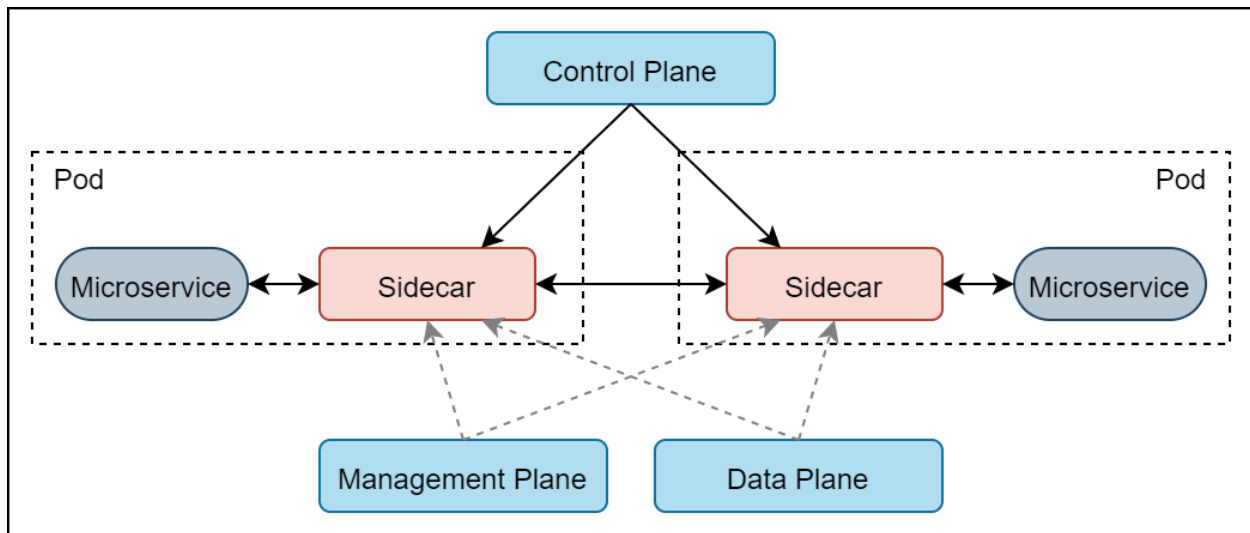


Figure 3: Sidecar pattern in service mesh

As evident from the previous diagram, all inbound (ingress) and outbound (egress) traffic of a given service goes through the service mesh proxy. Since the proxy is responsible for application-level network functions such as circuit breaking, the microservice is limited to primitive network functions such as invoking an HTTP service.

Service mesh architecture

Since Istio is an implementation of the service mesh, we will first discuss the architecture of the service mesh and then discuss how the components of Istio fill it up.

Since the concept of a service mesh is derived from physical network topologies, it is necessary to understand the concept of planes in networking. The following are the three planes or areas of operations in a [software defined network](#) (SDN) in increasing order of proximity to data packets being transmitted in the network:

- **Data plane:** Functions or processes that receive, parse, and forward network packets from one interface to another.

- **Control plane:** Functions or processes that determine the path that network packets use. This plane understands the various routing protocols such as spanning tree and LDP which it uses to update the routing table used by the data plane.
- **Management plane:** Functions or processes that control and monitor the device and hence, the control plane. In networking, this plane employs protocols such as SNMP to manage devices.

The concept of network plane components of layer 4 when applied to layer 5 and above form the concepts of a service mesh. There are over a dozen service mesh implementations; however, all implementations broadly consist of the same three planes that we previously discussed. Some service meshes such as Istio combine the management and control planes in the control plane.

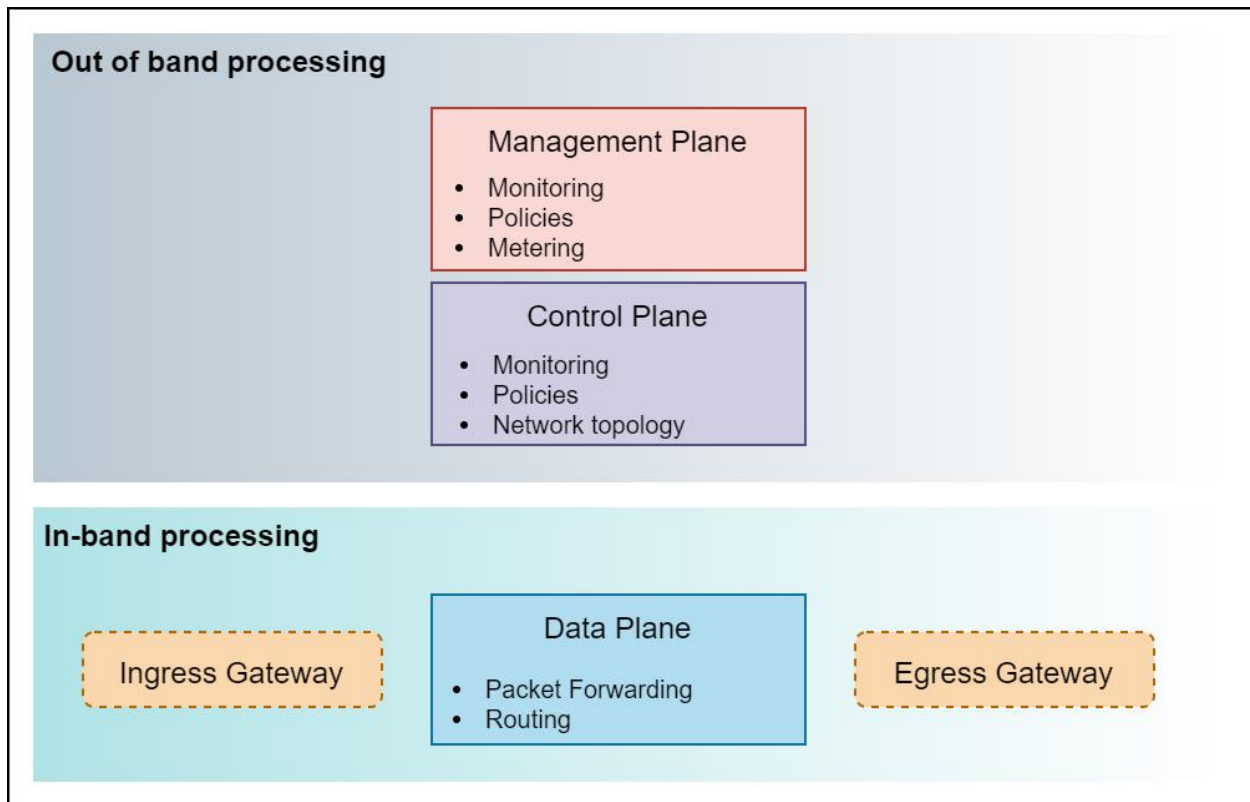


Figure 4: Service mesh planes

In a service mesh, the three planes play the following roles:

- **Data plane:** This component intercepts all traffic of all the requests that are sent to the applications on the mesh. It is also responsible for low-level application services such as service discovery, health checks, and routing. It manages load balancing, authentication, and authorization for the requests that are sent to the application. This plane collects metrics such as performance, scalability, security, and availability. This is the only component that touches the packets or requests on the data path.
- **Control plane:** This component monitors, configures, manages, and maintains the data planes. It provides policies and configurations to data planes, and thus converts the data planes deployed as isolated stateless sidecar proxies on the cluster to a service mesh.

- **Management plane:** This component extends the features of the system by providing capabilities such as monitoring and backend system integration. It enhances the control planes by adding support for continuous monitoring of policies and configurations applied by the control plane for ensuring compliance.

In addition to the three planes, a service mesh supports two types of gateways that operate at the edge of the service mesh. The ingress gateway is responsible for guarding and controlling access to services on the mesh. You can configure the ingress gateway to allow only a specific type of traffic such as SFTP on port 115, which blocks incoming traffic on any other port and of any other type. Similarly, the egress gateway is responsible for routing the traffic out of the mesh. The egress gateway helps you control the external services to which the service on the mesh can connect. Apart from security, this also helps operators monitor and control the outbound network traffic that originates from the mesh.

Istio architecture

To add services to the Istio service mesh, you only need to deploy an Istio sidecar proxy with every service. As we discussed previously, the Istio sidecar proxy handles the communication between services, which is managed using the Istio control plane. Figure 5 shows the high-level architecture of the Istio service mesh, in which we can see how Istio fills out the service mesh architecture with its own components. We will study the architecture of Istio in the context of how its deployment looks in a Kubernetes cluster.

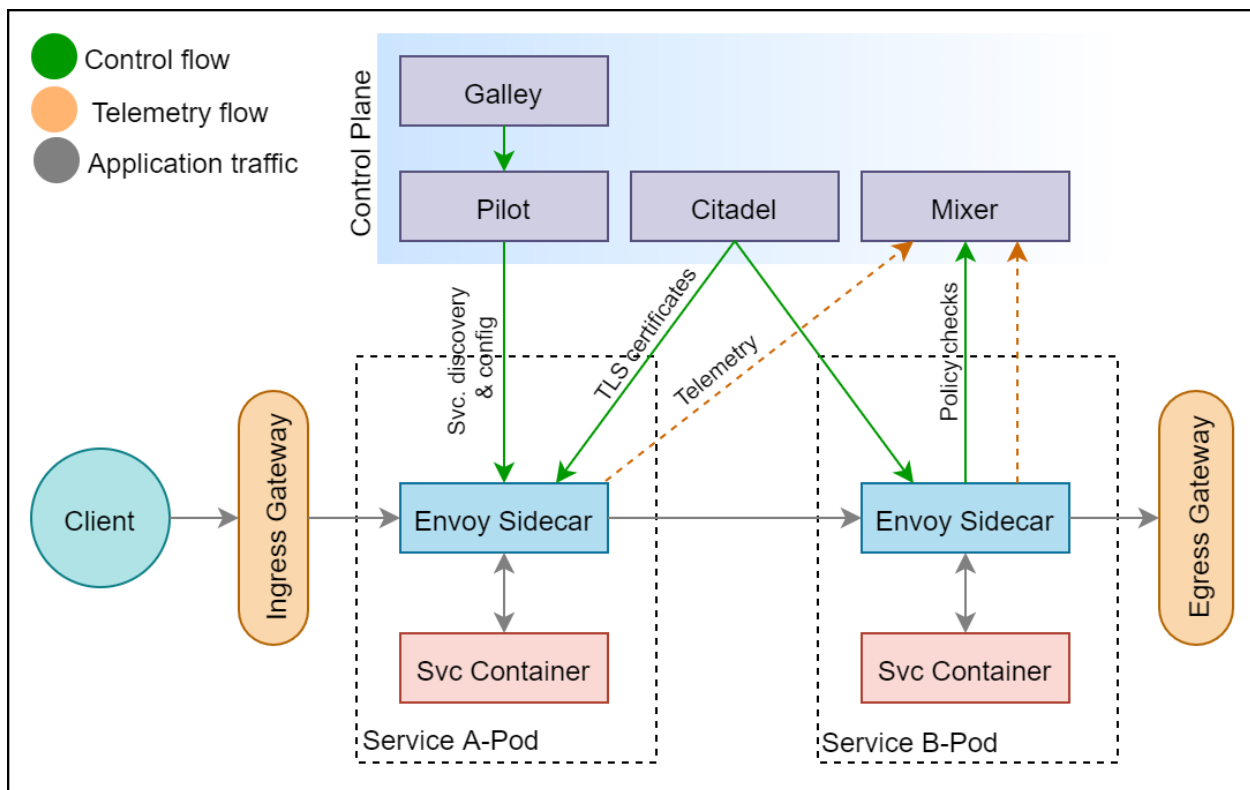


Figure 5: Istio architecture

As you can see in the previous diagram, the Istio proxy is deployed as a sidecar in the same Kubernetes pod as the service. The proxies thus live within the same namespace as your services, and they act as a conduit for inter-service communication inside the mesh. The control plane of Istio is made up of the Galley, Pilot, Citadel, and Mixer. These components live outside your application in a separate namespace that you will provision only for the Istio control plane services. Let us now discuss each component and its role in Istio.

Ingress/egress gateway

The first and last components that interact with the network traffic are the gateways. By default, in a Kubernetes cluster with Istio mesh enabled, services can accept traffic originating from within the cluster. To expose the services on the mesh to external traffic, Kubernetes natively supports an ingress controller named Kubernetes Ingress, amongst other options, which provides fundamental Layer 7 traffic management capabilities such as SSL termination and name-based binding to virtual hosts (for example, route traffic requested with hostname `foo.bar.com` to service 1).

Istio created its own ingress and egress gateways primarily for two reasons: first, to avoid duplicate routing configurations in the cluster, one for ingress and another for Istio proxy, both of which only route traffic to a service; second, to provide advanced ingress capabilities such as distributed tracing, policy checking, and advanced routing rules.

Istio gateways only support configuring routing rules at Level 4 to Level 6 of the network stack, such as specifying routes based on port, host, TLS key, and certificates, which makes it simpler to configure than Kubernetes Ingress. Istio supports a resource named virtual service that instructs the ingress gateway on where to route the requests that were allowed in the mesh by it.

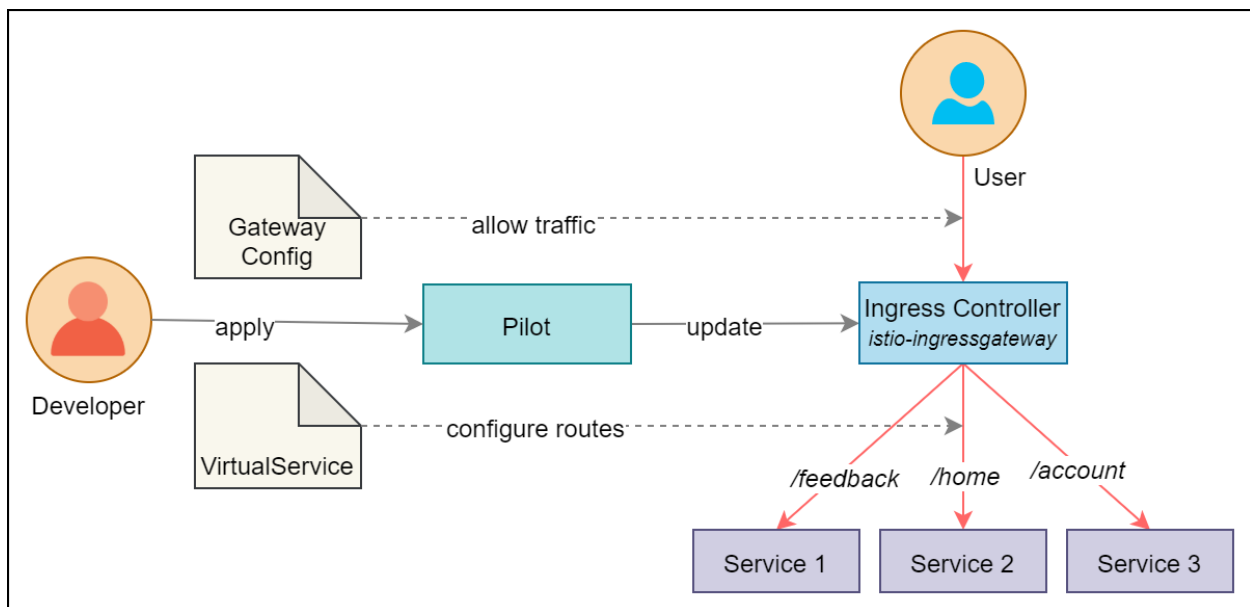


Figure 6: Istio ingress gateway

As shown in the previous diagram, the virtual service specifies routing rules, such as requests to path `/feedback` should be routed to service 1, and so on. By combining the Istio ingress gateway with virtual service, the same rules that are applied to external traffic for allowing external traffic inside the mesh can be used to control network traffic inside the mesh. For example, for the previous setup, service 2 can reach service 1 at the `/feedback` endpoint, and external users can reach service 1 at the same path `/feedback`.

Istio proxy

Istio uses a modified version of the Envoy proxy as the gateway. Envoy is a C++-based Layer 4 and Layer 7 proxy that was initially built by the popular ridesharing company Lyft, and is now a [Cloud Native Computing Foundation](#) (CNCF) project. The sidecar proxy makes up the data plane of Istio. Istio utilizes several built-in features of Envoy, such as the following, to proxy the traffic intended to reach a service:

- Dynamic service discovery with support for failover if the requested service is not healthy.
- Advanced traffic routing and splitting controls such as percentage-based traffic split, and fault injection for testing.
- Built-in support for application networking patterns such as circuit breaker, timeout, and retry.
- Support for proxying HTTP/2 and [gRPC](#) protocols, both upstream and downstream. With this capability, Envoy can receive HTTP/1.1 connections and convert them to HTTP/2 connections in either direction of application or user.
- Raising of observable signals that are captured by the control plane to support the observability of the system.
- Support for applying live configuration updates without dropping connections. The envoy achieves this by driving the configuration updates through an API, hot loading a new process with a new configuration, and dropping the old one.

As you can see in Figure 6, the Istio proxy or Envoy is deployed as a sidecar alongside your services to take care of ingress and egress network communication of your service. Services on the mesh remain unaware of the presence of the data plane, which acts on behalf of the service to add resilience to the distributed system.

Pilot

Pilot is one of the components in the control plane of Istio whose role is to interact with the underlying platform's service discovery system and translate the configuration and endpoint information to a format understood by Envoy or Istio service proxy. Envoy is internally configured using the following discovery services (collectively named xDS APIs):

- **Listener (LDS):** This service governs the port that Envoy should listen to and the filter conditions on the traffic that arrives on that port, such as protocols.
- **Route (RDS):** This service identifies the cluster traffic should be sent to, depending on request attributes such as HTTP header.

- **Cluster (CDS):** A service can be hosted on multiple hosts. This service governs how to communicate with the group of endpoints of a service, such as the certificate to use and the load-balancing strategy to apply.
- **Endpoint (EDS):** This service helps Envoy interact with a single endpoint of a service.

Pilot is responsible for consuming service discovery data from the underlying platform such as Kubernetes, Consul, or Eureka. It combines the data with Istio configurations applied by the developers and operations and builds a model of the mesh.

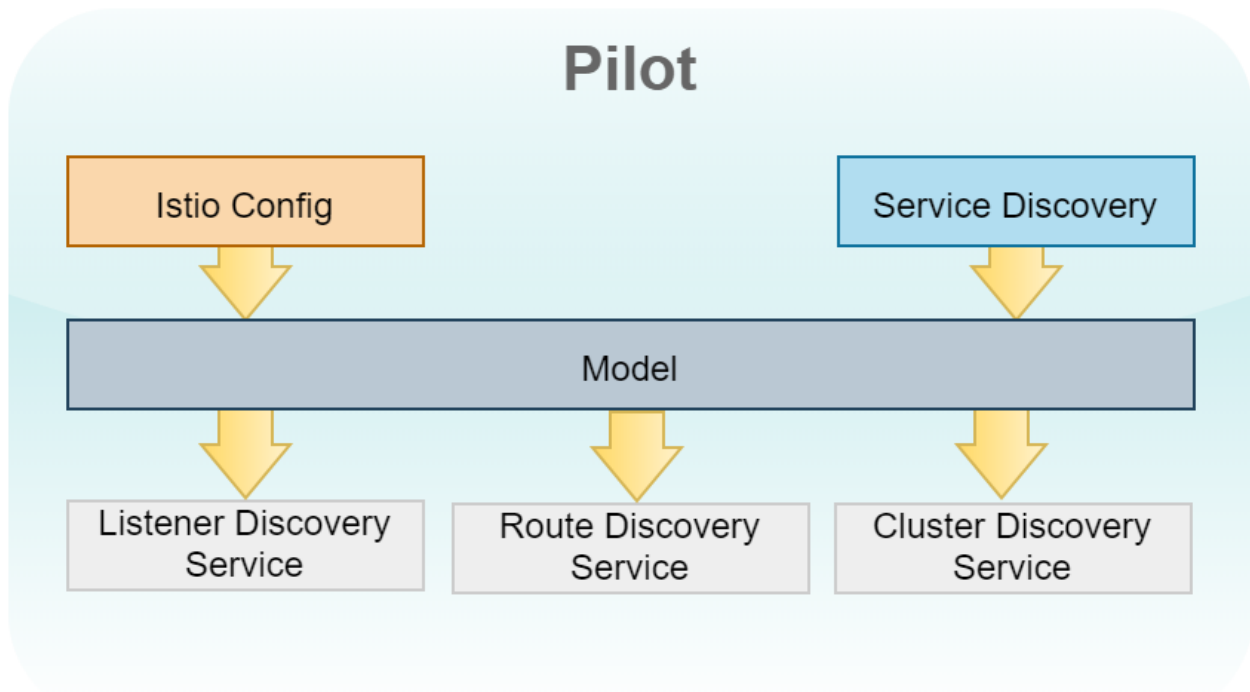


Figure 7: Pilot architecture

Pilot then translates the model to xDS configuration, which is consumed by Envoy services by maintaining a gRPC connection with Pilot and receiving data pushed by Pilot. The distinction between the model and xDS configuration helps Pilot maintain loose coupling between Pilot and Envoy.

Currently, Pilot has intrinsic knowledge of the underlying host platform. Eventually, another control plane component named Galley will take over the responsibility of interfacing with the platform. This shift of responsibility will leave Pilot with the responsibility of serving proxy configurations to the data plane.

Galley

Galley is responsible for ingesting, processing, and distributing user configurations, and it forms the management plane of the Istio architecture. In the near future, Galley will act as the only interface between the rest of the components of Istio and the underlying platform (such as Kubernetes and virtual machines). In a Kubernetes host, Galley interacts directly with the Kubernetes API server, which is the front-end of the Kubernetes cluster state **etcd**, to ingest and validate user-supplied configuration and to store it. Galley ultimately makes the configurations available to Pilot and Mixer using the Mesh Configuration Protocol (MCP).

In a nutshell, MCP helps establish a pub-sub (publisher-subscriber) messaging system using the gRPC communication protocol and requires a system to implement three models:

- **Source:** This is the provider of the configuration, which in Istio is Galley.
- **Sink:** This is the consumer of the configuration, which in Istio are the Pilot and Mixer components.
- **Resource:** This is the source of the resource that the sink pays attention to. In Istio, this is the configuration that Pilot and Mixer are interested in.

After the source and sinks are set up, the source can push changes to resource to the sinks. The sinks can accept or reject the changes by returning an ACK or NACK signal to the source.

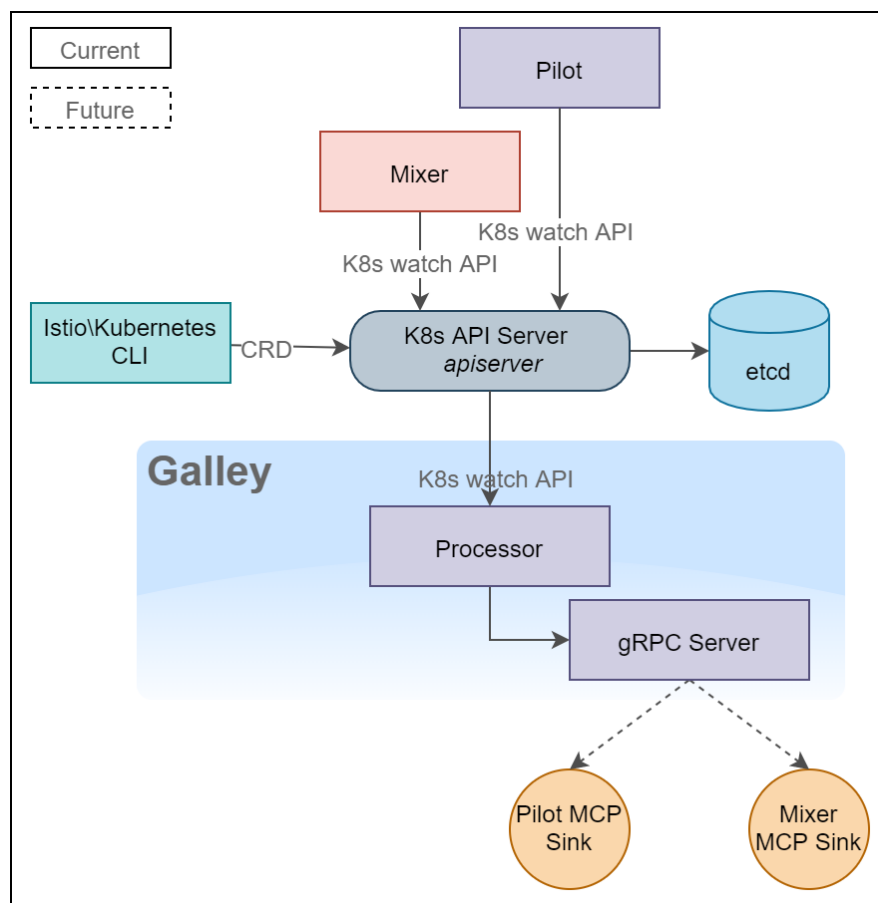


Figure 8: Galley architecture

The previous diagram depicts the dependencies between Galley, Pilot, and Mixer. Pilot, Mixer, and Galley detect changes in the cluster by long polling the Kubernetes watch API. When the cluster changes, the processor component of Galley generates events and distributes them to Mixer and Pilot via a gRPC server. Since Galley is still under development, Pilot and Mixer currently use adapters to connect to the Kubernetes API server. The interaction with the underlying host will be managed only by Galley in the future.

Mixer

Mixer directly interacts with the underlying infrastructure (integration to be replaced with Galley) to perform three critical functions in Istio:

- **Precondition checking:** Before responding to a request, Istio proxy interacts with Mixer to verify whether the request meets configured criteria such as authentication, whitelist, ACL checks, and so on.
- **Quota management:** Mixer controls quota management policies for services to avoid contention on a limited resource. These policies can be configured on several dimensions, such as service name and request headers.
- **Telemetry aggregation:** Mixer is responsible for aggregating telemetry from the data plane and gateways. In the future, Mixer will support aggregating tracing and billing data streams as well.

The following high-level design diagram presents how the various components of Istio interact with Mixer.

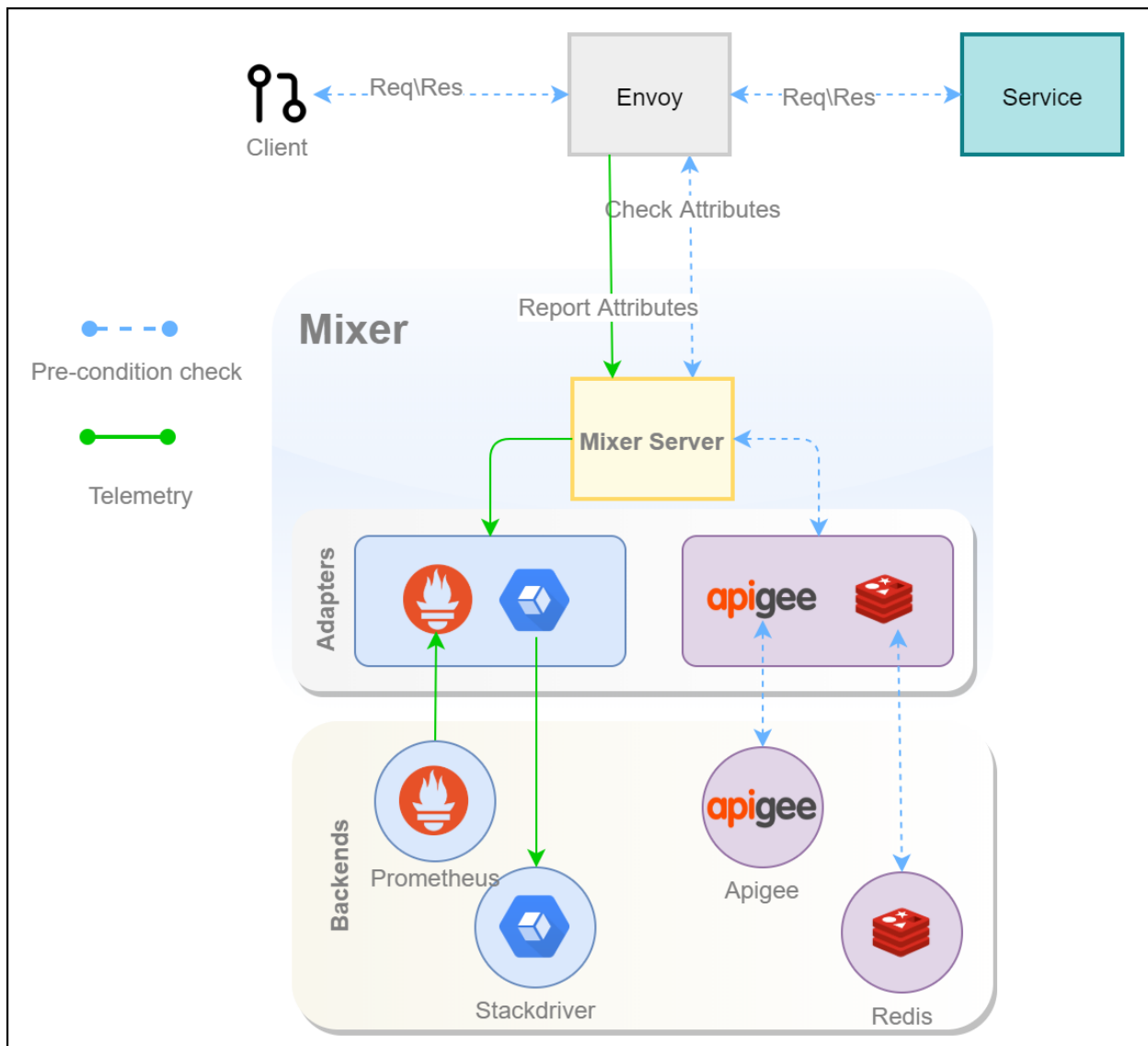


Figure 9: Mixer architecture

Let's discuss the life cycle of the two types of data flows in Istio that involve Mixer: precondition checks (including quota management), and telemetry aggregation. Service proxies and gateways invoke Mixer to perform request precondition checks to determine whether a request should be allowed to proceed based on policies specified in Mixer such as quota, authentication, and so on. Some of the popular adapters for precondition checks are Redis for request quota management and Apigee for authentication and request quota management.



Note: Istio release 1.4 has added experimental features to move precondition checks such as RBAC and telemetry from Mixer to Envoy. The Istio team aims to migrate all the features of Mixer to Envoy in the year 2020, after which Mixer will be retired.

Ingress/egress gateways and Istio proxy report telemetry once a request has completed. Mixer has a modular interface to support a variety of infrastructure backends and telemetry backends through a set of native and third-party adapters. Popular Mixer infrastructure backends include AWS and GCP. Some telemetry backends commonly used are Prometheus and Stackdriver. For precondition checks, adapters are used to apply configurations to Mixer, whereas for telemetry, an adapter configuration determines which telemetry is sent to which backend. Once the telemetry is received by a backend, a supported GUI application such as Grafana can be used to display the information persisted by the backend. Note that in Figure 9, the Prometheus connector points from backend to adapter, because Prometheus pulls telemetry from workloads.

Citadel

To ascertain the security of the service mesh, Istio supports encrypting the network traffic inside the mesh using mTLS (Mutual Transport Layer Security). Apart from being an encryption scheme, mTLS ensures that both the client and the server verify the certificate to ensure that only authorized systems are communicating with each other.

Citadel plays a crucial role in Istio security architecture by maintaining keys and certificates that are used for authorizing service-to-service and user-to-service communication channels. Citadel is entirely policy-driven, so the developers and operators can specify policies to secure the services that allow only TLS traffic to reach them.

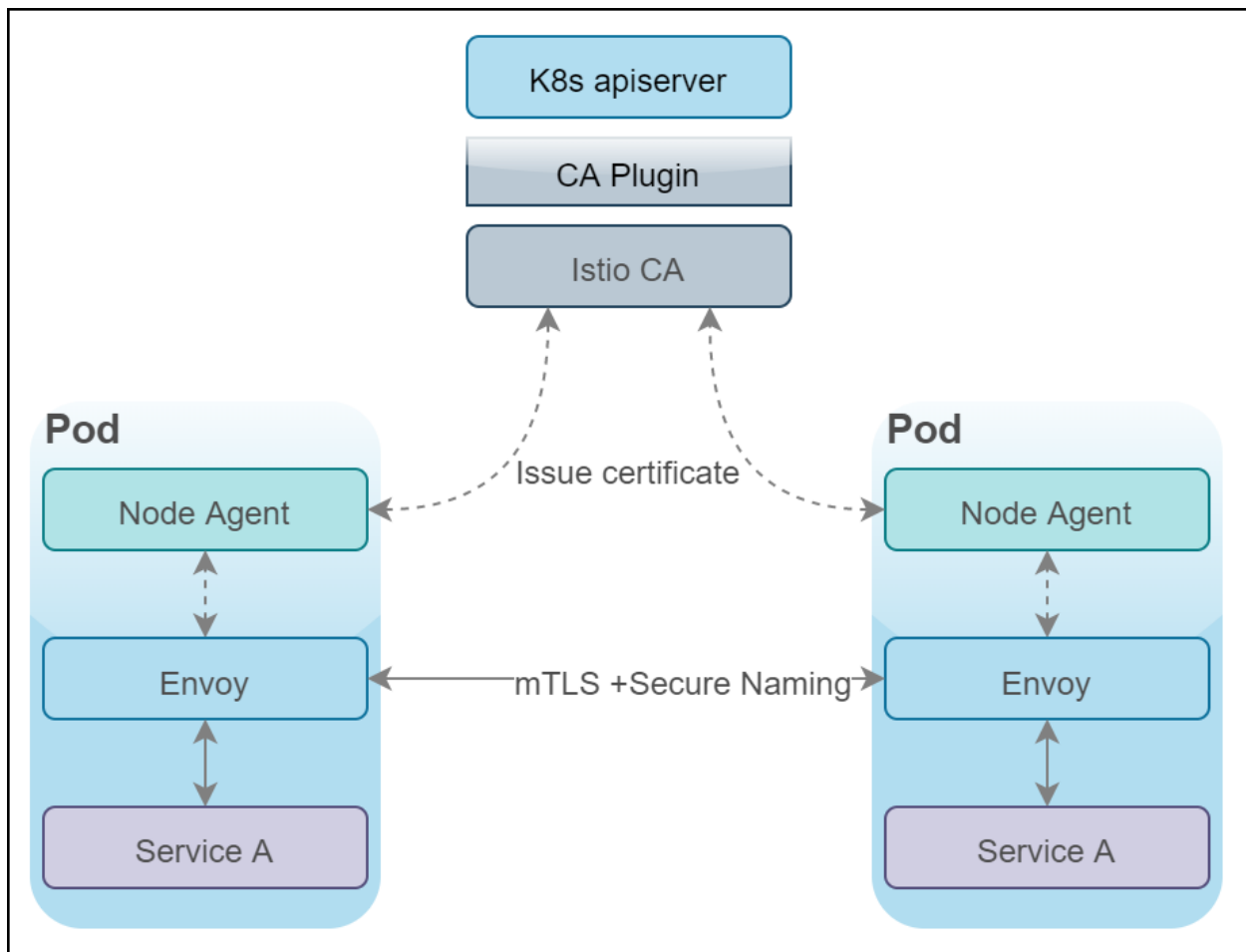


Figure 10: Citadel architecture

Citadel is responsible for issuing and rotating X.509 certificates used for mutual TLS. There are two major components of Citadel. The first one is the Citadel agent, which in the case of Kubernetes host, is called the node agent. The Node agent is deployed on every Kubernetes node, and it fetches a Citadel-signed certificate for Envoy on request. The second component is the certificate authority (CA), which is responsible for approving and signing certificate signing requests (CSRs) sent by Citadel agents. This component performs key and certificate generation, deployment, rotation, and revocation.

In a nutshell, this is how the identity provisioning flow works. A node agent is provisioned on all nodes of the cluster, which is always connected to Citadel using a gRPC channel. Envoy sends a request for a certificate and a key using the secret discovery service (SDS) API to the node agent. The node agent creates a private key and a CSR, and sends the CSR to Citadel for signing. Citadel validates the CSR and sends a signed certificate to the node agent, which in turn is passed on to Envoy. The node agent repeats the CSR process periodically to rotate the certificate.

Citadel has a pluggable architecture, and therefore, rather than issuing a self-signed certificate, it can be configured to supply a certificate from a known certificate authority (CA). With a known CA, administrators can integrate the organization's existing public key infrastructure (PKI) system with Istio. Also, by using the same CA, the communication between Istio services and non-Istio legacy services can be secured, as both types of services share the same root of trust.

The application security policies using Citadel vary with the underlying host. In Kubernetes-hosted Istio, the following mechanism is followed: When you create a service, Citadel receives an update from the **apiserver**. It then proceeds to create a new [SPIFFE \(Secure Production Identity Framework For Everyone\)](#) certificate and key pair for the new service and stores the certificate and key pairs as Kubernetes secrets. Next, when a pod for the service is created, Kubernetes mounts the certificate and key pair to the pod in the form of a Kubernetes secret volume. Citadel continues to monitor the lifetime of each certificate and automatically rotates the certificates by rewriting the Kubernetes secrets, which get automatically passed on to the pod.

A higher level of security can be implemented by using a security feature called secure naming. Using secure naming, execution authorization policies such as *service A is authorized to interact with service B* can be created. The secure naming policies are encoded in certificates and transmitted by Pilot to Envoy. For this activity, Pilot watches the **apiserver** for new services, generates secure naming information in the form of certificates, and defines what service account or accounts can execute a specific service. Pilot then passes the secure naming certificate to the sidecar Envoy, which ultimately enforces it.

Summary

In this chapter, we discussed the need for the service mesh to abstract network concerns from applications. We covered the overall architecture and the components that make up the data plane, control plane, and management plane of Istio. Istio is a highly customizable service mesh that allows developers and operators to apply policies to each component of Istio, and thus control its behavior. In the next chapter, we will light up Istio on a cluster on your machine.

Chapter 2 Installation

In the previous chapter, we learned the architecture of Istio in detail. Now it's time to get our hands dirty and set up Istio on a Kubernetes cluster. As we said earlier, we will deploy Istio to a Kubernetes cluster for working through the exercises in this book. However, you can also deploy Istio on [Nomad](#) or [Consul](#) clusters, which would require using a different set of commands to configure Istio and manipulate objects on them. The core components of Istio remain the same regardless of the host. Istio setup and installation is quite straightforward, and you can do it multiple ways. However, there are a few prerequisites that you must validate before installing Istio. In this chapter, we will discuss the process of installing Istio on a Kubernetes cluster using several different approaches.

Prerequisites

Before installing Istio on a Kubernetes cluster, we must make sure the services that you want to bring on the mesh satisfy the following conditions.:

- Names of the ports of services should follow the syntax `<protocol>-<name>`, such as `http-serviceA`, `https-serviceB`, and `grpc-serviceC`. This convention helps Istio detect the target ports on which it can enforce traffic routing policies.



Note: *The convention-based naming strategy is getting replaced with intelligent protocol detection strategy (an experimental feature in release 1.3) that will automatically determine the protocol of outbound traffic as HTTP or TCP, with other protocols and port detection for inbound traffic to follow in the future.*

- Container ports used inside a pod should be explicitly specified in the specification.
- Each deployment (known as a *workload* in Istio) should be part of at least one service so that the pods of the deployment can receive request traffic.
- All deployments should have app and version labels for configuring features such as canary deployments.
- The pod must also allow NET_Admin capability since the Envoy proxy diverts the incoming and outgoing traffic by manipulating iptables rules, which require this capability.
- Pods should not run applications with universally unique identifier (UUID) 1337, as this user ID is reserved for use by Istio.
- The following table lists the ports used by the Istio services. Don't use these ports within the service mesh for individual services.

Table 1: Ports used by Istio

Port	Protocol
8060, 8080, 9090, 9091, 9876, 15004, 15010, 15014, 15020, 15029, 15031, 15032, 15090	HTTP

15000, 15001, 15006, 15011, 42422	TCP
9901	gRPC
15443	TLS

To ensure that Istio runs smoothly on your Kubernetes cluster, allocate at least four cores and 5–8 GB of RAM to your cluster for running Istio services. This number may vary based on the number of components of Istio that you provision during setup. For local development, [Docker Desktop for Mac or Windows](#) is the easiest and fastest way to get up and running.



Tip: You can refer to [this link](#) for detailed instructions on installation and troubleshooting.

There are many approaches to installing Istio on the Kubernetes cluster, but the simplest and most popular one is to use Helm. Let's begin with downloading the latest version of Istio by following these steps:

1. Navigate to the [Istio releases webpage](#).
2. Select the appropriate release package based on your platform, such as ***.zip** for Windows or ***.tar.gz** for Linux.
3. Download the latest release version and navigate to the downloaded folder. The folder contains the following artifacts:
 - Platform-specific installation YAML files in the **install** folder.
 - Samples in the **samples** folder.
 - **istioctl** client binary, which is the Istio CLI tool in the **bin** folder. You can use this tool for validating setup, debugging, manual proxy injection, and a host of other activities.
 - The **helm** folder inside the Kubernetes installation contains several files named **values-*.yaml**, which help you select a particular type of installation, such as minimal or full.
4. Based on the version of Istio that you downloaded, add the location of the Istio Helm repository using the following command. Remember to substitute the version number 1.3.2 in the following command with the version that you downloaded.

Code Listing 1: Adding Helm repo

```
$ helm repo add istio.io https://storage.googleapis.com/istio-release/releases/1.3.2/charts/
```

```
"istio.io" has been added to your repositories
```

After adding the repo to your Helm repository, you can proceed to the next step of installing a Helm client. Download and install the appropriate Helm client from the [official Helm repository](#) based on your OS/platform. Make sure the Helm client version that you install is greater than or equal to 2.10.



Tip: You should add the full path of the `istioctl` executable to the `Path` environment variable. This will help you invoke the command from any location.

We will now proceed to install the various components of Istio to our Kubernetes cluster.

Installation

There are two ways of installing Istio on a Kubernetes cluster using the Helm chart. One of the methods involves using the Helm template, and the other method uses the Tiller server for installation.

For now, the recommended method for installing Istio is to use Helm and Tiller so that a Tiller pod in your cluster can manage the deployments and upgrades of Istio objects. You can refer to [this link](#) for guidance on installing Istio with Tiller. However, if you don't want to install Tiller on your cluster, you can choose the other method of using Helm templates for installation, which is the approach that we will use to customize installation of Istio.

Installing Istio with Helm

The Istio objects are just native Kubernetes objects, such as services and endpoints, that require a custom definition to set up. To install such objects, Kubernetes supports custom resource definitions (CRD), which are definitions of objects that are not part of default Kubernetes installation. Istio installs CRDs on Kubernetes to bring up objects such as gateways, virtual services, and destination rules. For certain use cases, you can create your own custom resources in Kubernetes. You can refer to [this link](#) for further details on CRDs.

Istio components should be scoped to a namespace in your cluster. Therefore, create a namespace named **istio-system** for provisioning Istio components using the following command.

Code Listing 2: Creating a namespace

```
$ kubectl create namespace istio-system  
  
namespace/istio-system created
```

To install Istio CRDs, from your terminal, change to the directory within the Istio release package that you downloaded and execute the following command.

Code Listing 3: Installing istio-init CRDs

```
$ helm template install/istio-releases/helm/istio-init --name istio-init  
--namespace istio-system | kubectl apply -f -  
  
NAME:      istio-init
```


LAST DEPLOYED: Tue Sep 24 13:22:14 2019
NAMESPACE: istio-system
STATUS: DEPLOYED

The previous command creates bootstrap resources such as cluster roles and config maps, which are required by Istio CRDs using Helm templates. You can also use a predefined Helm profile to install via Helm and Tiller.

There are many predefined configurations available for installing Istio components in the release directory at the path **install/Kubernetes/helm/istio**. You can choose one of the configurations or create your own configuration based on your requirements. The following is a brief description of the predefined configurations available in the release package.

Table 2: Istio configurations

Configuration File	Description
values.yaml	This configuration installs all components with default settings. It is the default Istio profile, and it is the recommended configuration for the production environment.
values-istio-demo.yaml	This configuration provisions the minimum components used by the sample applications available inside the release package. It enables tracing and logging for almost all components, and it is generally enough for simple scenarios that might require debugging.
values-istio-demo-auth.yaml	This configuration is similar to the istio-demo configuration, but also has authentication enabled.
values-istio-minimal.yaml	This is the minimal Istio configuration required for Istio to run.
values-istio-remote.yaml	This configuration creates the components used for managing Istio in a remote cluster, which is useful in a multi-cluster setup.
values-istio-sds-auth.yaml	This configuration is similar to the default configuration, but also has the Secret Discovery Service (SDS) feature enabled.

Most of the preconfigured templates install several components that we will not use for the exercises in this book. Therefore, we will customize the Istio Helm template to install just the components we need. Execute the following command, which will install the necessary components in the cluster.

Code Listing 4: Installing Istio

```
$ helm template install/kubernetes/helm/istio --name istio --namespace istio-system --set global.disablePolicyChecks=false --set gateways.istio-egressgateway.enabled=true --set global.proxy.accessLogFile="/dev/stdout" --set tracing.enabled=true --set kiali.enabled=true --set grafana.enabled=true | kubectl apply -f -
```

```
NAME: istio
LAST DEPLOYED: Tue Sep 24 13:23:36 2019
NAMESPACE: istio-system
NOTES:
Thank you for installing istio.
Your release is named istio.
```

The output of the previous command will inform you of the status of the installation of Istio components.



Tip: To configure CRDs during setup, instead of creating a new configuration with desired settings, you can add desired configuration options to the `helm install` command as we did previously. You can refer to [this link](#) for a list of available options.

After installing Istio components, you can verify the status of the components created by executing the following command.

Code Listing 5: Verify Istio installation

```
$ kubectl get pods -n istio-system
```

NAME	READY	STATUS	RESTARTS
istio-citadel-5cf47dbf7c-hzsst	1/1	Running	2
istio-cleanup-secrets-1.2.5-8xhvp	0/1	Completed	0
istio-egressgateway-7dd48cbd5-gtl4t	1/1	Running	2
istio-galley-7898b587db-69kqr	1/1	Running	3
istio-ingressgateway-7c6f8fd795-fcc5p	1/1	Running	2

istio-init-crd-10-dwgd2 24h	0/1	Completed	0
istio-init-crd-11-c8p5v 24h	0/1	Completed	0
istio-init-crd-12-hxjbp 24h	0/1	Completed	0
istio-pilot-5c4b6f576b-xkk5h 24h	2/2	Running	4
istio-policy-769664fcf7-bmqlx 24h	2/2	Running	9
istio-security-post-install-1.2.5-4b9n9 24h	0/1	Completed	0
istio-sidecar-injector-677bd5ccc5-wwptm 24h	1/1	Running	3
istio-telemetry-577c6f5b8c-4tjsg 24h	2/2	Running	9
prometheus-776fdf7479-d5lrg 24h	1/1	Running	2

In the output generated from the previous command, you should ensure that all pods are in running state except the initialization, cleanup, and post-install security setup pods, which should be in the completed state.

Summary

In this chapter, we went through the steps of setting up Istio on a local cluster, which will act as a debugging environment for us for the subsequent exercises. We also discussed some tips on how you can set up Istio on a multi-node cluster, which will act as staging or production environments for your applications. With Istio up and running on our cluster, we will now proceed to deploy a very simple application to the mesh.

Chapter 3 Envoy Proxy

Now that our cluster is up and running, we are ready to dig deep into the nuances of Istio. The capabilities of Istio can be broadly classified into four key areas:

- Traffic management
- Security
- Observability/telemetry
- Policy enforcement for resiliency and testing

To showcase the value proposition of Istio, we will deploy a simple application named Micro Shake Factory, which simulates a shop that serves fresh fruit shakes on demand. The application consists of two microservices (hence the name) that can communicate with each other over HTTP. We will incrementally deploy the microservices of this application to our service mesh. Since we are going to work with the Micro Shake Factory application throughout the rest of the book, let us first familiarize ourselves with it.

Micro Shake Factory

The following is a high-level design diagram of the Micro Shake Factory application.

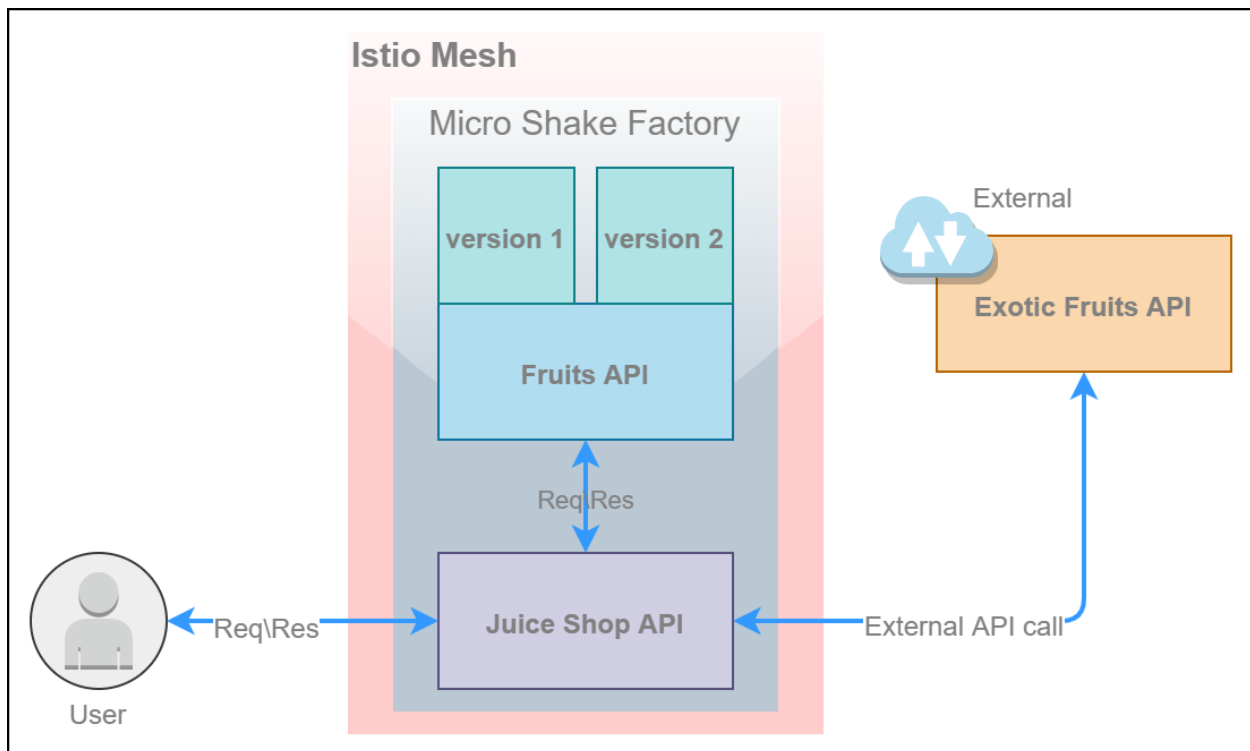


Figure 11: Micro Shake Factory architecture

The capabilities of Istio can be adequately demonstrated with a set of services that can communicate with each other. The Micro Shake Factory application consists of two REST-based microservices with the following capabilities.

- **fruits API:** This API manages the inventory of fruits and their prices by country. This API has the following endpoints.

Table 3: Fruits API endpoints

API Endpoint	Function
GET: /api/fruits/:country	Returns a list of fruits based on the country code provided. The valid country codes are usa , au , and ind .
GET: /api/fruits/special	Returns a specialty fruit that is specific to the version of the API that handles the request. Version 1 of the API returns Mango , and version 2 of the API returns Orange .
GET: /api/fruits/:country/:name/price	Returns the price of a fruit (specified in the parameter name) in a country (specified in the parameter country).

- **juice-shop API:** This API primarily communicates with the **fruits** API to fetch prices of fruits, and determines the price of the juice ordered by the customer. Additionally, this API communicates with an external API to present certain exotic fruits that are not available with the **fruits** API. This API has the following endpoints.

Table 4: Juice-shop API endpoints

API Endpoint	Function
POST: /api/juice-shop/blender	Takes the names of two fruits as input and communicates with the fruits API over HTTP to retrieve the prices of the fruits. This endpoint returns the name and price of the juice prepared from the fruits whose names were sent in the request.
GET: /api/juice-shop/exoticFruits	Fetches a list of fruits from an external API. Currently, this API is modeled as a static document available here .
GET: /api/juice-shop/hello	Returns a greeting message.
GET: /api/juice-shop/testMyLuck	Returns HTTP error status code 500 for around 80 percent of requests and a success response for the remainder of requests.

The microservices are deployed as services on Kubernetes with Linux nodes. To reiterate, Istio is designed to be platform-agnostic and does not require a container orchestrator such as Kubernetes. However, Kubernetes is the best platform supported by Istio.

Deploying services

Let's deploy our first service to the Istio mesh. We will start with deploying the first version of the **fruits-api** microservice to our Kubernetes cluster. As a reminder, you will find all the Kubernetes specification files (in YAML format) for deploying the services in a [repository named Policies](#) in the GitHub account of this ebook.



Note: *YAML is a human-readable data serialization format that is typically used for storing configuration values. YAML is just a key-value store. In the following discussion, we will examine the keys and values it supports for the specification under consideration. If a field is required, it will be marked with an asterisk (*) followed by its data type.*

Let's discuss the contents of the specification in a little detail before applying it to the cluster.

Code Listing 6: Fruits API v1 specification

```
apiVersion: v1
kind: Namespace
metadata:
  name: micro-shake-factory
  labels:
    istio-injection: enabled
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: fruits-api-deployment-v1
  namespace: micro-shake-factory
spec:
  selector:
    matchLabels:
      app: fruits-api
  replicas: 2
  minReadySeconds: 1
  progressDeadlineSeconds: 600
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
  template:
    metadata:
```

```

    labels:
      app: fruits-api
      version: "1"
  spec:
    containers:
      - name: fruits-api
        image: istiosuccinctly/fruits-api:1.0.0
        imagePullPolicy: IfNotPresent
        resources:
          limits:
            cpu: 1000m
            memory: 1024Mi
          requests:
            cpu: 100m
            memory: 100Mi
        ports:
          - name: http-fruits-api
            containerPort: 3000
        env:
          - name: app_version
            value: "1"
    ---
  apiVersion: v1
  kind: Service
  metadata:
    name: fruits-api-service
    namespace: micro-shake-factory
  spec:
    selector:
      app: fruits-api
    ports:
      - name: http-fruits-api-service
        port: 80
        targetPort: http-fruits-api

```

The first Kubernetes object specified is the namespace. Kubernetes objects support having user-defined labels, which are essentially key-value pairs, attached to them. Istio relies on a specific label named **istio-injection** to decide the namespace on which the Envoy proxies are applied. This label supports two values. Setting the value to **enabled** means that Istio automatically deploys sidecars for the pods of your service. On the other hand, setting this value to **disabled** means that Istio will not inject sidecars automatically; however, it won't affect the services within the namespace that have sidecars attached to them.



Note: Specifying the label `istio-injection: enabled` in the namespace will only install sidecars in newly created pods. If you want to inject a sidecar to existing pods, you need to either delete the pods in a controlled manner—which means deleting some pods and waiting for them to come up, then deleting the rest—or you can use

the `istioctl` CLI to perform a rolling update. We will discuss the rolling update option later in this chapter.

Deployment is the second object listed in the specification. A Kubernetes deployment represents an application running on the cluster. The deployment object uses the values specified in the template attribute to define the specification of the pods that it needs to provision. In this case, Kubernetes provisions two pods named **fruits-api** that host our API. Kubernetes deployment uses a selector, which is a core grouping primitive in Kubernetes, to identify the pods that are governed by it.

Finally, the specification contains the definition of a Kubernetes service object. A Kubernetes service is a networking construct that assigns a cluster IP to the underlying pods, making them easy to communicate with, since the pods and their IP addresses are ephemeral.

Istio uses a sidecar injector service that listens to all pod creation events and relies on the **istio-injection** namespace label to decide whether to add a sidecar to a pod. Let's verify the state of this service by executing the following command.

Code Listing 7: Verify sidecar injector deployment

```
$ kubectl -n istio-system get deployment -l istio=sidecar-injector
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
istio-sidecar-injector	1/1	1	1	14d

Since the sidecar injector service is ready, we can now deploy the **fruits-api** microservice to the mesh. Execute the following command to deploy the previous specification to the cluster.

Code Listing 8: Create fruits-api deployment

```
$ kubectl apply -f fruits-api.yml

namespace/micro-shake-factory created
deployment.apps/fruits-api-deployment created
service/fruits-api-service created
```

Now, we can verify that the sidecar was indeed injected by Istio to our pods by executing the following command.

Code Listing 9: Verify fruits-api deployment

```
$ kubectl get pods -n micro-shake-factory -o
jsonpath={.items[*].spec.containers[*].name}

fruits-api istio-proxy fruits-api istio-proxy
```

Since we deployed two instances of our API, the output of this command lists two instances of containers named **fruits-api** and **istio-proxy**.

Ingress gateway

Now that our first service is deployed on the mesh, we will make the service accessible from outside the cluster. We will deploy an ingress gateway that will route traffic originating from outside our cluster to the **fruits-api** service via the Envoy proxy. The following is the specification for the gateway.

Code Listing 10: Ingress gateway rule specification

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: fruits-api-gateway
  namespace: micro-shake-factory
spec:
  selector:
    istio: ingressgateway
  servers:
    - port:
        number: 80
        name: http-fruits-api-gateway
        protocol: HTTP
      hosts:
        - fruits.istio-succinctly.io
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: fruits-api-vservice
  namespace: micro-shake-factory
spec:
  hosts:
    - fruits.istio-succinctly.io
  gateways:
    - fruits-api-gateway
  http:
    - route:
        - destination:
            host: fruits-api-service
            port:
              number: 80
```

The specification in the previous listing creates a gateway object that enables the **fruits-api** service to accept traffic originating outside the cluster. We will read more about the gateway resource in the next chapter. To deploy the specification to the mesh, execute the following command.

Code Listing 11: Create ingress gateway rule

```
$ kubectl apply -f fruits-api-vs-gw.yml
```

```
gateway.networking.istio.io/fruits-api-gateway created
virtualservice.networking.istio.io/fruits-api-vservice created
```

On your local cluster, the gateway that you just created will be available at **hostname-localhost**; otherwise, you can locate the external IP of the ingress service by executing the following command.

Code Listing 12: Get ingress gateway service

```
$ kubectl get service istio-ingressgateway -n istio-system
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP
istio-ingressgateway	LoadBalancer	10.102.77.234	localhost

Let's send a simple HTTP **GET** request to our API to see whether it responds as expected.

Code Listing 13: Invoking fruits-api service

```
$ curl http://localhost/api/fruits/special -H "Host: fruits.istio-succinctly.io"

{"ver":"1","fruit":"Mango"}
```

Although being able to access services outside the cluster makes it easy to visualize the changes applied by Istio policies, the services are not required to be exposed outside the cluster for traffic manipulation. This means that you can affect traffic routing to a service by just using the virtual service object without adding a gateway for it.

Manual sidecar injection

The journey to virtualization in enterprises is an incremental process. This means that in some cases, you might find that organizations want to bring their existing workloads running on Kubernetes to the mesh. In such cases, you will have to manually inject Envoy sidecars into the existing services.

To simulate this scenario, let's deploy another service to our namespace with automatic sidecar injection turned off.

Code Listing 14: Juice-shop API specification

```
apiVersion: v1
kind: Namespace
metadata:
  name: micro-shake-factory
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: juice-shop-api-deployment
```

```

    namespace: micro-shake-factory
spec:
  selector:
    matchLabels:
      app: juice-shop-api
  replicas: 2
  minReadySeconds: 1
  progressDeadlineSeconds: 600
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
  template:
    metadata:
      labels:
        app: juice-shop-api
    spec:
      containers:
        - name: juice-shop-api
          image: istiosuccinctly/juice-shop-api:1.0.0
          imagePullPolicy: IfNotPresent
          resources:
            limits:
              cpu: 1000m
              memory: 1024Mi
            requests:
              cpu: 100m
              memory: 100Mi
          ports:
            - name: http-js-api
              containerPort: 3001
          env:
            - name: FRUITS_API
              value: ""
            - name: EXOTIC_FRUITS_API
              value: ""
      ---
apiVersion: v1
kind: Service
metadata:
  name: juice-shop-api-service
  namespace: micro-shake-factory
spec:
  selector:
    app: juice-shop-api
  ports:
    - name: http-js-api-service
      port: 80
      targetPort: http-js-api

```

You can see in the previous listing that we removed the label **istio-injection** from the namespace. This change will not alter the existing pods in our namespace. However, Istio will no longer inject sidecars to pods that we provision next.

We will now instruct Kubernetes to provision a new deployment for our service named **juice-shop-api** and provision a new service object for facilitating communication with the pods that get created. Execute the following command to deploy the specification to the cluster.

Code Listing 15: Create juice-shop API

```
$ kubectl apply -f juice-shop-api-no-sidecar.yml

namespace/micro-shake-factory configured
deployment.apps/juice-shop-api-deployment created
service/juice-shop-api-service created
```

You can validate whether a sidecar is injected into the new deployment, and whether it affected the existing pods, by enumerating the pods and their containers using the **kubectl get pods** command that we used previously.

We will now use the command-line utility of Istio named **istioctl** to inject a sidecar to our newly provisioned application, **juice-shop-api**, and thus onboard it to the mesh. The **kube-inject** command of **istioctl** can modify a deployment specification to provision sidecars for supported resources and leave the rest, such as secrets and policies, unaffected. Execute the following command, passing in the previous specification as an argument, to view the new specification generated by the **istioctl** CLI.

Code Listing 16: Inject Istio sidecar specification

```
$ istioctl kube-inject -f juice-shop-api-no-sidecar.yml

apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  name: juice-shop-api-deployment
  namespace: micro-shake-factory
spec:
  minReadySeconds: 1
  progressDeadlineSeconds: 600
  replicas: 2
  selector:
    matchLabels:
      app: juice-shop-api
    ...
  spec:
    containers:
      image: docker.io/istio/proxyv2:1.2.3
      imagePullPolicy: IfNotPresent
      name: istio-proxy
```

```
    initContainers:
      image: docker.io/istio/proxy_init:1.2.3
      name: istio-init
status: {}
```

In the previous listing, we can see that now Istio is relying on Kubernetes deployment to provision the sidecar proxy for the **juice-shop** API pods, which is what the sidecar injector service does in case of automatic sidecar injection. The generated specification instructs Kubernetes to provision two additional containers in each replica requested by us. The first container provisioned by Kubernetes in the deployment sequence is the **istio-init** container, which is a special container that must execute to a conclusion before any other containers can be provisioned. In Kubernetes specifications, such containers are specified as the value of the **initContainers** attribute. Next, the **istio-proxy** and **juice-shop-api** containers are provisioned as expected.



Note: The *kube-inject* command is not idempotent. It is essential that you preserve a copy of the original specification in your source control in case you wish to roll back at a later point in time, or wish to upgrade the sidecars to the latest version by applying the *kube-inject* command on the specification again.

To deploy the specification to your cluster, you can use the well-known bash pipes trick as follows.

Code Listing 17: Inject Istio sidecar

```
$ istioctl kube-inject -f juice-shop-api-no-sidecar.yml | kubectl apply -f -
namespace/micro-shake-factory unchanged
deployment.apps/juice-shop-api-deployment configured
service/juice-shop-api-service unchanged
```

Let's execute the same command that we executed previously to view the containers in our pods.

Code Listing 18: Verify juice-shop API deployment

```
$ kubectl get pods -n micro-shake-factory -o
jsonpath={.items[*].spec.containers[*].name}

fruits-api istio-proxy fruits-api istio-proxy juice-shop-api istio-proxy
juice-shop-api istio-proxy
```

Using pipes, you can also update an existing deployment to add a sidecar to it and bring it to the mesh by executing the following command.

Code Listing 19: Inject sidecar specification and apply

```
$ kubectl get deployment -o yaml | istioctl kube-inject -f - | kubectl apply
-f -
```

If you are wondering where the command **kube-inject** is getting data such as the sidecar image name from, the answer is that this data is present in a ConfigMap resource named **istio-sidecar-injector** that lives in the Istio control plane in the **istio-system** namespace.



Note: *istioctl now supports an experimental command that can bring either a service or a VM to the mesh using a single command: `add-to-mesh`. This experimental command can onboard the juice-shop API service to the mesh as well: `istioctl experimental add-to-mesh service juice-shop API service -n micro-shake-factory`.*

By modifying this ConfigMap, you can alter the behavior of Istio to support features such as disabling automatic sidecar proxy injection for some or all namespaces or even pods. You can edit the **istio-sidecar-injector** ConfigMap file by executing the following command.

Code Listing 20: Edit sidecar injector ConfigMap

```
kubectl -n istio-system edit configmap istio-sidecar-injector
```

If you don't want to make overarching changes to Istio, you can create your own configuration file that conforms to the schema of the **istio-sidecar-injector** ConfigMap and supply it as a parameter to the **kube-inject** command using the **--injectConfigFile** parameter. You can also create a custom ConfigMap and supply it as the value of the parameter **--injectConfigMapName** of the **kube-inject** command.

You must be wondering how Istio can detect services getting deployed or updated in the cluster. Kubernetes natively supports an object named admission controller, which can intercept requests to **apiserver** before they are persisted to the **etcd** store. Apart from some built-in controllers, Kubernetes supports admission controllers developed as extensions that execute as webhooks (code that receives HTTP callbacks from Kubernetes). The admission webhooks receive admission requests from Kubernetes, on which they can perform an operation. The admission webhooks are of two types:

- **Validating admission webhook:** They validate the admission request and either accept or fail the request.
- **Mutating admission webhook:** They can modify the objects sent to the **apiserver** to enrich the behavior of the request. Mutating webhooks are invoked before the validating admission webhooks in sequence.

The Istio sidecar injector is added as a mutating admission webhook during the installation of Istio. You can verify its presence in your cluster by executing the following command.

Code Listing 21: Get mutating webhook configurations

```
$ kubectl get mutatingwebhookconfigurations
```

NAME	CREATED AT
istio-sidecar-injector	2019-09-19T11:07:04Z

You can find the **istio-injection** label constraint that triggers this webhook by expanding the configuration of this webhook using the following command. The output from the execution of the command is truncated for brevity.

Code Listing 22: Istio sidecar injector YAML

```
$ kubectl get mutatingwebhookconfigurations istio-sidecar-injector -o yaml

apiVersion: admissionregistration.k8s.io/v1beta1
kind: MutatingWebhookConfiguration
webhooks:
- admissionReviewVersions:
  - v1beta1
  clientConfig:
    caBundle:
    service:
      name: istio-sidecar-injector
      namespace: istio-system
      path: /inject
  failurePolicy: Fail
  name: sidecar-injector.istio.io
  namespaceSelector:
    matchLabels:
      istio-injection: enabled
  rules:
  - apiGroups:
    - ""
    apiVersions:
    - v1
    operations:
    - CREATE
    resources:
    - pods
    scope: '*'
  sideEffects: Unknown
  timeoutSeconds: 30
```

Note the **matchLabels** constraint that must evaluate to **true** to activate the webhook. When trying to create pods that match the label constraint, the **istio-sidecar-injector** service augments your pod with a sidecar.

Let's get back to the **juice-shop** API that we just deployed. The API is accessible inside the mesh, but it is inaccessible to us. To test the API, we will create an ingress gateway and link it to our virtual service to expose it.

Code Listing 23: Ingress gateway rule specification

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
```

```

name: juice-shop-api-gateway
namespace: micro-shake-factory
spec:
  selector:
    istio: ingressgateway
  servers:
    - port:
        number: 80
        name: http-juice-shop-api-gateway
        protocol: HTTP
      hosts:
        - juice-shop.istio-succinctly.io
  ---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: juice-shop-api-vservice
  namespace: micro-shake-factory
spec:
  hosts:
    - juice-shop.istio-succinctly.io
  gateways:
    - juice-shop-api-gateway
  http:
    - route:
        - destination:
            host: juice-shop-api-service
            port:
              number: 80

```

Apply the configuration to the cluster using the **kubectl apply** command.

Code Listing 24: Create ingress gateway rule

```

$ kubectl apply -f juice-shop-api-vs-gw.yml

gateway.networking.istio.io/juice-shop-api-gateway created
virtualservice.networking.istio.io/juice-shop-api-vservice created

```

We can now send a request to our newly deployed service to validate that it works.

Code Listing 25: Call juice-shop API

```

$ curl http://localhost/api/juice-shop/hello -H "Host: juice-shop.istio-succinctly.io"

Welcome to the Juice Shop!

```


The output of the previous command will present a welcome message from the **juice-shop** API service.

Summary

In this chapter, we deployed the services that make up our demo application Micro Shake Factory to the mesh on our Kubernetes cluster. We used both the automatic and manual approach to injecting sidecars to our services. Manual sidecar injection is useful in brownfield scenarios where you want to gradually migrate services to the mesh. In the next chapter, we will discuss the traffic management capabilities of Istio in detail.

Chapter 4 Traffic Management: Part 1

Traffic management is a core capability of any service mesh. To fully understand the traffic management capabilities of Istio, we need to understand the life cycle of a request in an Istio-enabled cluster. Istio supports many networking APIs that enable you to control deployment propagation (canary deployment), set timeouts and retries, and test your application by deliberately injecting controlled faults. The networking APIs are also used to control the flow of traffic in and out of the service mesh.

Life of a network packet

Envoy and Pilot are the key components of Istio that enable service discovery and decide the direction of traffic between hosts. Envoy intercepts all traffic inside the service mesh at runtime using iptables rules or a [Berkeley Packet Filter](#) (BPF) program, which is a small, stateless, in-kernel program that provides user-level network packet capture. Whenever a network packet is received, the BPF program is executed to determine whether the packet should reach a user-space process (a memory area where applications and drivers execute). Envoy uses request attributes such as hostname, Server Name Indication (SNI), or a virtual IP address to determine the service to which the client wants to send the request (target service). Envoy then uses the target service routing rules to determine the actual service to which it sends the request (destination service). Then it applies destination routing rules, such as the load-balancer strategy, to pick an endpoint for the pod. Finally, Envoy sends traffic to the selected destination service endpoint.



Note: *Allowing the application to communicate with a sidecar without TLS encryption gives you the ability to use L7 policies (such as routing traffic based on user-agent, such as browsers and devices) since Istio can collect L7 metadata about requests. Otherwise, you will be limited to L4 policies that work on the contents of the IP packet, the source and destination address, and the port number.*

Pilot understands the topology of the mesh and the networking configurations supplied by the cluster administrator, which helps it build service proxies. Pilot stores the knowledge of the endpoints of all the services in the cluster in its registry. Pilot pushes this information to the Envoy proxy, and Envoy only understands the hostnames made available to it by Pilot. Envoy can only direct traffic to an endpoint inside the mesh by resolving the IP address of the service itself from the hostname (remember, it has access to the service registry), but it can be configured to forward traffic to a service outside the cluster only if the IP address of the target is specified.

Istio also supports client-side load balancing using the various configurable algorithms: round-robin, random, and least connection. Client-side load balancing removes the need to use a reverse proxy load balancer, and thus removes the need for an additional network hop by request.

Let's go through the life cycle of a single data packet across the mesh, which will help us understand the interactions between the various services in the mesh. The following diagram illustrates the path of a single packet across the mesh.

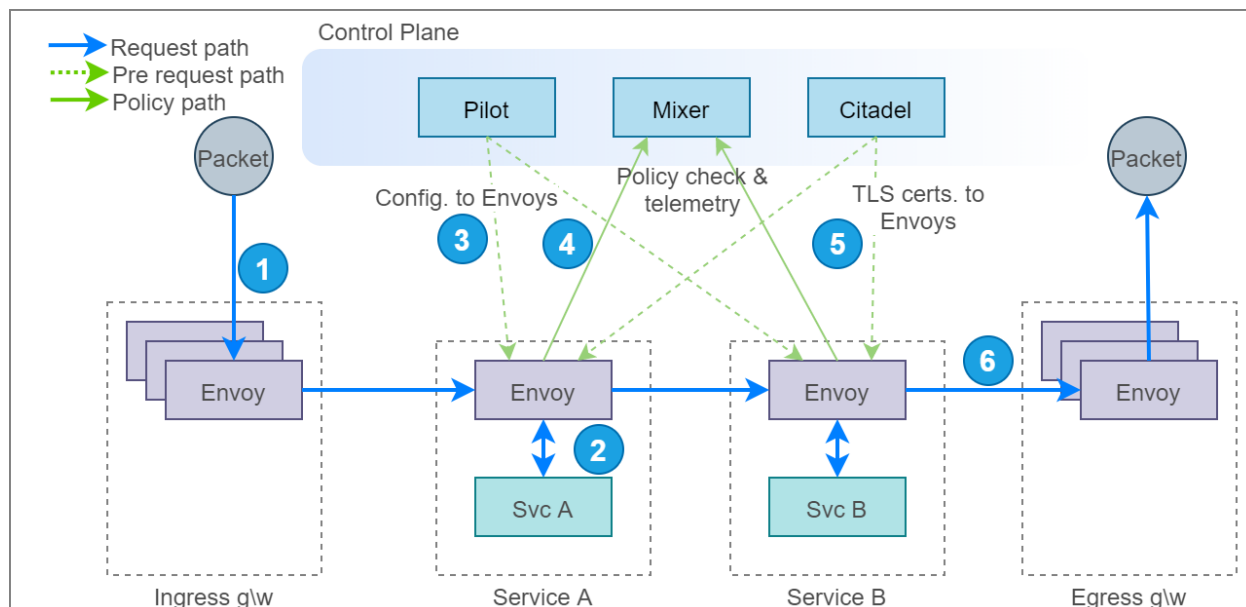


Figure 12: Life cycle of a packet across service mesh

The following steps illustrate the life cycle of a request that originates outside the cluster and moves through services A and B before leaving the mesh.

1. The request meets the ingress gateway, which is a pool of Envoy proxies, which resolves the destination of the packet by making decisions on layer 7 attributes of the request, such as host and path. The ingress directs the packet to another Envoy proxy that is responsible for intercepting requests for the service that should receive the request.
2. The Envoy proxy forwards the request to service A and receives the response. You can configure resilience policies such as timeouts and retries in the proxy so that requests don't fail immediately. Let's assume that the request succeeded; service A now wants to send a request to service B. We now have a challenging problem of resolving IP addresses of pods that are running service B.
3. Kubernetes exposes an API endpoint called **endpoints** that returns a list of endpoints of a service when you provide the name of a service as input. To reduce latency in service discovery, Pilot prefetches the service endpoints and sends them to Envoy instances so that the information necessary to discover services is available locally to them. Going back to our example, Envoy now has several endpoints of service B for forwarding the request.



Note: You can see the in-cluster addresses of the endpoints of the `fruits-api` service that is available to Envoy by executing this command: `kubectl get endpoints/fruits-api-service -n micro-shake-factory`. This is the same data that Pilot receives from the Kubernetes endpoint API. Envoy performs client-side load balancing to route traffic to one of the endpoints.

4. Before forwarding the request to an Envoy instance running service B, the Envoy instance on service A asynchronously sends the request details to the Mixer that maintains policies such as rate limits and whitelists. Mixer determines whether the request may reach service B and sends the appropriate response back to Envoy. Executing the policy on every request is not optimal; therefore, Mixer also sends a cache key that Envoy can use to persist the response that it received from Mixer. The cache stays valid for a fixed number of requests or a duration. For subsequent requests, Envoy queries the cache until it becomes invalid before forwarding the request again to Mixer. Another role of Mixer is to aggregate logs, metrics, and traces, which happens asynchronously with Envoy.
5. The final step before forwarding the request is to apply the right certificate on the request so that the request travels through a secure channel using mutual TLS authentication (mTLS). Citadel is responsible for supplying and rotating service certificates to Envoy that remain valid for a short duration. With the certificates in place, the request is securely transmitted to Envoy's instance of service B.
6. Finally, the request packet reaches the egress gateway, which is essentially the gateway for the services on the mesh to the internet, and it can apply policies such as whitelist and rate limits on the request. The egress gateway accesses the policies from the same set of control plane components as other Envoy instances.
7. The missing bit in this diagram is a component named Galley, which we discussed in [Chapter 1](#). Galley will be the only component that interacts natively with the underlying platform and supply necessary configurations to Pilot and Mixer.

Now that you understand the interactions between the components of Istio, let's dig deep into the Networking APIs of Istio that help you bring traffic management into effect in your service mesh.

Networking API

All of Istio's networking APIs use hostnames resolved by DNS instead of IP addresses to discover services. IP addresses are ephemeral, and can also follow different formats (v4 or v6), depending on the environment. The hostname-centric model is similar to what Kubernetes services use to discover and route traffic to pods. Overall, Istio has six networking APIs that pack all the features you need for shifting networking concerns of applications to the platform:

- **ServiceEntry:** Adds additional entries to the service registry, such as virtual machines outside the cluster, or services outside the mesh that require communicating with the services in the mesh.
- **DestinationRule:** Defines policies such as load balancing and connection pool size that get applied after routing has taken effect.
- **VirtualService:** Defines routing rules that control the behavior of traffic routing to services.
- **Gateway:** Used to configure L4–L6 policies in the form of the load balancer for HTTP/TCP traffic at the edge of the mesh.
- **EnvoyFilter:** Describes a filter for Envoy that can be used to customize the proxy configuration generated by Pilot. This is a very potent filter that can alter the internal working of Istio, and a wrong implementation can cripple the entire cluster. You will hardly find scenarios that require manipulating Envoy filters, and therefore, it is outside the scope of this book.
- **Sidecar:** Describes configuration of the sidecar proxy (remember, Envoy is just an implementation of sidecar). By default, Istio allows all traffic to a workload to go through

a proxy and the proxy to reach all instances in the mesh. You can use this resource to restrict access of the proxy to a set of services and to restrict the proxy to accept and forward traffic received on a set of ports and protocols. The sidecar networking API is outside the scope of this book.

Let's briefly discuss how the various APIs work together to logically drive a request to its target. Assume that a client wants to access version 1 of service A by sending a request to hostname **foo.com**. The following diagram presents the logical path that the request takes to reach the destination service both from outside the cluster and inside the cluster.

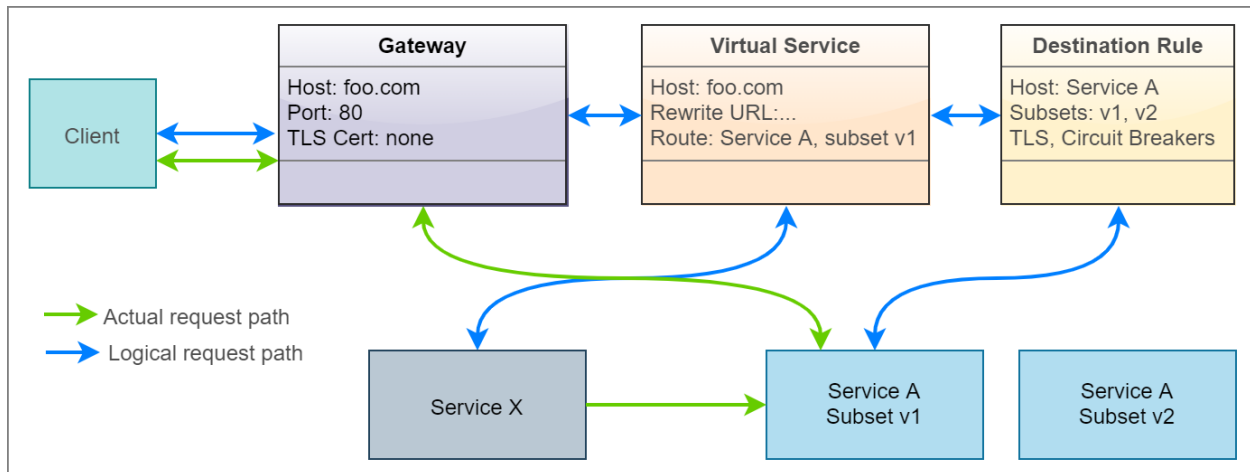


Figure 13: Actual and logical flow of request

The requests originating outside the cluster first meet the ingress gateway, which is configured to only accept HTTP requests to hostname **foo.com** on port 80. A virtual service is bound to the gateway so that it receives any request received by the gateway. For services that communicate within the mesh, the gateway component is absent, and therefore, the request is received directly by the virtual service, which then rewrites the input URL of the service to the URL of service A within the cluster (for example, **foo.com** to **serviceA.svc.default.cluster.local**). A destination rule defines the subsets of service A, (for example, version 1 and version 2), and the virtual service directs the requests to one of the subsets based on match conditions that you can specify on layer 7 request attributes.

Remember that the gateway, virtual service, and destination rule are just programming constructs that execute within the Envoy proxy, and therefore, physically, the request only travels from the ingress gateway Envoy instance to the destination service Envoy instance, from where it reaches the destination service.

Service entry

Using service entry, you can add or remove a service listing from the service registry of Istio. Once defined, the service names become well known to Istio, and they can then be mapped to other Istio configurations. By default, Istio allows services to send requests to hostnames that are not present in the service registry. To reduce the attack surface area of the application, this setting should be inverted, and only legitimate services outside the cluster should be whitelisted by creating service entries for them.



Tip: The `istioctl` utility has several useful commands that can help verify a policy before you end up applying it. The command `istioctl validate -f filename` can verify the syntax of the specification. The command `istioctl verify-install -f filename` can verify whether the resources specified in the specification are in a ready state.

The following are the keys of the service entry object in YAML format.

Code Listing 26: Service entry specification

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
spec:
  hosts:
  addresses:
  ports:
  - number:
    name:
    protocol:
  resolution: STATIC | DNS | NONE
  location: MESH_INTERNAL | MESH_EXTERNAL
  endpoints:
  - address:
    locality:
    labels:
      app:
      version:
    ports:
      <portname>: <portnumber>
  exportTo:
  subjectAltNames:
```

Creating a service entry does not create a DNS record in Kubernetes, so you won't be able to perform a DNS lookup from the application to fetch the resolved IP address. A core DNS Istio plugin can generate DNS records from service entry records, which enables Istio to populate DNS records outside Istio. Let's go through the various keys in the service entry definition and the values that they support.

- **hosts*: string[]**
DNS name with wildcard prefix, such as `*.foo.com`. For HTTP traffic, the HTTP `host/Authority` header will be matched against the specified value. For HTTPS/TLS

traffic, the SNI value will be compared against the specified value. If resolution is set to DNS and endpoints are empty, the specified value will be used as a DNS name to which traffic is sent.

- **ports*: port[]**
Ports associated with the external service. Every port specification requires a name, port number, and protocol name.
- **addresses: string[]**
The virtual IP address of the service whose name is specified in the **hosts** field to enable services in the mesh to send requests to an IP address. The value can also be in the CIDR format (IP address/decimal number). The destination of the request is identified by matching the IP address to one of the IP addresses or CIDR range; otherwise, the request is mapped by using the port specified in the request. In this case, no other service in the mesh should be using the same port.
- **location: location**
 - **MESH_INTERNAL**: Indicates that a service is part of the mesh. It is used for services inside the cluster but not part of the mesh, such as virtual machines and databases.
 - **MESH_EXTERNAL**: Indicates that a service is external to the mesh. It is used for external services that are consumed through APIs. Istio will not use mTLS to communicate with the service.
- **resolution*: resolution**
The value determines how the proxy will resolve the IP address of the network endpoint of the destination service, so that it can route the request to the service.
 - **NONE**: Assumes that the destination IP address has already been resolved. This doesn't trigger lookup, and the connection will be forwarded to the target.
 - **DNS**: The proxy will attempt to discover the IP address using the host DNS (in the case of Kubernetes, CoreDNS). If no endpoints are specified and no wildcards are used in hosts, proxy will use the values in the host's field to look up the DNS. If endpoints are specified, the address specified will be used for the DNS lookup.
 - **STATIC**: The proxy will use the static IP address specified in the endpoints/address field.
- **endpoints: endpoint[]**
An endpoint defines an IP address or a hostname associated with the service. It consists of the following fields:
 - **address***: IP address without the port or a hostname, which can be specified if the resolution is set to **DNS**. It also supports Unix domain socket endpoints.
 - **ports**: Denotes a set of ports associated with an endpoint in the format **map<string,uint32>**, where the key is the name of the port, and the value is the number of the port.
 - **labels**: Denotes labels associated with the endpoint in the form of **map<string,string>**, where the key is the name of the label, and the value is the value of the label.
 - **network**: Instructs Istio to group endpoints in the same network as specified in the string value of this field. All endpoints in the same network can communicate with each other.
 - **locality**: A string value that denotes the fault domain that the endpoint belongs to. In general, endpoints sharing the same fault domain share the same physical rack in a data center. Sidecar proxies prefer the endpoints that are in the same fault domain as them, as it guarantees the least latency.

- **weight:** Denotes the load balancer weightage for the endpoint. The endpoint with higher weightage receives more traffic.
- **exportTo: string[]**
This field contains the names of namespaces in which the sidecars, gateways, and virtual services can use this service. It is a mechanism for cluster administrators to restrict the visibility of services across network boundaries. Currently, this field only supports `.` and `*` as values to denote the current namespace and all namespaces.
- **subjectAltNames: string[]**
This is a security feature that helps Envoy verify whether the server certificate's subject alternate name matches one of the names specified as a value of this field.

Service entries are not only useful for making external services visible to the mesh; they are also useful for making services that are not part of the mesh visible to it.



Tip: DNS resolution capability in Istio is provided by the host platform. In Kubernetes, the *kube-dns* service (which uses CoreDNS) is responsible for resolving the fully qualified domain name (FQDN). You can test DNS resolution by installing BusyBox (a small utility that gives you a shell) in your cluster and then resolving the name of an external service and an internal service as follows.

```
$ kubectl apply -f https://k8s.io/examples/admin/dns/busybox.yaml
$ kubectl exec -ti busybox -- nslookup kubernetes.io
$ kubectl exec -ti busybox -- nslookup fruits-api-service.micro-shake-
factory.svc.cluster.local
```

Let's investigate the specifications for some common service entry scenarios. Note that the following listings contain only the value of the **spec** key for brevity.

1. Assign a hostname to an IP address so that services in the mesh can use a hostname to communicate with an external resource. The following specification assigns **hostname-host.domain.com** to IP address **1.2.3.4**.

Code Listing 27: Specification to assign a hostname to an IP address

```
spec:
  hosts:
  - host.domain.com
  ports:
  - number: 80
    name: http
    protocol: http
  resolution: STATIC
  endpoints:
  - address: 1.2.3.4
```

2. Translate one hostname or IP address (a virtual IP address or VIP) to another name so that the target hostname is abstracted from the workload on the mesh. The following specification forwards requests to either hostname **service1.micro-shake-factory.com** or IP address **1.2.3.4** to **service2.com**.

Code Listing 28: Specification to forward request to an IP address or hostname

```
spec:
  hosts:
    - service1.micro-shake-factory.com
  addresses:
    - 1.2.3.4
  location: MESH_EXTERNAL
  ports:
    - number: 443
      name: https
      protocol: HTTP
  resolution: DNS
  endpoints:
    - address: service2.com
```

3. Make a service on the Kubernetes cluster addressable by Istio. This helps services on the mesh communicate with other services in the cluster in the same manner as they communicate with other services on the mesh. The following specification will configure Istio to forward any requests to hostname **service2** to reach **service1** on the Kubernetes cluster.

Code Listing 29: Specification to make non-Istio service addressable

```
# Kubernetes spec for creating a service and its endpoint
apiVersion: v1
kind: Service
metadata:
  name: service1
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
---
apiVersion: v1
kind: Endpoints
metadata:
  name: service1
subsets:
  - addresses:
      - ip: 1.2.3.4
    ports:
      - port: 80
---
# Partial Istio spec for making service1 accessible to services on mesh
spec:
  hosts:
    - service2
  ports:
    - number: 80
```

```
name: http
protocol: HTTP
resolution: STATIC
endpoints:
- address: 1.2.3.4
```

Our **juice-shop** API service communicates with an external service to fetch exotic fruits for buyers. In the sample, the external service is modeled as a raw JSON document hosted on GitHub (at a `raw.githubusercontent.com` endpoint). The following specification creates a service entry that enables our service to communicate with the external service.

Code Listing 30: Service entry specification

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: exotic-fruits-service-entry
  namespace: micro-shake-factory
spec:
  hosts:
  - raw.githubusercontent.com
  location: MESH_EXTERNAL
  ports:
  - number: 443
    name: https
    protocol: TLS
  resolution: DNS
```

For this demo, we will first find out what happens when the **juice-shop** API service tries to invoke an external API that the service registry cannot resolve.



Tip: To fetch service entry configurations available in your mesh, execute the following command: `kubectl get serviceentry`.

To start afresh, delete the namespace **micro-shake-factory** and deploy the **juice-shop** API service and a gateway to expose it. Remember to substitute the `&` operator with the `&&` operator in the following command if you are using the Bash terminal.

Code Listing 31: Apply new specification

```
$ kubectl delete namespace micro-shake-factory & kubectl apply -f juice-shop-
api.yml & kubectl apply -f juice-shop-api-vs-gw.yml

namespace/micro-shake-factory deleted
namespace/micro-shake-factory created
deployment.apps/juice-shop-api-deployment created
service/juice-shop-api-service created
gateway.networking.istio.io/juice-shop-api-gateway created
virtualservice.networking.istio.io/juice-shop-api-vservice created
```

Before we can proceed with checking the response of the service, we must invert the generous traffic passthrough policy by updating the ConfigMap of Istio named **istio**. The various keys of this ConfigMap act as the switches that control the behavior of Istio.



Tip: If you ever forget to turn on a setting while deploying Istio, you can edit the ConfigMap file and turn it on later.

Let's retrieve this **configmap** object in YAML format by executing the following command.

Code Listing 32: Get Istio configmap

```
$ kubectl get configmap istio -n istio-system -o yaml
```

Copy the object in the result and locate the setting key named **outboundTrafficPolicy**, whose **mode** property is currently set to **ALLOW_ANY** to allow unbound access to external services. For a tightly regulated cluster, you might not want to allow services to have uncontrolled access to the internet. To alter this behavior, replace the value with **REGISTRY_ONLY** and save and apply this file back to the cluster. You can execute the following command in PowerShell Core or use the **sed** command in Bash.

Code Listing 33: Configure configmap

```
$ kubectl get configmap istio -n istio-system -o yaml | Foreach {$_ -replace "mode: ALLOW_ANY", "mode: REGISTRY_ONLY"} | kubectl apply -f -  
  
configmap/istio configured
```

The previous command will update the **mode** property and apply the resulting configuration to the cluster.



Note: To enable the test scripts to work across platforms, we have used PowerShell Core, which is a cross-platform shell built on the .NET Core runtime. The home page and installation instructions for PowerShell Core are available [here](#).

Let's now try issuing a request to the **exoticFruits** endpoint to fetch some fruits.

Code Listing 34: Invoke exoticFruits endpoint

```
curl http://localhost/api/juice-shop/exoticFruits -H "Host: juice-shop.istio-succinctly.io"  
  
{  
  "code": "Internal",  
  "message": "Error: FetchError: request to https://raw.githubusercontent.com/Istio-Succinctly/ExoticFruits/master/fruits.json failed, reason: Client network socket disconnected before secure TLS connection was established"  
}
```

The result of the previous command indicates that the egress gateway is terminating requests before they leave the mesh. Let's create a service entry object that adds the external service to the registry.

Code Listing 35: Create service entry

```
$ kubectl apply -f exotic-fruits-se.yml  
serviceentry.networking.istio.io/exotic-fruits-service-entry created
```

Let's issue another HTTP request to the **exoticFruits** endpoint.

Code Listing 36: Invoke exoticFruits endpoint

```
$ curl http://localhost/api/juice-shop/exoticFruits -H "Host: juice-shop.istio-succinctly.io"  
  
{"fruits":["Chom Chom","Mangosteen","Longan","Persimmon","Sapodilla"]}
```

The endpoint now works as expected. While we are on the topic of service-to-service communication, let's also discuss how services within the mesh communicate with each other. Services inside the mesh already have their DNS records present in the registry, and therefore, they can be easily addressed using their hostnames within the cluster. Also, service-to-service communication does not require gateway instances, which are only used to expose a service to the internet. Let's deploy the **fruits** API service without a gateway by executing the following command.

Code Listing 37: Apply updated specification

```
$ kubectl apply -f fruits-api.yml & kubectl apply -f fruits-api-vs.yml  
  
namespace/micro-shake-factory unchanged  
deployment.apps/fruits-api-deployment-v1 created  
service/fruits-api-service created  
virtualservice.networking.istio.io/fruits-api-vservice created
```

If you explore the specification present in the **juice-shop-api.yml** document, you see that we have set the value of environment variable **FRUITS_API** as **http://fruits-api-service**. The **juice-shop** API is programmed to treat this value as the hostname of the **fruits** API service. Let's invoke the **blender** endpoint of the **juice-shop** API to make it fetch prices of fruits from the **fruits** API and return the name and price of the juice requested.

Code Listing 38: Invoke blender endpoint

```
$ curl -H "Host: juice-shop.istio-succinctly.io" -H "Content-Type: application/json" -H "country: au" -d "{\"fruits\":[\"kiwi\",\"mandarin\"]}" http://localhost/api/juice-shop/blender  
  
{"juice":"kiwi-mandarin juice","price":4.9}
```

You can see that service entry records do not alter the behavior of services in the mesh. Even with Istio's default traffic policy in place, you can use service entry in conjunction with virtual service to force external traffic through an egress gateway. However, there are limited use cases of such implementations.

Destination rule

The destination rules determine how the destination services are invoked by the services on the mesh. The destination rules are applied to the connection after the route of the request has been determined by the proxy. The rules allow cluster operators to configure the following features:

- Load balancing strategy such as round robin and least connection.
- Ability to select a subset of services based on a condition, typically the version.
- Network-level and application-level connection pool settings to control connections that can be established with the destination service.
- SSL/TLS settings for the server.
- Outlier detection to remove unhealthy instances from the load balancer pool.

Since there are several independent features that are implemented by the destination rule, we will discuss the keys and supported values of the destination rule object in context of the feature under discussion. Overall, the destination rule specification uses the following scheme.

Code Listing 39: Destination rule specification

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: name-of-destination-rule
spec:
  host: host.namespace.svc.cluster.local
  trafficPolicy:
    subsets:
  exportTo:
```

The following are the values that each of the keys in the previous listing support:

- **host*: string**
FQDN of the service in the service registry. Applies to both HTTP and TCP services. Service names are looked up in the Kubernetes service registry and in hosts declared in service entry.
- **trafficPolicy: TrafficPolicy**
One or more of the traffic policies to apply: Load balancing, connection pool size, and outlier detection.
- **subsets: Subset[]**
One or more named sets that represent different versions of the service. Setting here has a higher priority than those in **trafficPolicy**.
- **exportTo: string[]**
List of namespaces to which the policies are visible. Currently, only `.` and `*` are supported to denote current namespace and all namespaces.

In the following discussion, you will see only partial specifications (for brevity) that are values of the key **spec** in the structure shown previously.

As you can see in the previous code listing, the destination rule is composed of two families of settings or keys: **trafficPolicy** and **subset**. Let's begin by exploring the traffic policy setting.

The traffic policy setting can be applied to all ports of the destination service or can be scoped to a specific port of the service. The following listing outlines the keys to the traffic policy setting. Since the same setting is scoped to two different levels, you will find that you can use the same policies at two scope levels.

Code Listing 40: Traffic policy specification

```
trafficPolicy:
  connectionPool:
  loadBalancer:
  outlierDetection:
  tls:

  portLevelSettings:
  - port:
      number:
      name:
      loadBalancer:
      connectionPool:
      outlierDetection:
      tls:
```

We will discuss the individual policies scoped to both a destination and a port in detail in the next section. Here, I would like to draw your attention to the **portLevelSettings** element. You can specify the port on which the policy should be applied by setting either the port number by specifying a value for the key **number**, or by specifying the name of the port in the format **<protocol name>-<DNS label>**, for example, **http-message** with the port number. If you don't specify port-level settings, the traffic policy settings are applied to all the ports of the host. Port-level settings have higher precedence than host-level settings.

With the policy specificity out of the way, let's continue our discussion of the various traffic policy settings. Just remember that you can apply any of the following policies at the scope that you desire.

Connection pool settings

The connection pool settings help you control the velocity of data transmitted to the destination service by limiting the number of concurrent HTTP or TCP connections that can be maintained with the host. For protocols that use HTTP (HTTP1.1, HTTP2, and gRPC), this setting allows you to limit the number of outstanding requests to the destination and limit the number of retries of each request to an overwhelmed service. The connection pool settings are further classified as the following:

- **TCP settings:** TCP is the underlying protocol for HTTP, and therefore, these settings are common to both HTTP and TCP server connections.
- **HTTP settings:** These settings apply to HTTP1.1/2 and gRPC server connections.

The following listing outlines the keys of TCP settings of the connection pool policy in YAML format.

Code Listing 41: TCP settings specification

```
host:
trafficPolicy:
  connectionPool:
    tcp:
      maxConnections:
      connectTimeout:
      tcpKeepalive:
        time:
        interval:
```

The following list explains the supported values for each key in the previous structure:

- **maxConnections: int32**
The maximum number of connections allowed to a destination host. The default number is 1,024.
- **connectTimeout: protobuf.Duration**
The TCP connection timeout.
- **tcpKeepalive: tcpKeepalive**
The **SO_KEEPALIVE** socket option is used to enable a heartbeat, which keeps the socket open by sending keep-alive packets at regular intervals over the socket connection. This key consists of the following fields:
 - **probes: int32**: The number of unacknowledged **SO_KEEPALIVE** probes that cause the connection to terminate. The default value is 9 in Linux.
 - **time: protobuf.Duration**: The duration at which if the connection is idle, keep-alive probes will be sent. The default value is 2 (hours).
 - **interval: protobuf.Duration**: The time duration between **KEEPALIVE** probes. The default value is 75 seconds.

Next, we will discuss the keys in the HTTP settings of the connection pool policy.



Note: Duration in protobuf is expressed in seconds and nanoseconds (10^{-9} seconds). The value of seconds must be from -315,576,000,000 to +315,576,000,000 inclusive. The value of nanoseconds must be from -999,999,999 to +999,999,999 inclusive. The format of values understood by Istio is 1h/1m/1s/1ms.

The following listing outlines the keys of HTTP settings of the connection pool policy in YAML format.

Code Listing 42: HTTP settings specification

```
host:
trafficPolicy:
  connectionPool:
    http:
      http1MaxPendingRequests:
```

```
http2MaxRequests:
maxRequestsPerConnection:
maxRetries:
idleTimeout:
h2UpgradePolicy:
```

The following section describes the values that the keys presented in Code Listing 42 support:

- **http1MaxPendingRequests: int32**
The maximum number of pending HTTP requests to destination. The default value is 1,024.
- **http2MaxRequests: int32**
The maximum number of HTTP requests that can be made to the server. The default value is 1,024.
- **maxRequestsPerConnection: int32**
The maximum number of requests that can be sent on a single connection. Setting it to **1** disables the **SO_KEEPALIVE** setting.
- **maxRetries: int32**
The maximum number of retries that can be outstanding for this host across all the connections. The default value is 1,024.
- **idleTimeout: protobuf.Duration**
The duration, after which an inactive connection will be closed. If not set, the connection will be closed after sending each request.
- **h2UpgradePolicy: H2UpgradePolicy**
It supports one of these values: **DEFAULT**, **DO_NOT_UPGRADE**, or **UPGRADE**. The connection will be updated to HTTP/2 based on the specified policy.

Let us now discuss another family of settings in the TLS settings destination rule.

TLS settings

These settings enforce how a sidecar should secure a connection with the destination endpoint on any channel (HTTP or TCP). The following is an extract of this setting that shows the various keys in it.

Code Listing 43: TLS settings specification

```
host:
trafficPolicy:
  tls:
    mode: MUTUAL | SIMPLE | ISTIO_MUTUAL | DISABLE
    clientCertificate:
    privateKey:
    caCertificates:
    subjectAltNames:
    sni:
```


There are four security modes that this setting supports. This is a required value in the specification.

- **DISABLE:** Disable TLS for TCP/HTTP connections.
- **SIMPLE:** Establish a TLS (also known as SSL) connection to the destination endpoint.
- **MUTUAL:** Establish a secure connection to the destination endpoint using mutual TLS (mTLS).
- **ISTIO_MUTUAL:** Just like the MUTUAL mode, this mode uses mTLS to establish a secure connection with the host. However, unlike the **MUTUAL** mode, this mode uses certificates generated by Istio for authentication.

Here is the gist of how these protocols function, which will bring out the differences between TLS and mTLS:

- **TLS:** The client requests a valid certificate issued by a trusted certificate authority (CA), which is trusted by the client and server, from the server. The client matches the DNS name on the certificate with the DNS name of the server, among other properties. Next, the client encrypts the data with the public key of the certificate, which the server can subsequently decrypt with its private key.
- **mTLS:** This is commonly used for server-to-server communication. The servers first exchange certificates issued by a mutually trusted CA with each other to prove their identity. Subsequently, servers can securely communicate with each other using the certificates.

Let's now go through the supported values of the remaining keys in the specification:

- **clientCertificate*: string**
This is required only if the mode is **MUTUAL**. The value is the path to the client-side TLS certificate to use for communication.
- **privateKey*: string**
This is required only if the mode is **MUTUAL**. The value is the path to the private key of the client, which will be used to decrypt data.
- **caCertificates: string**
Used to specify the path to the file containing CA certificates if you wish to verify whether the presented certificate is issued by a trusted CA.
- **subjectAltNames: string[]**
A list of alternate names that the proxy will use to match against the subject identity of the certificate of the destination service.
- **sni: string**
Server Name Indication (SNI) is an enabler for servers to host multiple TLS certificates for multiple sites under a single IP address. In this mechanism, the client needs to specify the hostname that it wants to connect to, and the server responds with a certificate that the client can use to connect with just that resource over the shared IP address. This setting is used to supply the desired hostname to the proxy.

Let's now discuss the versatile load balancing features that you can configure through destination rules.

Load balancer settings

Client-side load balancing is a crucial feature that you can configure through destination rules. By enabling clients to implement load balancing, we can eliminate the need for an external load balancer and enable clients to select the backend to send a request based on factors such as error rate (outlier detection).

Load balancing policies support two types of load-balancing strategies:

- **Simple:** This policy is used to specify a built-in load balancing algorithm, such as round robin, that will be applied to the destination service.
- **Consistent hash:** This is an HTTP-only Envoy configuration that provides session affinity based on attributes such as HTTP headers and cookies. This enables you to implement a custom load-balancing strategy strongly tied to your needs.



Note: The consistent hashing strategy uses the Ketama hashing algorithm that models hashed addresses of servers or a hashed key as points on a ring. One or more points on the computed ring maps to a server. Adding or removing servers from the ring affects only a small number of hashes, and therefore, all the hashes won't need to be recomputed with each operation.

The following listing shows the keys of the simple load balancing setting.

Code Listing 44: Simple load balancing specification

```
spec:
  host:
    trafficPolicy:
      loadBalancer:
        simple:
```

The simple load balancing has just one value to configure: **simple**.

- **simple: SimpleLB**
A single value out of **ROUND_ROBIN**, **LEAST_CONN**, **RANDOM**, and **PASSTHROUGH** that signifies the algorithm to use to select a destination service endpoint for a routing request. **ROUND_ROBIN** is the default algorithm. **LEAST_CONN** selects one of the healthy hosts with the least requests and routs traffic to it. **RANDOM** sends traffic to a random endpoint, and it performs better than **ROUND_ROBIN** if no health-check policy is configured. **PASSTHROUGH** forwards requests to a destination IP address to the destination without executing any load balancing algorithm.

The second category of load balancing strategy is consistent hashing. The following listing presents the keys that are used to configure this setting.

Code Listing 45: Consistent hashing specification

```
spec:
  host:
    trafficPolicy:
      loadBalancer:
        consistentHash:
```

```
httpHeaderName:
useSourceIp:
httpCookie:
  name:
  path:
  ttl:
minimumRingSize:
```

Let's discuss the values that each of the keys in the previous listing supports:

- **httpHeaderName: string**
As you know, the Ketama hashing algorithm requires a key to generate a hashed value. In this case, the hashed value of the header that you specify will be considered for selecting the destination service endpoint. You can select only one of **httpHeaderName**, **useSourceIp**, or **httpCookie** to generate a hash, not all of them.
- **httpCookie: HttpCookie**
The values of this setting help determine the cookie whose hashed value will be used to determine the endpoint to which the request should be forwarded. It consists of the following keys:
 - **name*: string**: Name of the cookie.
 - **path: string**: Path of the cookie.
 - **ttl*: protobuf.Duration**: Lifetime of the cookie.
- **useSourceIp: bool**
The value of this setting instructs Envoy to compute the hash using the IP address of the source.
- **minimumRingSize: uint64**
Ketama models the various hash values as points on a ring that has a many-to-one relationship with the host or destination endpoint. With this setting, you can specify the minimum number of points that you want on the Ketama ring. As you can guess, the more points there are, the finer the load distribution will be. If the number of destination service endpoints is higher than this value, their instance count will be considered instead of the value that you specify.

The last setting in the family of traffic policy settings is outlier detection. Let's now discuss how this policy can be configured.

Outlier detection

Circuit breaker is a typical pattern used in distributed systems to eliminate unresponsive services from processing requests. The basic idea behind the pattern is that you wrap every request in a monitoring component that monitors for failures. After the count of failures for a particular endpoint reaches a threshold, the circuit breaker trips for some time and requests are no longer routed to the endpoint. This gives the faulty service time to recover without getting overwhelmed with requests.

The outlier detection feature helps eliminate faulty endpoints from an active load balancing set based on the error threshold that you configure. Outlier detection applies to both HTTP and TCP services. For HTTP services, HTTP status codes 5xx are treated as errors. For TCP services, connection timeouts and connection failures are treated as errors. Envoy performs outlier detection for each client, since the error received from the server might be specific to a client. The following listing shows the keys that make up the configuration of this feature.

Code Listing 46: Outlier detection specification

```
spec:
  host:
    trafficPolicy:
      outlierDetection:
        consecutiveErrors:
        interval:
        baseEjectionTime:
        maxEjectionPercent:
        minHealthPercent:
```

Let's now go through the values that are supported by the keys of the policy:

- **consecutiveErrors: int32**
This is the threshold of the number of consecutive errors received by Envoy before evicting the host endpoint from the pool of active endpoints. The default value is 5.
- **interval: protobuf.Duration**
The value of this key denotes the periodic interval, after which Envoy scans all the endpoints and decides whether an endpoint should be evicted or included in the pool of active endpoints. The default value is 10s.
- **baseEjectionTime: protobuf.Duration**
An unhealthy endpoint remains evicted from the active connection pool for the duration (= **baseEjectionTime** * number of times the host has been ejected). This adds an exponential delay to the time during which the service can recover. The default value is 30s.
- **maxEjectionPercent: int32**
The maximum percent of endpoints of the destination service that can be ejected. The default value is 10%.
- **minHealthPercent: int32**
If the percentage of healthy endpoints falls below the percentage specified as the value of this key, the outlier detection setting will be disabled, and the traffic will be routed to all endpoints, regardless of their health. The outlier detection service will resume once the number of healthy endpoints crosses this threshold. The default value is 50%.

Let's go through a sample policy that adds clarity to what we previously discussed.

Code Listing 47: Traffic policy example

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: serviceA-dest-rule
```

```
spec:
  host: serviceA.default.svc.cluster.local
  trafficPolicy:
    outlierDetection:
      consecutiveErrors: 10
      interval: 1m
      baseEjectionTime: 3m
      maxEjectionPercent: 10
```

In the previous example, the destination rule instructs Envoy to evict the endpoints of **serviceA** that have reported 10 consecutive errors in three minutes. Every minute, Envoy cycles through all the endpoints of **serviceA** and decides whether it needs to be added or removed from the pool of active endpoints. Envoy keeps repeating the process until it removes 10 percent of endpoints from the endpoint pool of **serviceA**.

This concludes our discussion of the traffic settings in the destination rule policy. The other family of settings of destination rule policy is the subset, which we will discuss next.

Subset

Subset helps you classify a single destination service into subsets using labels. Within each subset of a service, you can configure all the traffic policies that we previously discussed. Subset settings have higher priority than traffic policy settings. In general, you will find that services are classified by a version using a label with the same name. Using versions and subsets, we can implement microservices patterns such as canary deployments, with which we can gradually shift traffic from an old version of a service to the new version.

The following listing presents the schema of the subset setting in the destination rule policy.

Code Listing 48: Destination rule subset specification

```
spec:
  host:
  subsets:
    - name:
      labels:
        <label-name>: <label-value>
      trafficPolicy:
```

Just like traffic settings, subset settings are applied to a host, which you specify as the value of the **host** key. Let's discuss the supported values of the rest of the keys:

- **name*: string**
This is the name of the subset. The destination service name (host) and subset name are used together for defining a routing rule in the virtual service. We will discuss the virtual service resource in detail in the next chapter.

- **labels: map<string,string>**
Labels are used to select the appropriate destination service endpoints from the service registry. If multiple label conditions are specified, then an endpoint must satisfy all of them for being selected (conditions are ANDed).
- **trafficPolicy: TrafficPolicy**
Specifies the traffic policies that apply to the subset. Subsets inherit traffic policies specified for the host, but will override any matching settings.

This concludes our discussion of two of the networking APIs in Istio: service entry and destination rule. There are two other networking APIs that we will discuss in the next chapter: virtual service and gateway. You will rarely find a practical use of individual networking APIs. However, when they are combined, they act as knobs that you can turn and tune to achieve fine-grained control over your services and implement several microservice architectural patterns. We will study several microservice networking patterns in the next chapter.

Summary

There are several APIs in Istio that help you manage network traffic. In this chapter, we covered two of them in detail: service entry and destination rule. We learned the various settings of service entry and destination rule that we can configure. In the next chapter, we will discuss the remaining networking APIs of Istio and discuss the application of these APIs on our demo application.

Chapter 5 Traffic Management: Part 2

In this chapter, we will start with an explanation of the virtual service and gateway networking APIs. Finally, we will discuss some key microservices traffic management patterns and apply them to our sample application.

Virtual service

A virtual service resource maps a fully qualified domain name (FQDN) to a set of destination services. It also provides other capabilities, such as defining the versions of the service (subsets) that are available, and routing properties, such as retries and timeouts. You have already seen a simple virtual service in action, which we used to deploy the first version of the **fruits** API. Let's revisit that policy now.

Code Listing 49: Fruits API virtual service

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: fruits-api-vservice
  namespace: micro-shake-factory
spec:
  hosts:
    - fruits.istio-succinctly.io
  gateways:
    - fruits-api-gateway
  http:
    - route:
        - destination:
            host: fruits-api-service
            port:
                number: 80
```

The policy in the previous listing maps any request arriving through the **fruits-api-gateway** with hostname **fruits.istio-succinctly.io** to the destination service **fruits-api-service.micro-shake-factory.svc.cluster.local**. A key concept of virtual service that we would like to introduce here is the match filter, which can help you introduce constraints using HTTP request attributes such as URI, scheme, method, and headers. For example, we can add an additional match condition to the previous virtual service specification so that only the requests whose path starts with the string **/api/fruits/** get routed to the **fruits-api-service**. This change won't impact anything, as all the request paths of the **fruits** API do start with the same prefix. However, you can see that this feature grants you much finer control over routing a request to the destination service.

A key aspect of virtual service is that the mapping between a host and match filters can be scoped to a destination service or a subset (see [destination rule](#) in the previous chapter) within the destination service. For example, the following policy directs requests with the prefix `/api/v2/fruits` to the subset `v2` of `fruits-api-service`. It rewrites the request URL before forwarding it to the destination service so that the destination service does not need to implement API versioning internally.

Code Listing 50: Match filters scoped to a subset

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: fruits-api-vservice
  namespace: micro-shake-factory
spec:
  hosts:
    - fruits.istio-succinctly.io
  gateways:
    - fruits-api-gateway
  http:
    - match:
        - uri:
            prefix: /api/v2/fruits/
      rewrite:
        uri: /api/fruits/
      route:
        - destination:
            host: fruits-api-service
            port:
              number: 80
          subset: v2
```

We will discuss a few implementations of subset-based routing soon. Let's now discuss some of the policies supported by the virtual service. Note that for discussing the syntax of any policy, we are only considering the keys within the `spec` field for brevity.

The following listing presents the keys in the schema of the virtual service policy.

Code Listing 51: Virtual service specification

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
spec:
  hosts:
  gateways:
  http:
  tls:
  tcp:
  exportTo:
```


The following are the supported values of the keys of the policy shown in the previous listing. We will revisit the keys with nested schema (complex types) later in this chapter.

- **hosts*: string[]**
This is the list of hostnames to which HTTP or TCP traffic is sent by the clients. The names specified can either be external FQDNs or service names available in the service registry (Kubernetes services or service entry objects). The hostnames can be prefixed with the wildcard character *, or the wildcard character can be used alone for flexible matching with the request. If a service entry is used to add an IP address to the service registry, then the IP address may be used as a value in this field.
- **gateways: string[]**
This is the list of names of gateway objects and sidecars to which the route rules will be applied. Since you can create a single virtual service resource for both the gateway and sidecar resources, while configuring HTTP/TCP routes, you can select the gateway or sidecar to which the rules should be applied—for example, by setting the name of the gateway in the **http.match.gateways** field. The keyword **mesh** is used to imply all sidecars in the mesh. If you want to expand the rules to apply to all sidecars and some gateways, use the word **mesh** as one of the entries in this list.
- **http: HTTPRoute[]**
This ordered list of route rules gets applied to HTTP1/2 and gRPC traffic. The first route rule that matches the request is used. As you can imagine, the routing rules require a port on whose traffic the rules will be applied. Istio uses a convention-based approach for shortlisting the ports. Any service port name prefixed with **http-***, **http2-***, or **grpc-*** is selected (this restriction is slated to be removed in future). If a gateway is used, then ports with protocol HTTP/HTTP2/gRPC/TLS-terminated-HTTPS are selected. In the case of service entry objects, ports with protocol HTTP/HTTP2/gRPC are selected.
- **tls: TLSRoute[]**
This ordered set of rules is applied to non-terminated TLS and HTTPS traffic (SSL termination at destination service instead of sidecar). Remember that since TLS traffic is encrypted, the routing rules rely on data present in the **ClientHello** message, which is an unencrypted handshake initiated by the client with information about TLS encryption supported by the client, such as supported version and cipher suite. The port selection follows a scheme similar to the HTTP setting, i.e. service ports named **https-*** and **tls-*** are selected, and gateway and service entry ports with HTTPS/TLS protocols are selected.
- **tcp: TCPRoute[]**
This ordered set of rules is applied to opaque TCP traffic. Istio shortlists the non-HTTP1/2/gRPC and -TLS ports as candidates on which the rules will be applied.
- **exportTo: string[]**
This is the list of namespaces to which the virtual service is visible. All the sidecars and gateways defined in the specified namespaces will be able to access the virtual service. If no namespaces are specified, the virtual service is exported to all the namespaces. Currently, this field only supports the **.** and ***** characters to denote the current namespace and all namespaces, respectively.

We will cover the HTTP, TCP, and TLS rules in detail later in this chapter.

As you can see from the schema in Code Listing 51, overall, there are three families of settings in a virtual service: **HTTPRoute**, **TCPRoute**, and **TLSRoute**. As evident from the name, **HTTPRoute** is responsible for the match conditions and actions for HTTP1/2/gRPC traffic, and **TCPRoute** is responsible for configuring match conditions and actions for routing TCP traffic. Finally, the **TLSRoute** is responsible for routing non-terminated TLS and HTTPS traffic.

HTTPRoute

The following diagram illustrates how the various components of **HTTPRoute** interact with each other to determine the action that handles the request.

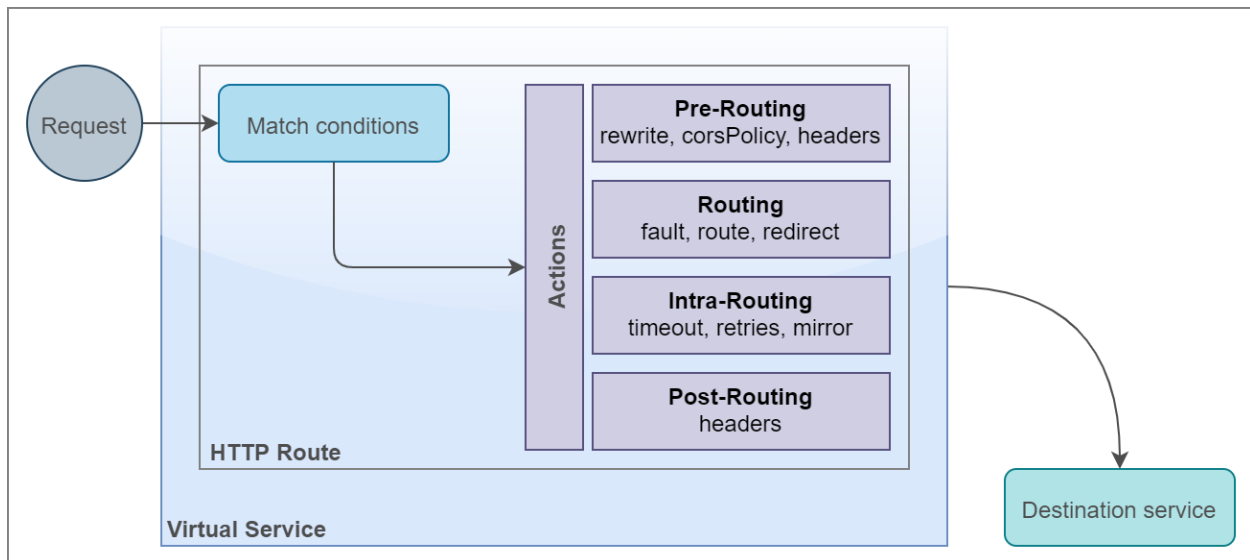


Figure 14: Anatomy of HTTPRoute

On receiving a request targeted to a configured HTTP port of service, the **HTTPRoute** policy executes a match condition on the request based on attributes such as URI, scheme, and headers. Once a match is found, one or more of the following actions take place with the request.

- **Pre-routing actions:** This family of actions prepares or handles the request before it reaches the destination. The following are the sub-policies with their type that make up these actions.
 - **rewrite: HTTPRewrite**
This policy rewrites the actual URIs and authority headers before forwarding the request to the destination route.
 - **corsPolicy: CorsPolicy**
Agents such as browsers send a preflight request before the actual request to ensure that scripts don't illegally request resources outside their domain unless authorized by the remote service. This policy handles and responds to the preflight request without forwarding the request to the destination.

- **headers/request: Headers**
This policy manipulates headers (set, add, remove) before forwarding the request to the destination.
- **Routing actions:** This family of actions forwards the request to the destination. The following sub-policies make up this family:
 - **route: HTTPRouteDestination[]**
The policy defined for this action directs the request to a service defined in the service registry of the platform. The destination service may be one of the subsets of the service defined using the destination rule object.
 - **redirect: HTTPRedirect**
This policy sends a 301 response to the client with the new location of the resource. It can additionally specify the authority/host header value that the client should use for the subsequent request.
- **Intra-routing actions:** This family of actions remains in play during the process of routing, and it can affect the behavior of the routing actions. The following sub-policies make up this family of actions:
 - **fault: HTTPFaultInjection**
This policy configures Envoy to respond to requests with a configured fault. By deliberately injecting faults, developers can test and monitor the behavior of the client applications when remote services are producing errors or delaying responses.
 - **timeout: protobuf.Duration**
This policy defines the interval for the HTTP request timeout. It supports values in the **protobuf.Duration** format.
 - **retries: HTTPRetry**
This policy defines the action to take when an HTTP request fails.
 - **mirror: Destination**
This policy causes Envoy to send requests in parallel to a destination service in addition to the actual target service. Envoy doesn't wait for the secondary destination service to respond before sending the response to the client. This feature helps developers test new versions of an application against actual traffic before replacing the deployed application with a new version of it.
- **Post-routing actions:** This family of actions manipulates the response from the destination before returning it to the client. The following sub-policies make up this family of actions:
 - **Headers/response: Headers**
This policy manipulates headers (set, add, remove) before returning the response to the client.

The following listing represents the schema of the **HTTPRoute** policy.

Code Listing 52: HTTPRoute policy specification

```
spec:
  hosts:
  http:
  - match:
    rewrite:
    route:
    redirect:
    timeout:
    retries:
    fault:
    mirror:
    corsPolicy:
    headers:
```

We have already discussed the effects of the various policies. Let's now discuss the individual sub-policies and their schema.

HTTPMatchRequest

This policy defines the conditions that should be met for the associated rule to be applied. The following code listing presents the keys that make up this sub-policy.

Code Listing 53: HTTPMatchRequest specification

```
spec:
  hosts:
  http:
  - match:
    - headers:
      <header-name>:
        prefix: <header-value>
        exact: <header-value>
        regex: <header-value>
    uri:
      prefix:
      exact:
      regex:
    scheme:
      prefix:
      exact:
      regex:
    method:
      prefix:
      exact:
      regex:
    authority:
      prefix:
      exact:
      regex:
```

```
port:
sourceLabels:
gateways:
queryParams:
  <query-parameter>:
    prefix: <value>
    exact: <value>
    regex: <value>
```

Let us now discuss the values that the keys in the previous listing support.

- **uri: StringMatch**
Specifies constraints that execute on the URI of the incoming request. The constraint matches are case sensitive. The match is performed using one of the following three selectors:
 - **exact**: Performs exact match with the input string.
 - **prefix**: Performs prefix match with the input string.
 - **regex**: Performs regex match conforming to ECMAScript regex specification.
- **scheme: StringMatch**
Specifies constraints that execute the URI scheme of the request. The specification is similar to that of the URI match.
- **method: StringMatch**
Specifies constraints that execute the HTTP method of the request. The specification is similar to that of the URI match.
- **authority: StringMatch**
Specifies constraints that execute the HTTP authority/host of the request. The specification is similar to that of the URI match.
- **headers: map<string, StringMatch>**
Specifies constraints that execute the HTTP headers of the request. The specification is similar to that of the URI match. The header keys should be lowercase and use a hyphen separator.
- **port: int32**
Specifies the port number of the host on whose traffic the match rules are applied. Specifying a port is unnecessary if the host only exposes a single port.
- **sourceLabels: map<string, string>**
Collection of one or more labels and desired value constraints that scope the applicability of the rules to resources that satisfy the conditions. For example, if you want to inject failures in responses from certain pods, you can use the **sourceLabel1** selector to do that.
- **gateways: string[]**
The values denote the names of gateways on which the rules will be applied. The request should flow via the specified gateway to be considered for matches and actions.

- **queryParams: map<string, StringMatch>**
Specifies constraints that execute the HTTP query string parameters. All string match specifications are the same as those of the URI match except the prefix match, which is currently not supported.

Let's discuss the actions in the family of pre-routing actions now.

CORS policy

Cross-origin resource sharing (CORS) restricts the domains that can request resources from a destination service. The HTTP CORS standard requires the client to send a preflight request with the value of HTTP header key **Origin** (optionally along with other headers, such as **Access-Control-Request-Method** and **Access-Control-Request-Headers**) set to the domain of the client. The server then responds to the request with **Access-Control-*** headers such as **Access-Control-Allow-Origin** (for example, `http://juiceshopapi.io`), and **Access-Control-Allow-Methods** (for example, **GET** and **PUT**) to inform the client the types of requests it can make. The agent sending the request is responsible for enforcing CORS constraints. For web applications, web browsers are responsible for enforcing CORS restrictions for requests initiated by scripts. The following listing presents the keys that constitute this setting.

Code Listing 54: CORS policy specification

```
spec:
  hosts:
    http:
      - route:
          - destination:
              host:
                subset:
                  corsPolicy:
                    allowOrigin:
                    allowMethods:
                    allowCredentials:
                    allowHeaders:
                    maxAge:
```

The specification presented in the previous listing enables CORS policy on the host and the subset defined in the **route.destination.host** key and **route.destination.subset** key, respectively. Envoy offloads the responsibility to handle preflight requests from the destination service. On receiving a preflight request from a client for the destination service, Envoy responds with CORS response headers with values as configured by you. Later, when the actual request arrives, Envoy will also use the same policy settings to accept or reject it. Let's discuss the supported values of the keys of the CORS policy.

- **allowOrigin: string[]**
The values denote all the origins that can send requests to the destination service. The values specified for this key are set as values of the **Access-Control-Allow-Origin** HTTP header. Specifying wildcard character ***** will allow all origins to access the service.

- **allowMethods: string[]**
The values denote the HTTP verbs such as **GET** and **PUT** that are allowed for the client by the destination service. The input values are set as values of the **Access-Control-Allow-Methods** HTTP header. It supports the wildcard character ***** to allow all methods.
- **allowHeaders: string[]**
The values denote the headers that the clients can send in a request. The input values are set as values of the **Access-Control-Allow-Headers** HTTP header. It supports the wildcard character ***** to allow any header in the client request.
- **exposeHeaders: string[]**
The value denotes the names of the HTTP headers that will be exposed to the client by the server. The input values are set as values of the **Access-Control-Expose-Headers** HTTP header. It supports wildcard character ***** to expose all headers only in cases when requests don't contain authorization information (as cookies or headers).
- **maxAge: protobuf.Duration**
The value denotes the duration for which the response of a preflight request can be cached. The input value is set as the value of the **Access-Control-Max-Age** HTTP header. The value **1** will disable caching.
- **allowCredentials: bool**
The value denotes whether a client can bypass pre-flight requests using credentials. The input value is set as the value of the **Access-Control-Allow-Credentials** HTTP header.

HTTPRewrite

This policy is used to rewrite parts of an HTTP request before forwarding it to the destination. The following listing presents where **HTTPRewrite** sits in the overall schema of virtual service. On a successful match, the policy will rewrite the request URI and/or authority and forward it to the destination.

Code Listing 55: HTTPRewrite specification

```
spec:
  hosts:
  http:
    - match:
      - uri:
      rewrite:
        uri:
        authority:
      route:
      - destination:
```

HTTPRewrite supports the following values for the keys in its schema:

- **uri: string**
Replaces the path or prefix portion of the original URI with the input value.

- **authority: string**
Replaces the authority/host header with the input value.

Headers

This policy manipulates the HTTP headers of both the request sent to the destination and the response produced by the destination. The following schema presents the various keys of this policy and where it sits within the **HTTPRoute** schema.

Code Listing 56: HTTPRoute headers specification

```
http:
- headers:
  response:
    remove:
    add:
    set:
  request:
    remove:
    add:
    set:
  route:
    - destination:
```

Let's discuss the values that the keys in the previous schema supports:

- **request: HeaderOperations**
These manipulations get applied before the request is forwarded to the destination. The **HeaderOperations** type consists of the following fields:
 - **set: map<string, string>**
Overwrites the value of the request header specified by key with the input value.
 - **add: map<string, string>**
Adds the given values to the headers specified by keys.
 - **remove: string[]**
Removes the specified headers.
- **response: HeaderOperations**
These manipulations are configured similarly to the request manipulations, except that they are applied before returning a response to the client.

Let's proceed on our journey by going through the routing actions next.

Destination

Before discussing the routing actions, it is necessary to understand the concept of destination. The type destination encapsulates the details of the target or destination service to which the request will be forwarded. A critical feature of virtual service is to direct requests targeted at a host to a destination service or subset based on the specified routing rule. We have already seen an example of this policy in action where we directed requests to the host `fruits.istio-succinctly.io` to destination `fruits-api-service.micro-shake-factory.svc.cluster.local` based on routing rules. The destination policy can be used to direct incoming traffic to any service in Istio's service registry, which consists of services declared in Kubernetes as well as services declared through the service entry object.



Note: The destination service name defined as the value of the `route.destination.host` key can be a short name as well, such as `serviceA`. This name will be interpreted as `serviceA.<namespace of virtual service>.svc.cluster.local`. If the namespace of your virtual service is different from the namespace of `serviceA`, this may cause issues. It is recommended to use FQDN instead of short names to avoid confusion.

The following code sample shows where the destination fits into the structure of a virtual service.

Code Listing 57: Virtual service destination specification

```
spec:
  hosts:
  - services
  http:
  - match:
    - uri:
        prefix: "v2/path1"
    route:
    - destination:
        host: service2
        subset: v2
    - route:
        - destination:
            host: service1
            subset: v1
```

The previous configuration executes a URI prefix check on every request that is sent to the HTTP endpoint of the `services` host. In the case of a match, the destination of the request is set to a host named `service2`. Otherwise, the request is directed to the destination service `service1`.

The destination is one of the properties in the route key of the virtual service schema, with a specification as follows.

Code Listing 58: Virtual service destination key specification

```
route:
- destination:
  host:
  subset:
  port:
```

Let's discuss the values that the schema in the previous listing supports:

- **host*: string**
The name of the service available in the platform service registry, including hosts declared by the service entry. Traffic bound to unknown hosts is discarded.
- **subset: string**
The name of the subset as declared by the destination rule object.
- **port: PortSelector**
Specifies the port on the host to which the traffic will be sent. If the destination service has a single port exposed, this value is not required.

HTTPRouteDestination

Now that you understand what destination is, let's discuss the **HTTPRouteDestination** type, which encapsulates one or more destinations and is responsible for deciding the proportion in which the traffic should be split between specific destinations. The following is the schema of this policy and where it fits in the overall schema of virtual service. We have deliberately expanded the schema so that it is easier to understand.

Code Listing 59: HTTPRouteDestination specification

```
spec:
  hosts:
  http:
  - route:
    - destination:
      host:
      subset:
      weight:
      headers:
        request:
        response:
    - destination:
      host:
      subset:
      weight:
      headers:
        request:
        response:
```

The key **weight** in the schema is a numeric value that determines the proportion of traffic that should be allowed to the destination that it is associated with. You can also configure the **request** and **response** headers, which give you more refined control than **http.headers** does. The keys in the schema support the following values:

- **destination*: Destination**
Determines the destination to which the request should be sent.
- **weight*: int32**
Determines the proportion of traffic directed to the associated destination. The total of weights across all destinations must be 100. In the case of a single destination, the weight is assumed as 100 by default.
- **headers: Headers**
Allows you to manipulate the request and response headers for the associated destination.

HTTPRedirect

This policy is used to send a 301 response to the client. Optionally, the response can also direct the client to use an alternative host/authority. The following is the schema of this policy.

Code Listing 60: HTTPRedirect specification

```
spec:
  hosts:
  http:
  - match:
    redirect:
      uri:
      authority:
```

Here are the values that the keys of the policy support:

- **uri: string**
This is the new path of the resource, which would replace only the path of the original request, for example, **v2/newPath**.
- **authority: string**
This is the new host or authority that the client should use to make the subsequent request.

HTTPFaultInjection

To simulate faults in the destination service, you can configure this policy to manipulate the behavior of the destination service in two ways: add a fixed delay in the response from the destination service and abort the request, and send a defined HTTP status code in response.

This policy is beneficial for developers and operators to test the behavior of a service when its dependent services respond erratically. The following is the schema of this policy that also shows where it sits in the overall schema of virtual service.

Code Listing 61: HTTPFaultInjection specification

```
spec:
  hosts:
    http:
      - route:
          - destination:
              host:
            fault:
              abort:
                percentage:
                  value:
                httpStatus:
              delay:
                percentage:
                  value:
                fixedDelay:
```

As we previously discussed, there are two types of fault policies that we can associate with a destination. The **delay** policy makes Envoy wait for the configured duration before sending the request to the destination. The **abort** policy short-circuits the request and returns the configured HTTP status code to the client. Both policies are independent of each other.

The following are the values that are supported by the keys in the schema:

- **percentage: Percent**
Specifies the percent of requests that will be aborted or receive a response after a delay. The percentage/value key accepts a numeric value between 0.0 and 1.0.
- **httpStatus*: int32**
The numeric HTTP status code that will be returned to the client.
- **fixedDelay*: protobuf.Duration**
The duration for which the proxy will wait before responding to the client request.

Let's discuss the elements of intra-routing actions that remain active during the process of routing requests.

Timeout

You can set up this policy to configure the duration for which Envoy will wait for the destination service to respond to the forwarded request. The following schema shows the position of this policy within the **HTTPRoute** policy.

Code Listing 62: Timeout specification

```
http:
```

```
- timeout:
  route:
- destination:
  host:
```

The **timeout** field accepts a single value of type **protobuf.Duration**.

HTTPRetry

This policy sets up the retry policy to use when the HTTP request to the destination fails. The following schema shows the **HTTPRetry** policy with its placement in virtual service.

Code Listing 63: HTTPRetry specification

```
spec:
  hosts:
  http:
    - route:
      - destination:
        retries:
          attempts:
          perTryTimeout:
          retryOn:
```

The following are the valid values for the keys in the policy:

- **attempts*: int32**
This denotes the number of retries Envoy will make for a request. The interval between retries is automatically determined.
- **perTryTimeout: protobuf.Duration**
Denotes the timeout duration ($\geq 1\text{ms}$) for each retry attempt.
- **retryOn: string**
This is the comma-delimited string (CSV) of names of policies for which the retry operation will take place. The following policies are supported:
 - **5xx**: The destination service returned **5xx** as a response code or timed out.
 - **gateway-error**: The destination service returned a 502, 503, or 504 response code.
 - **reset**: The destination service failed to respond after the connection was established (disconnect, reset, or read timeout).
 - **connect-failure**: The client failed to connect to the destination service, for example, a connection timeout.
 - **retriable-4xx**: The destination service returned a retriable error status code in response, i.e. 4xx. Currently, it only retries for code 409.
 - **refused-stream**: The destination service reset the stream and responded with the **REFUSED_STREAM** error code.
 - **retriable-status-code**: The destination service responded with a retriable status code for an HTTP or gRPC request.

Mirror

This policy configures Envoy to route traffic to a secondary destination in addition to the primary destination. The following is the schema of this policy.

Code Listing 64: Mirror specification

```
spec:
  hosts:
    http:
      - route:
          - destination:
              host:
              subset:
        mirror:
          host:
          subset:
```

The value of the **mirror** policy is of type **destination**. As you can see from the specification, **mirror** can route requests to a host and subset such that all the traffic targeted to a destination host and subset also reaches its mirrored counterpart.

Let's briefly cover the second key member of the virtual service family: **TCPRoute**.

TCPRoute

Just like **HTTPRoute**, this policy covers the match conditions and actions for routing TCP traffic. The following is the schema of **TCPRoute**, and it presents all the keys required by this policy.

Code Listing 65: TCP route specification

```
spec:
  hosts:
    tcp:
      - match:
          - port:
              destinationSubnets:
              sourceLabels:
              gateways:
        route:
          - destination:
              host:
              subset:
              port:
              number:
              weight:
```

The schema of **TCPRoute** resembles that of **HTTPRoute** except that the rules are applicable at Level 4 rather than Level 7 because the contents of individual TCP packets can't be read. Let's discuss the values that the keys support:

- **match: L4MatchAttributes[]**
Each value in the list specifies the set of conditions that must match for the rule to evaluate to true (conditions are ANDed). You can specify multiple match conditions as well, at least one of which must match for the rule to apply (matches are ORed). The following are the supported values of the key type **L4MatchAttributes**.
 - **port: int32**
This specifies the port number of the host on whose traffic the match rules are applied. Specifying a port is unnecessary if the host only exposes a single port.
 - **destinationSubnets: string[]**
The IPv4 or IPv6 address of the destination with optional subnet, such as **1.2.3.4/8** or **1.2.3.4**.
 - **sourceLabels: map<string, string>**
These label constraints restrict the rule to matching workloads.
 - **gateways: string[]**
This is the name of the gateways where the rule should be applied.
- **route: RouteDestination[]**
These values denote all the destinations to which the request should be forwarded. The **RouteDestination** consists of two required elements: **destination** of type **Destination** and **weight** of type **int32**. We have discussed these types in detail previously.

The final member of this family of settings is **TLSRoute**, which we will discuss next.

TLSRoute

Just like other policies of its type, it describes match conditions and actions for routing passthrough TLS and HTTPS traffic. This is not as flexible as the **HTTPRoute** policy; because the content of request is encrypted, routing decisions can't be made based on the content of the request. The following is the schema of this policy and its placement in virtual service.

Code Listing 66: TLSRoute specification

```
spec:
  hosts:
  tls:
  - match:
    - port:
    - sniHosts:
    - destinationSubnets:
    - sourceLabels:
    - gateways:
  route:
  - destination:
    host:
    subset:
    weight:
```

Let's discuss the keys and the values they support:

- **match*: TLSMatchAttributes[]**
These are match conditions that must be satisfied for the rule take effect. Different match blocks are ORed, and conditions in a single match block are ANDed. The following are the supported values of the keys of type **TLSMatchAttribute**.
 - **sniHosts*: string[]**
This is the Server Name Indication (SNI) to match on. The values support wildcard character * as well. For example, *.microshake-factory.io will match fruits.microshake-factory.com.
 - **destinationSubnets: string[]**
This is the IPv4 or IPv6 address of destinations with optional subnet.
 - **port: int32**
This is the port on the host that will be monitored. It is not required if the service has a single port.
 - **sourceLabels: map<string, string>**
These are the label constraints to select the applicable workload.
 - **gateways: string[]**
These are the names of gateways on which the rule is applied.
- **route: RouteDestination[]**
These values denote the destination to which the request should be forwarded. **RouteDestination** has two nested keys whose value must be specified: **destination** of type **Destination** and **weight** of type **int32**. We have already discussed these types in detail.

Phew! Those were a lot of policies to cover. There is one final component that we need to discuss before we can get ready to put what we've learned into action.

Gateway

By default, services on the mesh are not exposed outside the cluster. Istio gateways are proxies that control the traffic that flows in and (optionally) out of the trusted network boundary of the mesh. To allow the traffic into the cluster, by default, Istio creates a Kubernetes load balancer resource named **istio-ingressgateway**. Execute the following command to view the public IP address of the service.

Code Listing 67: Get ingress gateway

```
$ kubectl get svc istio-ingressgateway -n istio-system
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
istio-ingressgateway	LoadBalancer	10.106.126.169	localhost	15020:30535/TCP,...

On Docker Desktop for Mac/Windows, the ingress gateway service is exposed on **localhost:80**. The load balancer or ingress gateway can bring traffic into the cluster, but it lacks a route that will connect it to the mesh. You use the Istio gateway resource to configure the ingress gateway to allow incoming traffic destined for selected hosts or ports.

The other category of gateways, called the egress gateways, allow access to external HTTP/S services from services in the mesh. Essentially, both types of gateways are load balancers that apply L4–L6 (transport, session, and presentation) policies on requests to route them to appropriate destinations. In [Chapter 3](#), we deployed a gateway resource to expose our services to external clients. Let's go through the schema of a gateway resource.

Code Listing 68: Gateway resource specification

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name:
  namespace:
spec:
  selector:
    istio: ingressgateway | egressgateway
  servers:
  - port:
      number:
      name:
      protocol:
    hosts:
    tls:
      mode: SIMPLE | MUTUAL | PASSTHROUGH | AUTO_PASSTHROUGH
      serverCertificate:
      privateKey:
      caCertificates:
      credentialName:
      subjectAltNames:
      minProtocolVersion: TLS_AUTO | TLSV1_0 | TLSV1_1 | TLSV1_2 | TLSV1_3
      maxProtocolVersion: TLS_AUTO | TLSV1_0 | TLSV1_1 | TLSV1_2 | TLSV1_3
      cipherSuites:
    defaultEndpoint:
```

You may have noticed that we used a label selector to apply the policies on all the gateway pods that have a label named **istio** with the value set to **ingressgateway** (for ingress gateway) and **egressgateway** (for egress gateway). These are the default labels applied by Istio to the pods of the **istio-ingressgateway** and **istio-egressgateway** services. However, you can customize the default definitions of the gateway, and the selectors would change accordingly.

Let's discuss the values that the keys in the schema support:

- **servers*: Server[]**
These values denote the L4–L6 configurations that will be applied to the gateway. The following are the keys and their supported values for this type:

- **port*: Port**
These are the ports on the proxy that will accept traffic. The type **Port** itself is made up of three keys: **number***, which takes a numeric value for the port number, **protocol***, which is the name of the protocol of incoming traffic, and **name**, which is the label assigned to the port. The protocol can have one of these values: **HTTP**, **HTTPS**, **GRPC**, **HTTP2**, **MONGO**, **TCP**, or **TLS**. The value **TLS** means that TLS would not be terminated at the proxy.
- **hosts*: string[]**
These values specify the hostnames that are exposed by this gateway. The hostnames are specified in the format **namespace/dnsName** where **namespace** is an optional string, and **dnsName** should be a FQDN. The **dnsName** supports the ***** wildcard character as either ***.example.com** or *****, which satisfies the host constraint of the associated virtual service. The namespace supports wildcard character ***** to represent services from any namespace and **.** to represent the namespace of the gateway resource. The default value is *****.
- **tls: TLSOptions**
These settings control HTTP to HTTPS redirection and the TLS mode to use. We'll cover the various keys of TLS options right after this section.
- **defaultEndpoint: string**
This is the localhost endpoint or Unix domain socket to which traffic should be sent by default. It can be one of **127.0.0.1:<port>** or **unix://<abstract namespace>**.
- **selector*: map<string, string>**
As discussed, the values of this key denote one or more label-value pairs, all of which would be used to select the pods on which the configuration will be applied. The pods are looked up in the same namespace as the gateway; therefore, the namespace should be the same as that of the gateway.

Let's discuss the various keys that make up the **TLSOptions** type.

- **httpsRedirect: bool**
When set to **true**, the value directs the load balancer to send 301 redirects for HTTP connections.
- **mode: TLSMode**
Determines the policy that the proxy enforces on the connection. It supports one of the following values:
 - **PASSTHROUGH**: Use the SNI string of the request to match the criteria in the virtual service TLS route.
 - **SIMPLE**: Secure connection with simple TLS (SSL).
 - **MUTUAL**: Secure connection using mutual TLS (mTLS).
 - **AUTO_PASSTHROUGH**: Similar to the **PASSTHROUGH** mode without the requirement of matching the SNI string to a service in the registry. It requires both the client and destination to use mTLS for security.
 - **ISTIO_MUTUAL**: Similar to the **MUTUAL** mode, but uses certificates generated by Istio.

- **serverCertificate*: string**
This is required if the mode is **SIMPLE/MUTUAL**. The value denotes the path to a server-side TLS certificate.
- **privateKey*: string**
This is required if the mode is **SIMPLE/MUTUAL**. The value denotes the path to the server's private key.
- **caCertificates*: string**
This is required if the mode is **MUTUAL**. The value denotes the path to CA certificates used to verify the client certificates.
- **credentialName: string**
Denotes a unique identifier that can be used to identify the **serverCertificate** and the **privateKey**. If specified, the **serverCertificate** and **privateKey** will be queried from credential stores such as Kubernetes certificates.
- **subjectAltNames: string[]**
This list of alternate names verifies the identity of the subject in the client certificate.
- **verifyCertificateSpki: string[]**
This is a list of base 64-encoded SHA-256 hashes of SPKIs of authorized client certificates.
- **verifyCertificateHash: string[]**
This is a list of hex-encoded SHA-256 hashes of the authorized client certificates.
- **minProtocolVersion: TLSProtocol**
This is the minimum TLS protocol version. Value can be one of **TLS_AUTO**, **TLSV1_0**, **TLSV1_1**, **TLSV1_2**, or **TLSV1_3**.
- **maxProtocolVersion: TLSProtocol**
This is the maximum TLS protocol version.
- **cipherSuites: string[]**
The cipher suite to support. Otherwise it defaults to Envoy's default cipher suite.

By now, we have discussed all the networking APIs and their configurations. Individually, the APIs don't have much to offer, but you can mix and match them together to add robust traffic management features to a service. Let's now use the networking APIs to realize some common microservices patterns and improve our sample application iteratively. Download the source code from [GitHub](#) and navigate to **Policies > Chapter 4 5** to follow along with us through the exercises.

Pattern 1: Request routing

Versioning is a critical aspect of microservices to maintain compatibility with existing clients. A microservice will need to change over time, and when that happens you will need to incrementally deploy the new version of your service to ensure that clients can gradually migrate to the updated service. We will use subsets to differentiate between the different versions of the same service.



Note: Since subset settings are external to Kubernetes, you can address scenarios where different versions of your service are deployed inside the Kubernetes cluster and outside the cluster. This is also a powerful technique for iteratively migrating your workloads to Kubernetes.

In this task, we will deploy two versions of the **fruits** API service and route traffic to appropriate endpoints using gateway rules. Execute the following command to delete the namespace to ensure a clean working environment.

Code Listing 69: Delete namespace

```
$ kubectl delete namespace micro-shake-factory  
namespace "micro-shake-factory" deleted
```

Execute the following command to create two versions of the **fruits** API service as two different deployments.

Code Listing 70: Fruits API versioned deployment

```
$ kubectl apply -f fruits-api-versioned.yml  
namespace/micro-shake-factory created  
deployment.apps/fruits-api-deployment-v1 created  
deployment.apps/fruits-api-deployment-v2 created  
service/fruits-api-service created
```

Let's now create a virtual service and gateway resource to route requests to the services that we just created.

Code Listing 71: Fruits API subset deployment

```
$ kubectl apply -f fruits-api-dr-vs-gw.yml  
gateway.networking.istio.io/fruits-api-gateway created  
virtualservice.networking.istio.io/fruits-api-vservice created
```

The configuration worth paying attention to here is the following.

Code Listing 72: Fruits API virtual service

```
apiVersion: networking.istio.io/v1alpha3  
kind: VirtualService  
metadata:  
  name: fruits-api-vservice  
  namespace: micro-shake-factory  
spec:  
  hosts:  
    - fruits.istio-succinctly.io
```

```
gateways:
- fruits-api-gateway
http:
- match:
  - headers:
    version:
      exact: "2"
  route:
  - destination:
    host: fruits-api-service-v2
    port:
      number: 80
- route:
  - destination:
    host: fruits-api-service-v1
    port:
      number: 80
```

The previous policy routes requests with a header key named **version** with the value set to **2** to version 2 of the **fruits** API service, and others to version 1 of the service. Let's test the policy by executing the following commands.

Code Listing 73: Invoke fruits API

```
$ curl http://localhost/api/fruits/special -H "Host: fruits.istio-
succinctly.io"

{"ver":"1","fruit":"Mango"}

$ curl http://localhost/api/fruits/special -H "Host: fruits.istio-
succinctly.io" -H "version: 2"

{"ver":"2","fruit":"Orange"}
```

You can also use one or more filters based on the various Level 4–7 attributes that we previously discussed to configure traffic routing.

Pattern 2: Fault injection

It is important to test individual microservices of the application for resiliency to ensure that the application degrades gracefully if one or more of its supporting services behave inconsistently. Istio allows you to configure faults for some percentage of HTTP traffic. You can inject arbitrary delays or return specific response codes (such as 500) and use the failures to write integration tests that ascertain the application behavior in the presence of failures of its dependencies.

Let's introduce deliberate failures in the traffic routed to version 1 of the **fruits** API service. Apply the following configuration to update the **fruits** API virtual service.

Code Listing 74: Fruits API fault injection

```
$ kubectl apply -f fruits-api-dr-vs-gw-test-fault.yml

gateway.networking.istio.io/fruits-api-gateway unchanged
virtualservice.networking.istio.io/fruits-api-vservice configured
```

The following configuration is responsible for injecting faults in the service.

Code Listing 75: Faults specification

```
- route:
  - destination:
      host: fruits-api-service-v1
      port:
        number: 80
    fault:
      delay:
        fixedDelay: 5s
        percentage:
          value: 40
      abort:
        httpStatus: 500
        percentage:
          value: 60
```

The configuration in the previous listing causes 40 percent of the requests to process with a delay of 5 seconds, and 60 percent of them to fail with HTTP status code 500. The following test script sends 10 requests to the service and outputs the response received. Use the PowerShell Core terminal to execute the following script.

Code Listing 76: Test fruits API faults

```
$ .\fruits-api-fault-test.ps1

Request 1: OK
Request 2: OK
Request 3: InternalServerError
Request 4: OK
Request 5: InternalServerError
Request 6: InternalServerError
Request 7: OK
Request 8: InternalServerError
Request 9: OK
Request 10: InternalServerError
```

You will notice that because of the applied configuration, the script takes some time before returning the 200/OK response, while it returns immediately for errors. Note that you won't always get the desired split of errors and successes, since the faults are averaged over time.

Pattern 3: Traffic shifting

Traffic shifting, or canary deployment, is the practice of routing a small portion of the application traffic to a newly deployed workload to validate the quality of the increment. As the validations succeed, you can keep ramping up the percentage of traffic to the new workload until all traffic reaches the new workload.

The canary deployment model is a little different from the blue/green deployment model. Unlike routing all the traffic to one of the two replicas of the application (blue and green), which requires double the capacity of deployment, you require a constant amount of additional resources to propagate the changes.

You can direct the traffic to the canary service in several ways, such as splitting the traffic by percentage or by directing traffic from internal users to the canary service. For this demo, we will split the traffic in a 1:10 ratio between version 2 and version 1 of the **fruits** API service. Execute the following script to deploy the two subsets of service.

Code Listing 77: Fruits API canary deployment

```
$ kubectl apply -f fruits-api-dr-vs-gw-canary.yml

gateway.networking.istio.io/fruits-api-gateway unchanged
virtualservice.networking.istio.io/fruits-api-vservice configured
```

Let's now test this deployment by using another PowerShell script that will send 10 requests to the same endpoint. The results are averaged over time and, therefore, may look different from the ones shown here.

Code Listing 78: Test fruits API canary deployment

```
$ .\fruits-api-canary-test.ps1

Request 1: {"ver":"1","fruit":"Mango"}
Request 2: {"ver":"2","fruit":"Orange"}
Request 3: {"ver":"1","fruit":"Mango"}
Request 4: {"ver":"1","fruit":"Mango"}
Request 5: {"ver":"1","fruit":"Mango"}
Request 6: {"ver":"1","fruit":"Mango"}
Request 7: {"ver":"1","fruit":"Mango"}
Request 8: {"ver":"1","fruit":"Mango"}
Request 9: {"ver":"1","fruit":"Mango"}
Request 10: {"ver":"1","fruit":"Mango"}
```

The following virtual service configuration is responsible for the canary split.

Code Listing 79: Destination weight specification

```
http:
- route:
  - destination:
    host: fruits-api-service-v2
```

```
    port:
      number: 80
  weight: 10
- destination:
  host: fruits-api-service-v1
  port:
    number: 80
  weight: 90
```

The task to split the canary traffic by HTTP attributes is left to you as an exercise. Let's discuss the next pattern now.

Pattern 4: Resilience

A resilient system degrades gracefully when failures occur in downstream systems. To build resilient systems, Istio provides several turnkey features such as client-side load balancing, outlier detection, automatic retry, and request timeouts. We have already discussed the various client-side load balancing strategies; now we will see how you can combine outlier detection (also called circuit breaker), request timeouts, and retries to ensure reliable communication between services. We will configure the following policies together:

- Circuit breaker/outlier detection to eject faulty instances from the load balancer pool.
- Timeout to avoid waiting on a faulty service.
- Retries to forward the request to another instance in the load balancer pool if the primary instance is not responding.

For this demo, we will use an endpoint in the **juice-shop** API service that has a probability of 0.8 to fail. In the real world, you rarely encounter such services that perform so poorly. Our goal is to implement multiple resiliency policies such that we receive much better quality of service (QoS) without making any changes to the API itself. To ensure that we don't have any existing resources that can affect our demo, execute the following command to delete all resources within the namespace and the namespace itself.

Code Listing 80: Delete namespace

```
$ kubectl delete namespace micro-shake-factory

namespace/micro-shake-factory deleted
```

Let's first deploy the **juice-shop** API service and its associated virtual service and gateway resources with the following command.

Code Listing 81: Apply fruits API resilience policy

```
$ kubectl apply -f juice-shop-api.yml -f juice-shop-api-vs-gw.yml

namespace/micro-shake-factory created
deployment.apps/juice-shop-api-deployment created
service/juice-shop-api-service created
```



```
gateway.networking.istio.io/juice-shop-api-gateway created
virtualservice.networking.istio.io/juice-shop-api-vservice created
```

You saw these specifications in action when we discussed service entry, and these policies do not implement any resiliency strategies yet. Let's execute a test that makes 50 requests to the **TestMyLuck** (`/api/juice-shop/testMyLuck`) endpoint that has a high rate of producing errors. Execute the following script to launch the test.

Code Listing 82: Test fruits API resilience

```
$ .\juice-shop-api-resilience-test.ps1
```

```
Request 1: Better luck next time
Request 2: Lucky
Request 3: Better luck next time
Request 4: Better luck next time
Request 5: Better luck next time
Request 6: Better luck next time
```

In my test run, only two out of 50 requests succeeded (just 4 percent). Let's now implement the resiliency strategies that we discussed previously. The following specification of virtual service adds a timeout period of 30 seconds to each request received from the client. Within this period, Envoy makes 10 attempts to fetch results from the **juice-shop** API service with a waiting period of 3 seconds between each attempt. We have also specified the failure conditions as the value of the **retryOn** key on which the retry policy kicks in.

Code Listing 83: Virtual service with retry specification

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: juice-shop-api-vservice
  namespace: micro-shake-factory
spec:
  hosts:
    - "juice-shop.istio-succinctly.io"
  gateways:
    - juice-shop-api-gateway
  http:
    - route:
        - destination:
            host: juice-shop-api-service
            port:
              number: 80
      timeout: 30s
      retries:
        attempts: 10
        perTryTimeout: 3s
        retryOn: "5xx,connect-failure,refused-stream"
```

Next, we add a destination rule that detects and ejects the outliers from the load balancer pool. The following specification limits the number of parallel requests to the **juice-shop** API service, and on receiving a single error (should not be 1 in real-world scenarios) ejects the endpoint from the load balancer pool for a minimum of three minutes. Envoy decides on the endpoints to eject and the ones to bring back in the pool every second, and it can eject up to 100 percent of the endpoints from the pool.

Code Listing 84: Resilience specification

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: juice-shop-api-destination-rule
  namespace: micro-shake-factory
spec:
  host: juice-shop-api-service
  trafficPolicy:
    connectionPool:
      http:
        http1MaxPendingRequests: 1
        maxRequestsPerConnection: 1
    outlierDetection:
      consecutiveErrors: 1
      interval: 1s
      baseEjectionTime: 3m
      maxEjectionPercent: 100
```

This new specification for the **juice-shop** API service is present in the **juice-shop-api-dr-vs-gw-resilience.yml** file. Let's apply this configuration to our cluster with the following command.

Code Listing 85: Apply fruits API with resilience policy

```
$ kubectl apply -f juice-shop-api-dr-vs-gw-resilience.yml

gateway.networking.istio.io/juice-shop-api-gateway unchanged
virtualservice.networking.istio.io/juice-shop-api-vservice configured
destinationrule.networking.istio.io/juice-shop-api-destination-rule created
```

Let's execute the same test that we previously executed to verify the effectiveness of this policy.

Code Listing 86: Testing fruits API resilience

```
$ ./juice-shop-api-resilience-test.ps1

Request 1: Lucky
Request 2: Lucky
Request 3: Lucky
Request 4: Lucky
Request 5: Lucky
Request 6: Lucky
```

When you execute the test this time, you will notice that the tests execute much slower due to the request timeout policy and retries happening in the mesh. On this run, only 2 out of 50 requests resulted in errors for me. That is a significant improvement without any code alterations. Note that these policies are not universal and are scoped to each client of the **juice** API service, as the service may fault for a single client while still functioning for others.

Pattern 5: Mirroring

Traffic mirroring or dark launch is a model of deployment by which a new version of a service is deployed to production without affecting the traffic to the existing service. To enforce dark launch, Istio duplicates the requests to the primary service and sends them asynchronously to the service under test without waiting for the secondary service to respond. The asynchronous mirroring of traffic ensures that the critical path of live traffic remains unaffected. As an exercise, refer to the mirror setting specification in this chapter and try to route traffic in parallel to both versions of the **fruits** API service.

Summary

In this chapter, we demonstrated the full power of Istio's networking APIs. Although there are several traffic management features in Istio, you should follow an incremental approach to applying them to your services. You should find the primary traffic management challenge in your microservices and fix it with an appropriate policy. Reiterate the process so that you incrementally gain confidence with stability of the mesh.

Chapter 6 Mixer Policies

We discussed earlier that the Istio Mixer component receives configurations from the user, mixes them with different data sources, and then dispatches them to different channels. Mixer supports the following three categories of policies:

- **Precondition checking:** Checks conditions such as ACLs and authorization.
- **Quota management:** Checks conditions such as rate limits.
- **Telemetry collection:** Checks on metrics, logs, and traces.

Given the two completely different areas of responsibilities of Mixer, it is deployed as two different pods, each of which either enforces policy or collects and transmits telemetry. Execute the following command to view the pods that make up Mixer.

Code Listing 87: Get Mixer pods

```
$ kubectl get pods -n istio-system -l istio=mixer
```

NAME	READY	STATUS	RESTARTS	AGE
istio-policy-769664fcf7-8vtz8	2/2	Running	60	6d1h
istio-telemetry-577c6f5b8c-8qb7x	2/2	Running	63	6d1h

Before we discuss the Mixer policies in detail, let's discuss the high-level architecture of Mixer. The following diagram illustrates the components of Istio and their interactions.

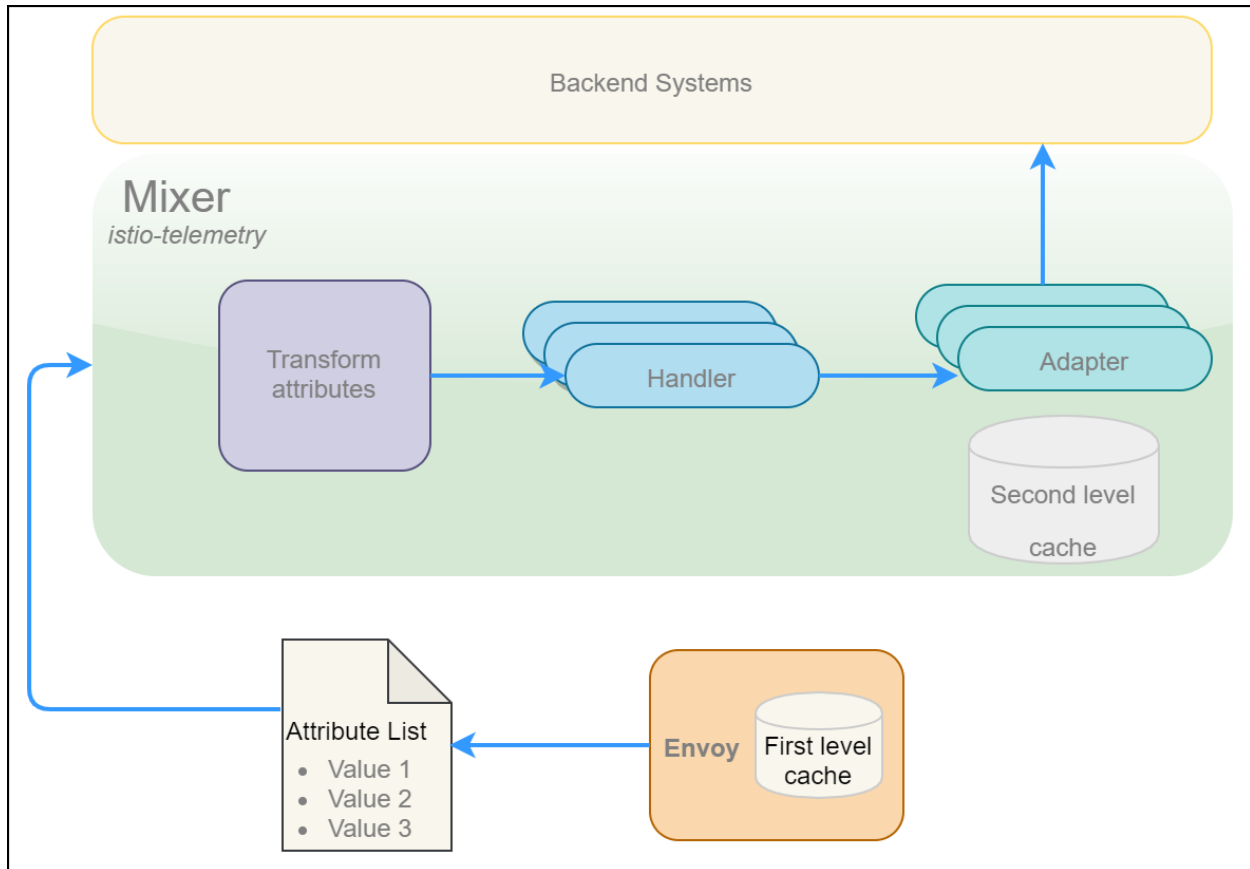


Figure 15: Mixer architecture

The path of the request varies depending on whether the request is for evaluating a policy or for persisting telemetry. Let's study the two paths that a request can take.

Policy control flow

Service proxies send request details in the form of attributes to Mixer before each request to check preconditions, and again afterwards to report telemetry. Mixer determines whether the request complies with preconditions and/or quota restrictions by invoking the appropriate handler, which is a logical representation of an adapter. After receiving the check results from Mixer once, the service proxies cache the result locally, which enables them to serve many subsequent requests without referring each one to Mixer. Mixer also maintains a cache so that it can refer to it when other Envoy instances send a request for evaluating the same policy.



Note: Mixer is a highly available component. It is stateless and has several active replicas processing requests together. It also uses caching and buffering techniques and a robust underlying design that attempts to ensure an uptime of 99.999 percent.

Let's briefly discuss what attributes are. Attributes are a collection of typed key-value pairs that describe the request traffic and its context. Attributes are not only used for making decisions in Mixer, but are also logged after the completion of every request by Mixer (see telemetry control flow). There are a finite but exhaustive number of attributes, which are documented [here](#). The attributes are primarily produced by the proxies, but can also be generated by the adapters.

Telemetry control flow

For each request received, Envoy captures an array of attributes of the request. Envoy continuously processes these attributes and sends them asynchronously to Mixer's **Report** API. After a buffer of reports is captured by Envoy, they are pushed at once to Mixer, which transforms the attributes and pushes them to the backend using handlers (which compile to an adapter).

In this chapter, we will focus on the policy enforcement aspect of Mixer, and discuss observability in a later chapter. Mixer's policy rules are a little complex; therefore, their configuration is split into six major parts: the adapter, the handler, the instance, the rule, the quota spec, and the quota spec binding. Let's discuss all these components one by one.

Adapter

Adapters are independent components of Mixer that can perform a logical operation. As you can see in the previous architecture diagram, while some adapters connect Mixer to backend systems, others perform a complete functionality within themselves, such as **quota** and **whitelist**. You can find the complete list of adapters [here](#).

By default, Mixer binaries contain several adapters, such as **denier**, **OPA**, and **statsd**. This ties the vendor releases of these adapters with Mixer releases. Mixer is now moving to an out-of-process adapter model in which the adapter process executes separately from the Mixer process. Mixer communicates with such adapters using gRPC, and it is therefore not influenced by crashes of adapter processes.

Handler

A handler is an instance of the adapter that is configured with operation parameters. To draw an analogy with object-oriented programming, if the adapter is a class, then a handler is an object of the class, and the operational parameters are properties defined in the class whose values will be specified in the handler. These properties in handlers are called attributes.

The following handler specification limits the number of requests to the **fruits** API service to one request every five seconds.

Code Listing 88: Handler specification

```
apiVersion: config.istio.io/v1alpha2
kind: handler
```

```

metadata:
  name: throttle-handler
  namespace: istio-system
spec:
  compiledAdapter: memquota
  params:
    quotas:
      - name: request-quota-instance.instance.istio-system
        maxAmount: 500
        validDuration: 1s
        overrides:
          - dimensions:
              destination: fruits-api-service
              maxAmount: 1
              validDuration: 5s

```

The previous handler specification will apply a throttle limit of 500 requests per second on all other client requests that get mapped to this handler. This throttling limit gets down to one request every five seconds if the requests are sent to the **fruits** API service. Note that the value of the **maxAmount** attribute is configured through another resource named **QuotaSpec**. Think of it as the maximum currency available to a client to spend; **QuotaSpec** will specify how much each request will cost.

Instance

An instance is an object that contains request data. We know that adapters require configuration data that is supplied to them through attributes of handlers. Instances map the attributes of the incoming request to the values that are passed to the handler. Instances support the mapping of attributes through attribute expressions.

Attribute expressions are configurations written in Configuration Expression Language (CEXL), which is a subset of GoLang expressions. You can read about the syntax of CEXL expressions [here](#). The following configuration demonstrates how you can configure an instance and shows an attribute expression in use.

Code Listing 89: Instance specification

```

apiVersion: "config.istio.io/v1alpha2"
kind: instance
metadata:
  name: request-quota-instance
  namespace: istio-system
spec:
  compiledTemplate: quota
  params:
    dimensions:
      destination: destination.service.name | "unknown"

```

The instance specified in the previous listing declares an attribute named **destination** and conditionally sets its value to one of the properties specified through the attribute expression. You can see that the **dimension** attribute expression consists of canonical attributes (**destination.service.name**) that form the value of the attribute. To view all the supported attributes, visit [this link](#).

Rule

Rules map the handlers and instances with each other, and specify the condition in which a handler will be invoked with values from an instance. Each rule specifies a match condition and actions in the form of mapping between handlers and instances when the match condition evaluates to **true**. In a rule, you can specify the empty match condition or set the value of the match key to **true**, which would make the rule perform the actions in all conditions.

Code Listing 90: Rule specification

```
apiVersion: config.istio.io/v1alpha2
kind: rule
metadata:
  name: quota-rule
  namespace: istio-system
spec:
  match: match(request.headers["user-agent"], "curl*")
  actions:
    - handler: throttle-handler
      instances:
        - request-quota-instance
```

In the previous listing, we specified that the handler named **throttle-handler** should be invoked with data stored in the instance named **request-quota-instance** when the user agent specified in the request headers is **curl**. The match value **curl*** ensures that the match conditions succeed evaluation regardless of the version of the curl utility used.

QuotaSpec

Some services are more resource intensive than others, so you may want finer-grained control over the traffic targeted to those services. Note that in the handler specification, we configured the attribute **maxAmount** to some value. The **QuotaSpec** allows you to specify how much each request will deplete from your quota. For this sample, we will make each request to pay one unit.

Code Listing 91: QuotaSpec specification

```
apiVersion: config.istio.io/v1alpha2
kind: QuotaSpec
metadata:
  name: request-count
  namespace: istio-system
```



```
spec:
  rules:
    - quotas:
      - charge: 1
        quota: request-quota-instance
```

The schema in the previous listing charges one unit for each request sent to the instance **request-quota-instance**.

QuotaSpecBinding

This is the glue that binds the policies with actual services in the mesh. You may have noticed that the namespace of all the policies is **istio-system**, whereas the actual service might be present in a different namespace. The **QuotaSpecBinding** helps attach the services with policies across namespaces. The following policy binds the request-count quota specification with the **fruits** API service.

Code Listing 92: QuotaSpecBinding specification

```
apiVersion: config.istio.io/v1alpha2
kind: QuotaSpecBinding
metadata:
  name: request-count
  namespace: micro-shake-factory
spec:
  quotaSpecs:
    - name: request-count
      namespace: istio-system
  services:
    - name: fruits-api-service
      namespace: micro-shake-factory
```

These are all the building blocks that we need to write any Mixer policy. Note that different policies have their own specifications, and we can't cover all of them here. However, with the understanding of the building blocks, you will be able to configure any policy with relative ease. Let's now test what we have built so far.

Testing the Mixer policy

Like before, delete any resources from the **micro-shake-factory** namespace so that they don't interfere with our demo.



Note: The required files for this demo are present in the [GitHub repository at Policies/Chapter 6](#).

Let's first create the **fruits** API service and its associated virtual service and gateway by executing the following command.

Code Listing 93: Create fruits API service

```
$ kubectl apply -f fruits-api.yml -f fruits-api-vs-gw.yml

namespace/micro-shake-factory created
deployment.apps/fruits-api-deployment-v1 created
service/fruits-api-service created
gateway.networking.istio.io/fruits-api-gateway created
virtualservice.networking.istio.io/fruits-api-vservice created
```

Now, aggregate all the configurations that you declared previously in a single YAML file and execute the specification with the following command.

Code Listing 94: Apply throttle policy

```
$ kubectl apply -f curl-throttle-policy.yml

handler.config.istio.io/throttle-handler created
instance.config.istio.io/request-quota-instance created
rule.config.istio.io/quota-rule created
quotaspec.config.istio.io/request-count created
quotaspecbinding.config.istio.io/request-count created
```

To test the throttling limit, we have prepared a test script that dispatches two requests to the **fruits** API service, one after the other. The first request is sent from the curl utility, which succeeds only once every five seconds, and the other from the Invoke-WebRequest PowerShell cmdlet, which succeeds every time. This process is repeated five times in a batch, after which the process waits for five seconds before repeating itself 10 times for a total of 100 requests (50 each from curl and Invoke-WebRequest). Execute the following PowerShell test script.

Code Listing 95: Test fruits API throttling

```
$ .\curl-throttle-test.ps1

Request 1.1
{"ver":"1","fruit":"Mango"}
{"ver":"1","fruit":"Mango"}

Request 1.2
RESOURCE_EXHAUSTED:Quota is exhausted for: request-quota-instance
{"ver":"1","fruit":"Mango"}
```

In the previous listing, you can see the partial output of data generated by the test. You can see that for the first request, both the curl and PowerShell requests succeeded, whereas the next request succeeded only for the request that originated from PowerShell.

Delete the policies that you just created by executing the following command, which removes the restrictions from your service.

Code Listing 96: Delete policy

```
$ kubectl delete -f curl-throttle-policy.yml

handler.config.istio.io "throttle-handler" deleted
instance.config.istio.io "request-quota-instance" deleted
rule.config.istio.io "quota-rule" deleted
quotaspec.config.istio.io "request-count" deleted
quotaspecbinding.config.istio.io "request-count" deleted
```

We encourage you to experiment with other Mixer policies and find out for yourself where you can use them.

Summary

In this chapter, we discussed the responsibilities of Mixer and how the Mixer policies work. We defined the various components that formed a Mixer policy and applied it to our mesh. An important point to remember is that Mixer v2 is positioned to move much of Mixer's functionality to Envoy proxy. This move of functionality will reduce latency and avoid the chatty interaction between the proxy and Mixer. In the next chapter, we will discuss the various approaches to securing the services on an Istio mesh.

Chapter 7 Security

In the world of microservices, security can quickly become unwieldy with poor implementation and management. Traditionally, security policies have revolved around the network identity of a service, which is its IP address. However, IP addresses of workloads are ephemeral in Kubernetes and any other container orchestrator, so Istio solves this problem by decoupling the identity of a workload from the host.

Istio relies on the name rather than the network IP address of the resource as its identity. The Citadel component of Istio implements the SPIFFE framework to issue identities to services.

SPIFFE requires that identities of resources should be URIs in the format:

spiffe://domain/universally-unique-identifier (UUID), where the domain is the root of trust of the identity. Going by the specification, the identity created by Citadel for a service follows this format: **spiffe://cluster.local/ns/<namespace-name>/sa/<service-account-name>** (the service account name is **default** unless configured otherwise). This guarantees a unique identity of service in the mesh. Citadel's definition of identity varies with the host platform. For example, in Kubernetes, the identity is a Kubernetes service account; in GCP, it is a GCP service account; and in AWS, it is an AWS IAM role/user.

Next, following the SPIFFE specifications, Citadel encodes the service identity in an X.509 certificate, also known as a SPIFFE Verifiable Identity Document (SVID), which can be verified to prove identity of service. Finally, the SPIFFE specifications demand an API for issuing and retrieving the SVIDs. Citadel implements this requirement by acting as a certificate authority (CA). The Citadel agents on mesh nodes send a certificate signing request (CSR) to the CA when they detect a new workload, and the CA returns a new SVID to it. The certificate has a lifetime of a couple of hours, after which the process is repeated, which ensures that any attack can't continue for a long period of time.

Let's briefly discuss how the service-to-service communication takes place. Istio uses mTLS, which requires the client and server to exchange certificates before initiating communication. In Istio, Envoy presents and receives the SVID of the service that it wants to communicate with. This ensures that the connection is authorized before actual communication.

The concept of access control ultimately boils down to two components: *authentication* and *authorization*. In Istio, a service can authenticate itself using an X.509 certificate issued by a trusted CA (Citadel). If the receiver can validate the certificate, the identity stored in the certificate is considered authenticated for further operations.

Authorization, on the other hand, determines whether an authenticated entity may perform an action on the resource. In Istio, the authorization of communication between services is configured with role-based access control (RBAC) policies.

Istio provides the following key security features to the services in the mesh:

- Automatic generation and rotation of SVID certificates.
- Service authentication with mTLS.
- Transport-level authorization using an RBAC system.
- Automatic traffic auditing and monitoring.

- Support for external authentication protocols like OpenID.
- TLS termination at the gateway.

We will explore these features in detail in this chapter. First, let's discuss the architecture of Istio's security management feature.

Security architecture

The following diagram shows the various components involved in Istio's security architecture.

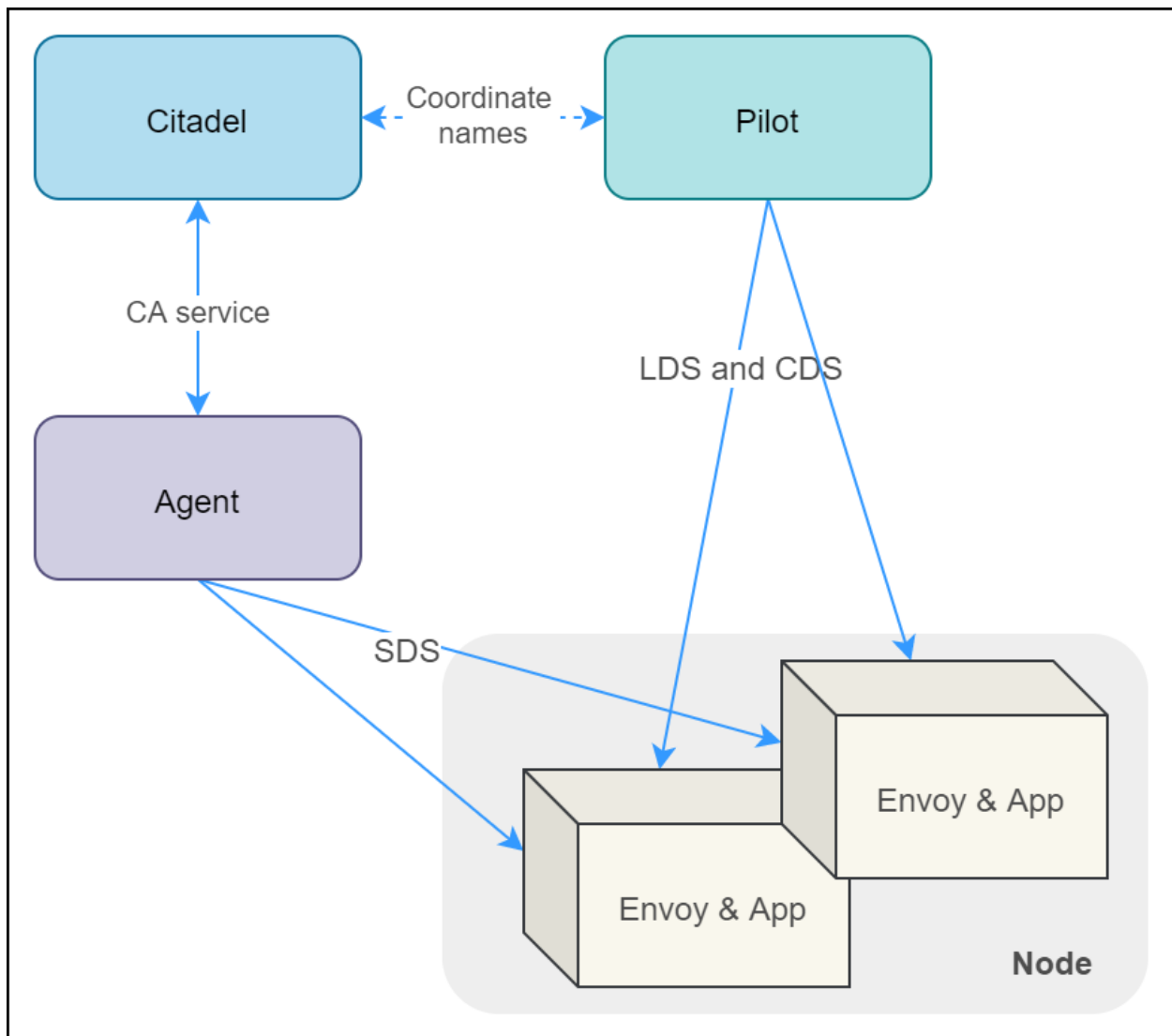


Figure 16: Istio security architecture

Let's discuss the roles of the various components in the architecture:

- **Citadel:** Watches the Kubernetes **apiserver**, generates SVID for every service, and stores them as secrets. It fetches names of services from Pilot (remember, Istio is platform agnostic) to generate SVIDs.
- **Node agent:** The Citadel-trusted agent on the node that coordinates certificate issuance and rotation between Envoy and Citadel. Node agent uses SDS API to configure secrets for Envoy (Envoy only understands SDS) and helps validate the authority of the certificate received by Envoy during mTLS communication flow, which Envoy itself is incapable of.
- **Envoy:** Uses SVID certificates to communicate with other services.
- **Pilot:** Sends topology and listener information to Envoy using LDS and CDS API, including the name of certificates that Envoy should use to communicate with each service.

You can revisit Chapter 1 where we discussed the various components shown in the previous diagram in detail. Let's proceed to discuss the authentication and authorization facets of Istio.

Istio authentication

Istio has two different sets of configurations for provisioning authentication and authorization policies. The authorization policies require prior provisioning of authentication policies. Let's discuss the authentication policies of Istio.

Transport authentication: mTLS

Istio supports transport authentication using mutual TLS (mTLS), which requires both the client and the server to own identity in the form of TLS certificates (TLS is modern SSL) issued by a mutually trusted CA (Citadel). Authentication is provisioned at the service and namespace level using the authentication policy object in Istio. At the level of the cluster, you can provision a default authentication policy using the mesh policy object. The following diagram illustrates how Pilot delivers identity to services in the mesh.

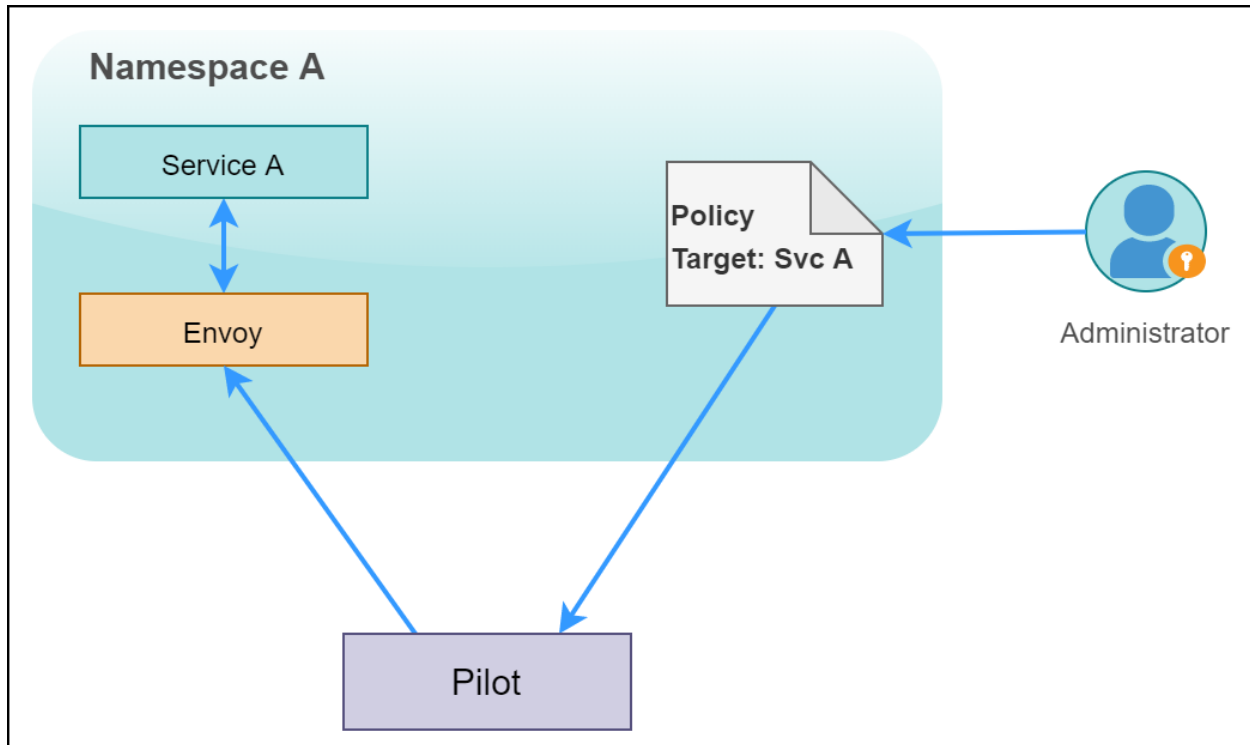


Figure 17: Pilot applying the policy to the service

The responsibility to watch and propagate authentication policies to Envoy lies with Pilot. Pilot translates changes in policies to appropriate LDS and CDS calls and ensures that the Envoy proxy sidecars are updated with the latest authentication mechanisms, such as JWT or path, to the certificate to use for mTLS.

For authentication purposes, the client proxies immediately start following the new authentication policy applied by Pilot. For example, for origin authentication, Envoy will attach a JWT token to each request, and for mTLS, Envoy will ensure that the request is using the appropriate destination rule to make the TLS connection with the server.



Tip: Istio supports a lenient permissive mode of mTLS, which helps smooth the migration of existing services to Istio mTLS. In this mode, Istio allows requests to succeed with and without mTLS.

We will now look into the schema of an authentication policy, which determines the authentication method and the principal that gets generated after the authentication process. The authentication policy can authenticate one or both of the following:

- **peer**: Credentials of the service sending the request.
- **origin**: Credentials of the user/service account/device from where the request originates.

After validation, you need to choose whether you want to choose the identity of the peer (default) or the origin as the identity principal. In the following schema, you can set the kind of policy as **meshPolicy** that will make the policy applicable to the entire mesh.

Code Listing 97: Authentication policy specification

```
apiVersion: authentication.istio.io/v1alpha1
kind: Policy
metadata:
  name:
  namespace:
spec:
  targets:
  - name:
    ports:
    - number:
  peers:
  - mtls:
    mode: PERMISSIVE | STRICT
  peerIsOptional:
  origins:
  - jwt:
    issuer:
    audiences:
    jwksUri:
    jwks:
    jwtHeaders:
    jwtParams:
    triggerRules:
    - excludedPaths:
    - includedPaths:
  originIsOptional:
  principalBinding: USE_ORIGIN | USE_PEER
```

The keys described in this specification support the following values:

- **targets: TargetSelector[]**
An array of targets on which the policy is applied. If empty, the policy is applied to all workloads in the namespace. The array elements consist of the following keys:
 - **name***: Short name of the service from the service registry.
 - **ports**: Port exposed by the service as **PortSelector** type. Empty value matches all the ports of the service.
- **peers: PeerAuthenticationMethod[]**
An ordered list of authentication methods used for peer authentication. An empty value means that peer authentication is not required. Currently, this array supports the key of type **mtls** to denote that mTLS should be used. The value of the **mtls** key is another key of type **mode**, which can be one of the following:
 - **STRICT**: Client certificate must be presented over the TLS channel.
 - **PERMISSIVE**: Client certificate can be omitted, and connection can be either plaintext or use TLS.
- **peerIsOptional: bool**
If set to true, it will override the decision of the peer authentication policy and accept the request from the peer authentication perspective.

- **origins: OriginAuthenticationMethod[]**
An ordered list of authentication methods used for origin authentication. If all rules are skipped because of unmatched requests, the policy is ignored. An empty value means that origin authentication is not required. The type **OriginAuthenticationMethod** currently only supports a key named **jwt** of type **Jwt**, which means that JSON Web Token (JWT) is required for authentication of the client. The **Jwt** type consists of the following fields:
 - **issuer**: The value denotes the principal that issued the token.
 - **audiences**: String array of JWT audiences that can access the token.
 - **jwkUri**: URL of the provider's public key set that can be used to validate the signature of JWT.
 - **jwtKeys**: JSON Web Key Set of public keys used to validate the signature of JWT.
 - **jwtHeaders**: String array of headers where JWT tokens will be available if the JWT tokens are part of the request header.
 - **jwtParams**: String array of query parameters where JWT tokens will be available if JWT tokens are part of the query string.
 - **triggerRule**: An array of type **TriggerRule** that specifies whether the received JWT token should be validated. If the request matches any trigger rule, then validation is not carried out for the request, which helps exclude certain paths, such as health checks, from validation. The type **TriggerRule** contains two keys of type **StringMatch[]**:
 - **excludedPaths**: List of request paths that should be excluded from checks.
 - **includedPaths**: List of request paths that should be included for checks.
- **originIsOptional: bool**
If set to **true**, it will override the decision of origin authentication policy and accept the request from the origin authentication perspective.
- **principalBinding: PrincipalBinding**
This key can take one of two values: **USE_PEER** (default) or **USE_ORIGIN** to determine whether peer authentication or origin authentication will set the identity of the request.

Let's start locking down the **fruits** API service by applying a few authentication policies. Before you proceed, delete the namespace **micro-shake-factory** to start with a clean canvas.

Applying transport authentication

Deploy the **fruits** API service and a gateway that makes this service accessible with the following command. Remember to replace the **&** operator with **&&** on the Bash terminal.

Code Listing 98: Creating fruits API

```
$ kubectl apply -f fruits-api.yml & kubectl apply -f fruits-api-vs-gw.yml

namespace/micro-shake-factory created
deployment.apps/fruits-api-deployment-v1 created
service/fruits-api-service created
gateway.networking.istio.io/fruits-api-gateway created
virtualservice.networking.istio.io/fruits-api-vservice created
```

Currently, the **fruits** API service responds over HTTP. Let's alter the behavior and apply a blanket mTLS-only policy over the entire mesh using the mesh policy.

Code Listing 99: Mesh policy specification

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

The policy in the previous listing will ensure that any communication in the mesh requires mTLS. After applying this policy, any new service created within the mesh will automatically have mTLS configured with Citadel-managed certificates. The name of the policy is **default** so that it overwrites the default policy that is automatically provisioned on the installation of Istio. Also, the policy mentions no targets for blanket application on all services. Apply the policy using the following command.

Code Listing 100: Creating default mesh policy

```
$ kubectl apply -f default-mesh-policy.yml

meshpolicy.authentication.istio.io/default configured
```

The mesh-wide policy configures services in the mesh to communicate with each other over the TLS-encrypted channel. However, now you won't be able to use the **curl** command to send a request to the service, because that communication is still using clear text.



Tip: To view all authentication policies active in your cluster, use this command: `kubectl get policies.authentication.istio.io -A`. For mesh policies, use this command: `kubectl get meshpolicies.authentication.istio.io`.

To configure mTLS on the client, we will create a destination rule to enforce TLS on the client to the service channel.

Code Listing 101: Destination rule specification

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: default-destination-rule
  namespace: istio-system
spec:
  host: "*.local"
  trafficPolicy:
    tls:
      mode: ISTIO_MUTUAL
```

The wildcard match `*.local` makes the policy in previous listing applicable to all services in the mesh.



Note: Ideally, you should enforce mTLS mesh policy on the mesh. However, for scenarios where you need mTLS for specific services, use namespace or service-specific policies. For details, refer to [this link](#).

Apply the destination rule that you just created with the following command.

Code Listing 102: Applying default destination rule

```
$ kubectl apply -f default-dr.yml
```

```
destinationrule.networking.istio.io/default-destination-rule created
```

Use curl to send a request to the **fruits** API service, which will succeed this time. There are a few other types of requests that require your attention based on your use case:

- **Non-Istio service to Istio service:** Since non-Istio services can't access the authentication infrastructure of Istio, this communication will fail. You can whitelist services or endpoints from Istio authentication policies if such communication is a requirement, so that authentication policies do not apply to the communication.
- **Istio service to Kubernetes apiserver:** Since **apiserver** does not have an Envoy proxy to manage TLS, you will have to create a destination rule with the **trafficPolicy.tls.mode** key set to **DISABLE** for the host **kubernetes.default.svc.cluster.local**.
- **Istio service to non-Istio service:** The problem and solution is the same as the one for Kubernetes **apiserver** that we just discussed.

Depending on your use case, you can whitelist a namespace, a service, or even a port (specificity of a rule is specified as value of the **spec.target** key) from mTLS using authentication policies (have the same syntax as a mesh policy), which have higher priority than mesh policies.



Tip: Istio CLI has a useful command that can list the authentication policies applied to a service and its health: `istioctl authn tls-check <name of pod> <FQDN of service on cluster>`.

Let's now discuss the process to grant access only authenticated users of the **fruits** API service.

Applying origin authentication

Istio supports origin (end-user or device) authentication with JSON Web Tokens (JWT). It allows you to validate nearly all the fields of a JWT token received and the identity provider. JWT is an industry-standard token format, and therefore, this feature helps Istio support integration with OpenID connect providers such as Auth0, Google Auth, and Keycloak.

Let's enable JWT authentication on **fruits** API service to allow only authenticated requests to reach the service.

Code Listing 103: JWT authentication specification

```
apiVersion: authentication.istio.io/v1alpha1
kind: Policy
metadata:
  name: fruits-api-authn-policy
  namespace: micro-shake-factory
spec:
  targets:
    - name: fruits-api-service
  origins:
    - jwt:
        issuer: "istio-succinctly.io"
        jwksUri: "https://raw.githubusercontent.com/Istio-
Succinctly/Policies/master/Static/jwks.json"
    principalBinding: USE_ORIGIN
```

We have published a JSON Web Key Set (JWKS) endpoint at [this link](#), which primarily serves the key ID (**kid**), algorithm (**alg**), and RSA public key (**n**) to the client. Istio uses the RSA public key to validate the issuer and key ID of the JWT token that it receives from the request. Apply this policy to the **fruits** API service by using the following command.

Code Listing 104: Applying authentication policy

```
$ kubectl apply -f fruits-api-authn-policy.yml

policy.authentication.istio.io/fruits-api-authn-policy configured
```

After this policy is applied to our service, any request that does not provide user credentials will return HTTP error 401-unauthorized.



Note: After applying the authentication policy, the mesh policy will be ignored entirely for the *fruits* API service. Note that the previous authentication policy did not specify a peer authentication configuration, but the mesh policy did. However, this policy will disable the mTLS requirement enforced by the mesh policy.

For ease of use, we have generated and stored a JWT token with a very long expiry duration (October 5, 2100), which is [available here](#). If you decode the token using a script or online tools such as [JWT.io](#), you will find that the token has the following header and payload.

Code Listing 105: Decoded JWT token

```
# Header
{
  "alg": "RS256",
  "typ": "JWT",
  "kid": "Ow4Yuv9ACv5y6Hpt4XeICn7UjGVUd21"
}
```

```
# Payload
{
  "iss": "istio-succinctly.io",
  "name": "Istio",
  "role": "learner",
  "iat": 1516239022,
  "exp": 4126377167
}
```

You can try to invoke an API endpoint of the **fruits** API service with and without (or wrong) JWT, as follows. Remember to replace the placeholder **{TOKEN}** with the actual token value.

Code Listing 106: Invoke API with auth header

```
$ curl http://localhost/api/fruits/special -H "Authorization: Bearer {TOKEN}"
-H "Host: fruits.istio-succinctly.io"

{"ver": "1", "fruit": "Mango"}
```

You can also restrict the audience of a service or an endpoint by requiring users to have a certain role claim in the JWT token. Let's discuss the authorization policy of Istio in the next section.

Authorization policy

With an authorization policy, we can use identities to enforce access control on service-to-service communication channels. Istio describes the authorization policy using an RBAC system called **ClusterRbacConfig**. It is a custom resource definition (CRD) that is installed as part of Istio, and you can only have a single instance of the **ClusterRbacConfig** object in a mesh. You require two objects to configure RBAC in Istio:

- **ServiceRole**: Sets of actions that can be performed on a set of services by a principal with a role.
- **ServiceRoleBinding**: Assigns roles to a principal, which in Kubernetes is a **ServiceAccount**.

The **ClusterRbacConfig** object allows the incremental rollout of authorization policy, as just like authentication policies, its indiscriminate application may disrupt operations. The RBAC policy supports four modes:

- **OFF**: No RBAC required for communication. This is the default mode.
- **ON**: RBAC is required for service communication in the mesh.
- **ON_WITH_INCLUSION**: RBAC is required for communicating with services in the namespaces specified in the inclusion field.
- **ON_WITH_EXCLUSION**: RBAC is required for communicating with services outside the namespaces specified in the exclusion field.

For incremental rollout, enable RBAC with **ON_WITH_INCLUSION** mode, and after full rollout, switch it to **ON** mode.

Let's briefly discuss the schema of the resources required for configuring authorization in Istio. The following schema illustrates the keys in the **ClusterRbacConfig** policy.

Code Listing 107: Cluster RBAC config specification

```
apiVersion: rbac.istio.io/v1alpha1
kind: ClusterRbacConfig
metadata:
  name: default
  namespace: istio-system
spec:
  mode:
  inclusion:
    namespaces:
    services:
  exclusion:
    namespaces:
    services:
```

We will now go through the values supported by the keys in the spec field of the schema:

- **mode: Mode**
As we previously discussed, the value of can be one of **ON**, **OFF**, **ON_WITH_INCLUSION**, or **ON_WITH_EXCLUSION**.
- **inclusion: Target**
The value contains the names of services or namespaces on which RBAC policy is enforced. The key contains two nested keys named **namespaces** and **services**, both of which are of type **string[]**, in which you supply a list of names of namespaces and services.
- **exclusion: Target**
The supported value is similar to that of the inclusion key. However, the specified namespaces and services are excluded from RBAC policies.

The following is the schema of the **ServiceRole** object.

Code Listing 108: Service role specification

```
apiVersion: rbac.istio.io/v1alpha1
kind: ServiceRole
metadata:
  name:
  namespace:
spec:
  rules:
    - services:
    - paths:
    - methods:
```

```
constraints:
- key:
  values:
```

The following are the supported values of the individual keys in the **spec** field:

- **rules*: AccessRule[]**
The attributes determine the permissions that a role has. Each rule is composed of the following keys:
 - **services***: String array of names of the services on which the rule is applied. The names may be matched by prefix, suffix, or through wildcards. Therefore, you should use service FQDNs as names to avoid a collision.
 - **paths**: A list of HTTP paths or gRPC methods to which the rule is applicable. It helps make the application of rules granular. For example, by setting the value to **/pathA** and specifying the service name as **serviceX.default.cluster.local**, you can restrict RBAC to just a path in the service. It supports the same matches as services.
 - **methods**: List of HTTP request verbs such as **GET**, **POST**, and **PUT** on which the rule is applicable. This field is not applicable to TCP services and supports only **POST** for gRPC services. It supports the wildcard character ***** to match any verb.
 - **constraints**: The value specifies additional constraints to restrict the applicability of rules, such as applying constraints only on a version of the service. It consists of two nested keys: **key**, to specify the key of the constraint; and **values**, which is a string array to specify the desired value of the key.

Finally, let's investigate the schema of the object that binds the service role with request attributes: **ServiceRoleBinding**.

Code Listing 109: Service role binding specification

```
apiVersion: rbac.istio.io/v1alpha1
kind: ServiceRoleBinding
metadata:
  name:
  namespace:
spec:
  subjects:
  - user:
  - properties:
    source.namespace:
  roleRef:
    kind: ServiceRole
    name:
```

Again, we will dive straight into the nested keys of the **spec** field:

- **subjects*: Subject[]**
The value determines the list of objects that are assigned to a **ServiceRole** object. The type **Subject** defines an identity that is either a user or a collection identified by a set of properties. It consists of the following keys:
 - **user**: The name or ID of the user represented by the subject.

- **properties**: A `map<string,string>` object that identifies the properties of the subject.
- **roleRef*: RoleRef**
The value refers to a Kubernetes **ServiceRole** object. The type **RoleRef** consists of the following keys:
 - **kind***: The value references the kind of role to which the subject is mapped. Currently, it only supports the value **ServiceRole**.
 - **name***: The name of the underlying service role.

Let's apply an authorization policy to the **fruits** API service so that only authorized requests can access the service. We will configure the policy such that the principal with role **learner** gets viewing (**GET** requests only) rights to the API. Revisit the JWT token payload that we used for authentication earlier. The body of the token contains a key named **role** with the value **learner**, which ultimately translates to the role claim of the principal.

First, we will enable RBAC with our service in the inclusion list of the policy using the following specification.

Code Listing 110: RBAC config policy

```
apiVersion: rbac.istio.io/v1alpha1
kind: RbacConfig
metadata:
  name: default
  namespace: istio-system
spec:
  mode: ON_WITH_INCLUSION
  inclusion:
    services:
      - fruits-api-service.micro-shake-factory.svc.cluster.local
```

Next, we need to describe the roles that exist for our service, which currently is just one: **fruits-api-viewer**. So, we will declare a service role for it with the following specification.

Code Listing 111: Service role policy

```
apiVersion: rbac.istio.io/v1alpha1
kind: ServiceRole
metadata:
  name: fruits-api-viewer
  namespace: micro-shake-factory
spec:
  rules:
    - services:
        - fruits-api-service.micro-shake-factory.svc.cluster.local
      methods:
        - GET
```

Finally, we will configure the binding that maps the role to the service account used by the **fruits** API service.

Code Listing 112: Service role binding policy

```
apiVersion: rbac.istio.io/v1alpha1
kind: ServiceRoleBinding
metadata:
  name: fruits-api-viewer-binding
  namespace: micro-shake-factory
spec:
  subjects:
    - properties:
        request.auth.claims[role]: "learner"
  roleRef:
    kind: ServiceRole
    name: fruits-api-viewer
```

The previous policy will match any principal that has its claim named **role** set to the value **learner**. Let's apply this policy to our cluster with the following command.

Code Listing 113: Apply fruits API authz specification

```
$ kubectl apply -f fruits-api-authz-policy.yml

rbacconfig.rbac.istio.io/default created
servicerole.rbac.istio.io/fruits-api-viewer created
servicerolebinding.rbac.istio.io/fruits-api-viewer-binding created
```

To test the policy, execute the same command that you previously used to test the authentication policy. Use the same token that you used earlier.

Code Listing 114: Testing RBAC in fruits API

```
$ curl http://localhost/api/fruits/special -H "Authorization: Bearer {TOKEN}"
-H "Host: fruits.istio-succinctly.io"

{"ver": "1", "fruit": "Mango"}
```

There is another token in the repository [located here](#), which only differs from the previous token in the role attribute (verify it at [JWT.io](#)). The role property of this token is set to value **juice-hater**, which has no business accessing our API. When you execute the same command with this token set in the authorization header, you will receive the following response.

Code Listing 115: RBAC error

```
RBAC: access denied
```

Having taken care of in-cluster security and application security, we will now discuss how you can secure the channel over which the clients access your services.

Istio Authorization Policy

Istio 1.4 added support for [AuthorizationPolicy](#), which will supersede the RBAC policy that we just discussed. One of the major issues with the RBAC policy is that you can specify the Kubernetes deployment to which you want to apply an RBAC policy by specifying the service that it belongs to. However, a single deployment can be a part of multiple services, which can lead to confusion with the application of RBAC policies.

The **AuthorizationPolicy** CRD allows you to specify access policy for an individual deployment, and Istio 1.6 and later will only support **AuthorizationPolicy** objects instead of **ClusterRbacConfig**, **ServiceRole**, and **ServiceRoleBinding** objects. However, there are only semantic differences between the current RBAC policy specifications and the **AuthorizationPolicy**-based specifications. Istio 1.4 added an experimental migration tool (`istioctl experimental authz convert`) that can help migrate v1alpha1 policies to v1beta1 policies. You can read more about the policy migration path [here](#).

Securing ingress

Clear text transmission of network traffic is not only a bad security practice, but it also impacts your site's search rankings and gets flagged in modern browsers such as Google Chrome. SSL or its successor TLS ensures that network communication is secure. TLS protocol allows the connection between a client and server to be encrypted, which helps ensure that a third party can't read or tamper the data in transit (a man-in-the-middle attack).

Istio gateway is the single resource that connects services within the cluster to the services outside the cluster. The gateway and virtual service split ensures that L4 (transport) and L5 (session) routing properties, which are managed by the gateway, remain decoupled from L7 (application) routing concerns, which are managed by the virtual service.

The Istio gateway allows you to terminate ingress TLS traffic, redirect non-TLS traffic to TLS ports, pass the TLS traffic to a backend service, and implement mutual TLS. Remember that after terminating TLS at the gateway, you can use the mTLS channel within the cluster to secure the communication between services in the mesh.

Simple TLS

The simple TLS implementation allows you to set up TLS on the gateway so that traffic is served only over HTTPS. To configure TLS for ingress traffic, we configure a private key and a public certificate that the gateway will use. Here is the highly simplified version of how TLS works. The encryption public key is published by the server in the form of a certificate that is trusted by the client since the certificate is issued by a trusted certificate authority (CA). The server uses the private key to initiate a handshake with the client, which is validated by the client using the public key. After the verification, a symmetric session key is created, and it is used by both the client and the server to encrypt the bi-directional network data.

There are two approaches to serving certificates through the gateway:

- Mount a Kubernetes volume containing certificates and keys to the proxy running the **istio-ingressgateway** service.
- Use Citadel to generate certificates for the gateway and manage certificate distribution through secret discovery service (SDS). The procedure for implementing this approach is [documented here](#).

For the following series of examples, we will deploy the **juice-shop** API service to our mesh. Use the following command to deploy the services to the mesh.

Code Listing 116: Creating juice-shop API

```
$ kubectl apply -f juice-shop-api.yml -f juice-shop-api-vs-gw.yml

namespace/micro-shake-factory unchanged
deployment.apps/juice-shop-api-deployment created
service/juice-shop-api-service created
gateway.networking.istio.io/juice-shop-api-gateway created
virtualservice.networking.istio.io/juice-shop-api-vservice created
```

After applying the previous policy, you can access the service on HTTP. Let's find out the default location from where the ingress gateway picks its certificates by executing the following command. Remember to replace the placeholder **<pod-identifier>** with the actual value.

Code Listing 117: Describe ingress gateway pod

```
$ kubectl describe po/istio-ingressgateway-<pod-identifier> -n istio-system

Name: istio-ingressgateway-7c6f8fd795-whq94
Containers:
  istio-proxy:
    Mounts:
      /etc/certs from istio-certs (ro)
      /etc/istio/ingressgateway-ca-certs from ingressgateway-ca-certs (ro)
      /etc/istio/ingressgateway-certs from ingressgateway-certs (ro)
      /var/run/secrets/kubernetes.io/serviceaccount from istio-ingressgateway-service-account-token-bj7md (ro)
Volumes:
  istio-certs:
    Type: Secret (a volume populated by a Secret)
    SecretName: istio.istio-ingressgateway-service-account
    Optional: true
  ingressgateway-certs:
    Type: Secret (a volume populated by a Secret)
    SecretName: istio-ingressgateway-certs
    Optional: true
  ingressgateway-ca-certs:
    Type: Secret (a volume populated by a Secret)
    SecretName: istio-ingressgateway-ca-certs
    Optional: true
  istio-ingressgateway-service-account-token-bj7md:
    Type: Secret (a volume populated by a Secret)
```

```
SecretName: istio-ingressgateway-service-account-token-bj7md
Optional:   false
```

The output from the previous listing is very informative. We now know that the gateway will load certificates from the volume **ingressgateway-certs**, which maps to a secret named **istio-ingressgateway-certs**. We will require certificates issued by a local CA to configure both the client (**curl**) and the gateway for which you can use the [OpenSSL](#) library. For convenience, we have added the certificate (**juice-shop.istio-succinctly.crt**) and key (**juice-shop.istio-succinctly.key**) in the [code repository](#) of this book that you can easily apply. A helper script in the certificates folder named **cert-generator.sh** can help you regenerate the certificates if needed.

Execute the following command to create the **istio-ingressgateway-certs** secret in your cluster.

Code Listing 118: Create secret

```
$ kubectl create -n istio-system secret tls istio-ingressgateway-certs --key
certificates/juice-shop.istio-succinctly.key --cert certificates/juice-
shop.istio-succinctly.crt

secret/istio-ingressgateway-certs created
```

The previous command will create a secret containing two data fields named **tls.crt** and **tls.key** containing certificate details that we can now use to configure the gateway as follows.

Code Listing 119: Ingress gateway rule simple TLS

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: juice-shop-api-gateway
  namespace: micro-shake-factory
spec:
  selector:
    istio: ingressgateway
  servers:
    - port:
        number: 443
        name: https-juice-shop-api-gateway
        protocol: HTTPS
      tls:
        mode: SIMPLE
        serverCertificate: /etc/istio/ingressgateway-certs/tls.crt
        privateKey: /etc/istio/ingressgateway-certs/tls.key
      hosts:
        - juice-shop.istio-succinctly.io
```

With the policy described in the previous listing, we opened port 443 on our gateway and restricted the ingress protocol to HTTPS. We also defined the TLS specification and supplied the path to the key and certificate to the appropriate TLS settings. Now, apply this specification to the cluster using the following command.

Code Listing 120: Update ingress gateway

```
$ kubectl apply -f juice-shop-api-tls-vs-gw.yml

gateway.networking.istio.io/juice-shop-api-gateway configured
virtualservice.networking.istio.io/juice-shop-api-vservice unchanged
```

To test the updated gateway, we need to map the hostname and port specified in the certificate to the IP address and port of the gateway through the `--resolve` flag of the `curl` command. Also, we need to pass the CA certificate that we generated to the `curl` command so that it can validate the certificate that it receives from the gateway.

Code Listing 121: Access juice-shop API

```
$ curl -H "Host:juice-shop.istio-succinctly.io" --resolve juice-shop.istio-
succinctly.io:443:127.0.0.1 --cacert ca-chain.cert.pem https://juice-
shop.istio-succinctly.io:443/api/juice-shop/hello

Welcome to the Juice Shop!
```

Let's discuss how you can enforce HTTP to HTTPS redirection next.

HTTPS redirect

We can configure the gateway to direct all requests to the HTTP endpoint to be redirected to its HTTPS endpoint, thereby enforcing TLS on all network traffic. The following configuration enforces the redirection constraint. Notice the `tls.httpsRedirect` key in the following specification, whose value is set to `true`.

Code Listing 122: Ingress gateway HTTPS redirect

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: juice-shop-api-gateway
  namespace: micro-shake-factory
spec:
  selector:
    istio: ingressgateway
  servers:
    - port:
        number: 80
        name: http-juice-shop-api-gateway
        protocol: HTTP
      hosts:
        - juice-shop.istio-succinctly.io
```

```

    tls:
      httpsRedirect: true
  - port:
      number: 443
      name: https-juice-shop-api-gateway
      protocol: HTTPS
    tls:
      mode: SIMPLE
      serverCertificate: /etc/istio/ingressgateway-certs/tls.crt
      privateKey: /etc/istio/ingressgateway-certs/tls.key
    hosts:
      - juice-shop.istio-succinctly.io

```

Apply this configuration to the cluster using the following command.

Code Listing 123: Apply ingress rule

```

$ kubectl apply -f juice-shop-api-tls-only-vs-gw.yml

gateway.networking.istio.io/juice-shop-api-gateway configured
virtualservice.networking.istio.io/juice-shop-api-vservice unchanged

```

Let's try sending a request to the HTTP endpoint of the service now.

Code Listing 124: Access juice-shop API

```

$ curl http://localhost/api/juice-shop/hello -H "Host: juice-shop.istio-succinctly.io" -v

*   Trying::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 80 (#0)
> GET /api/juice-shop/hello HTTP/1.1
> Host: juice-shop.istio-succinctly.io
> User-Agent: curl/7.55.1
> Accept: */*
>
< HTTP/1.1 301 Moved Permanently
< location: https://juice-shop.istio-succinctly.io/api/juice-shop/hello
< date: Mon, 07 Oct 2019 08:10:07 GMT
< server: istio-envoy
< content-length: 0
<
* Connection #0 to host localhost left intact

```

The output from the execution of the previous command shows that the gateway responded with HTTP 301 (redirect) response. With this policy in place, we can expect the ingress traffic through our gateway to always be encrypted.

Mutual TLS

In the simple TLS implementation, the client requests the server to present its public certificate that the client subsequently verifies with the trusted CA. With mutual TLS, both parties exchange their public certificates and verify them at their end before processing the actual request. For establishing mTLS, the client needs to provide a CA-issued certificate to the gateway for verification. The procedure to upload the certificate is the same as the one we followed for simple TLS. We will create another Kubernetes secret using the following command (revisit the output of [Code Listing 119](#)).

Code Listing 125: Create Kubernetes secret

```
$ kubectl create -n istio-system secret generic istio-ingressgateway-ca-certs
--from-file=certificates/istio-succinctly.ca.crt

secret/istio-ingressgateway-ca-certs created
```

We will now update the gateway resource to use the CA certificate and change the protocol to mutual TLS.

Code Listing 126: Mutual TLS gateway

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: juice-shop-api-gateway
  namespace: micro-shake-factory
spec:
  selector:
    istio: ingressgateway
  servers:
    - port:
        number: 443
        name: https-juice-shop-api-gateway
        protocol: HTTPS
      tls:
        mode: MUTUAL
        serverCertificate: /etc/istio/ingressgateway-certs/tls.crt
        privateKey: /etc/istio/ingressgateway-certs/tls.key
        caCertificates: /etc/istio/ingressgateway-ca-certs/istio-succinctly.ca.crt
      hosts:
        - juice-shop.istio-succinctly.io
```

Apply the previous specification using the following command.

Code Listing 127: Apply mTLS gateway rule

```
$ kubectl apply -f juice-shop-api-mTLS-vs-gw.yml

gateway.networking.istio.io/juice-shop-api-gateway configured
```

```
virtualservice.networking.istio.io/juice-shop-api-vservice unchanged
```

To test this configuration, in addition to the CA certificate, the `curl` command needs the client's certificate and its private key in the `--cert` and `--key` parameters, respectively.

Code Listing 128: Call juice-shop API with mTLS

```
$ curl -HHost:juice-shop.istio-succinctly.io --resolve juice-shop.istio-succinctly.io:443:127.0.0.1 --cacert istio-succinctly.ca.crt --cert client.juice-shop.istio-succinctly.crt --key client.juice-shop.istio-succinctly.key https://juice-shop.istio-succinctly.io:443/api/juice-shop/hello
```

```
Welcome to the Juice Shop!
```

Finally, you can also establish a client-to-destination service TLS channel, which is not affected by the gateway. We will discuss this workflow next.

TLS passthrough

With TLS passthrough the gateway doesn't validate the TLS certificate presented by the client, nor does it participate in the HTTPS handshake. It simply directs the request to the destination service and allows the service to manage validation of the request and security of the channel. By default, the `juice-shop` API service supports secure communication over port 443 in addition to port 80. For this demo, we will establish connection between the client and the service on the secure channel available with the service.

The following gateway configuration will route the traffic without any modification to the destination service when the TLS mode is set to **PASSTHROUGH**.

Code Listing 129: TLS passthrough specification

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: juice-shop-api-gateway
  namespace: micro-shake-factory
spec:
  selector:
    istio: ingressgateway
  servers:
    - port:
        number: 443
        name: https-juice-shop-api-gateway
        protocol: HTTPS
      tls:
        mode: PASSTHROUGH
      hosts:
        - juice-shop.istio-succinctly.io
```


In addition to the gateway, we will also make a small change to the virtual service, which will now only receive traffic at port 443, instead of 80 from the gateway.

Code Listing 130: Virtual service passthrough specification

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: juice-shop-api-vservice
  namespace: micro-shake-factory
spec:
  hosts:
    - juice-shop.istio-succinctly.io
  gateways:
    - juice-shop-api-gateway
  tls:
    - match:
        - port: 443
          sniHosts:
            - juice-shop.istio-succinctly.io
      route:
        - destination:
            host: juice-shop-api-service
            port:
              number: 443
```

The change that we made to the virtual service from its previous version is the substitution of the **HTTPRoute** field with the **TLSRoute** field. Also, we specified the match condition with an SNI hostname and a route to which traffic is directed on a successful match. You can see that specification for a **TLSRoute** entity is similar to the specification for an **HTTPRoute** entity.



Note: The *sniHosts* key of the virtual service configuration supports wildcard character *** as well.

TLS passthrough should only be used when TLS responsibilities can't be offloaded to the gateway, or when an application is operating in an unsecure mesh. Since the network packets can't be decrypted by the gateway, this policy limits the routing capabilities of the virtual service to only L4 attributes of a request.

Let's apply the policy to our mesh with the following command.

Code Listing 131: Update gateway and virtual service for TLS passthrough

```
$ kubectl apply -f juice-shop-api-tls-pass-vs-gw.yml

gateway.networking.istio.io/juice-shop-api-gateway configured
virtualservice.networking.istio.io/juice-shop-api-vservice configured
```

With the policies in place, we are now ready to test the service by executing the following command.

Code Listing 132: Access juice-shop API

```
$ curl https://juice-shop.istio-succinctly.io:443/api/juice-shop/hello --  
resolve juice-shop.istio-succinctly.io:443:127.0.0.1 -k
```

```
Welcome to the Juice Shop!
```

The output of the previous command shows that the service was able to successfully respond to our request. We have now covered all the critical aspects of security of services operating in the mesh.

Cleanup

To ensure that our mesh is ready for subsequent demos, we will remove the mesh policy and the default destination rule that we created previously. We will also delete the namespace **micro-shake-factory** to remove the services and Istio objects from the mesh. Execute the following command to restore the cluster to a clean state.

Code Listing 133: Cleanup cluster

```
$ kubectl delete namespace micro-shake-factory & kubectl delete -f default-  
mesh-policy.yml & kubectl delete -f kubectl apply -f default-dr.yml
```

The previous command will delete the **micro-shake-factory** namespace and its resources, as well as remove the default mesh policy and the default destination rule.

Summary

Identity and access control form the boundary of service mesh, allowing only desired traffic to reach the services. Istio supports multiple authentication and authorization policies as well as channel security through mTLS. Istio allows you to incrementally adopt service-to-service mTLS and RBAC policies without affecting existing services. In the next chapter, we will discuss the monitoring and tracking features of Istio that bring transparency to operations of services on the mesh.

Chapter 8 Observability

Istio and Kubernetes are effective for operationalizing dense microservices deployments on a fixed number of nodes. When there are many services running on a platform, it is important for operations to detect failures and degradation of services on the platform in real time. As we saw through demos earlier, Istio adds a high level of network resiliency to the system, and a monitoring system might not even detect the degradation of a subset of services. The Envoy proxy of Istio sits in the hot path of the request, and therefore, it can generate high-fidelity instrumentation that can be used to detect and fix issues that can otherwise stay undetected in a system.

In a distributed system, a request may travel through multiple microservices before returning a response to the user. Therefore, it is critical for microservices to implement distributed tracing with a correlation identifier so that the request flow graph of any request can be traced. Istio implements the [OpenTracing](#) standard. Istio also supports integration with popular visualization tools such as [Prometheus](#), [Grafana](#), and [Kiali](#), which can help developers and operations visualize the state of the mesh.

Mixer is the key component that collects and compiles telemetry that it receives from the Envoy proxies. The sidecar proxies buffer telemetry at runtime and periodically flush the aggregated data to Mixer. Previously, we discussed that Mixer interfaces with various backends using adapters. Mixer can push the telemetry that it receives from proxies in parallel to multiple telemetry adapters and wait for them to complete. Therefore, you can provision multiple adapters of the same type (such as telemetry adapters that supply data to different backends) at the same time.

Telemetry is a generic term that encompasses three components: metrics, logs, and traces. As we saw earlier, telemetry is periodically sent in the form of attributes by the sidecars to Mixer through its service named `istio-telemetry`, which is present in the `istio-system` namespace.

Metrics

Envoy automatically collects some of the common metrics, such as error rate, total request count, and response size. Mixer adapters interested in receiving metrics are required to consume data in the form of a [metric adapter template](#). Istio aggregates metrics at two levels:

- **Envoy proxy:** Operators can configure the metrics that they want each proxy instance to generate. Some examples of metrics at this level are upstream successful request count, requests completed, connection errors, and upstream failed request count.
- **Service:** Operators can configure proxy to garner service-level metrics to monitor service-to-service communication in terms of latency, traffic, errors, and saturation.

By default, the metrics are exported to Prometheus by Mixer, but you can configure other backends as well. For configuring a new metric that Envoy should capture, you need three types of resources: an instance, a handler, and rules.

Let's create these resources to produce a new metric that counts each request twice. The first step is to create a metric that defines what attributes it contains. In Istio, this collection of attributes is called dimensions. The following listing is the declaration of our metric as an instance.

Code Listing 134: Instance specification

```
apiVersion: config.istio.io/v1alpha2
kind: instance
metadata:
  name: doublerequestcount
  namespace: istio-system
spec:
  compiledTemplate: metric
  params:
    value: "2" # count each request twice
    dimensions:
      reporter: conditional((context.reporter.kind | "inbound") ==
"outbound", "client", "server")
      source: source.workload.name | "unknown"
      destination: destination.workload.name | "unknown"
      message: '"Counting requests twice!'"
      monitored_resource_type: '"UNSPECIFIED"'
```

The previous listing defines a new schema for our new metrics called **doublerequestcount**. Because we have set the value of the key **params.value** to 2, on every request from the client when Istio requests for an instance of **doublerequestcount**, it will receive two objects, thereby recording the metric twice on each request. We have used CEXL expressions to supply values to each of the properties in the **dimensions** key.

Next, we need to specify the handler to which this metric should be sent. We will now define a Prometheus handler with the following specification.

Code Listing 135: Handler specification

```
apiVersion: config.istio.io/v1alpha2
kind: handler
metadata:
  name: doublehandler
  namespace: istio-system
spec:
  compiledAdapter: prometheus
  params:
    metrics:
      - name: double_request_count # Prometheus metric name
        instance_name: doublerequestcount.instance.istio-system
        kind: COUNTER
        label_names:
          - reporter
          - source
          - destination
```

- message

The previous specification defines a handler named **doublehandler**. It also defines a new Prometheus metric named **double_request_count**, which will show up in Prometheus as **istio_double_request_count**. The custom metric has four labels that map to the metric that we previously created.

Finally, we require a rule to bind the metric to the handler and specify the conditions on which the rule should be triggered. The following is the specification for the rule.

Code Listing 136: Rule specification

```
apiVersion: config.istio.io/v1alpha2
kind: rule
metadata:
  name: doubleprom
  namespace: istio-system
spec:
  actions:
    - handler: doublehandler
      instances: [doublerequestcount]
```

Now that we have the configurations ready, let's clear the namespace for our demo by executing the following command.

Code Listing 137: Delete namespace

```
$ kubectl delete namespace micro-shake-factory

namespace "micro-shake-factory" deleted
```

Now, let's bring back our APIs by executing the following commands.

Code Listing 138: Create fruits API

```
$ kubectl apply -f fruits-api-all.yml -f juice-shop-api-all.yml

namespace/micro-shake-factory created
deployment.apps/fruits-api-deployment-v1 created
service/fruits-api-service created
virtualservice.networking.istio.io/fruits-api-vservice created
namespace/micro-shake-factory unchanged
deployment.apps/juice-shop-api-deployment created
service/juice-shop-api-service created
gateway.networking.istio.io/juice-shop-api-gateway created
virtualservice.networking.istio.io/juice-shop-api-vservice created
```

We are now ready to enable the metrics by executing the following command.

Code Listing 139: Create a custom metric

```
$ kubectl apply -f double-request-metric.yml
```

```
instance.config.istio.io/doublerequestcount created
handler.config.istio.io/doublehandler created
rule.config.istio.io/doubleprom created
```

Send a few requests to the **juice-shop** API service so that our metrics start showing up. Since the metrics are buffered by Envoy, it would take some time to have our metrics processed. Once the metrics are processed, they will start appearing in the Prometheus UI, which is already installed in your cluster. Let's port forward the Prometheus service by executing the following command so that we can access it from our browser.

Code Listing 140: Port forward Prometheus

```
$ kubectl -n istio-system port-forward service/prometheus 9090 9090
```

Navigate to the Prometheus graph endpoint at <https://localhost:9090/graph> and enter the metric name for the records to show up in the grid: **istio_double_request_count**. You can also write Prometheus-supported queries such as **sum(istio_double_request_count)** to operate with the metric.

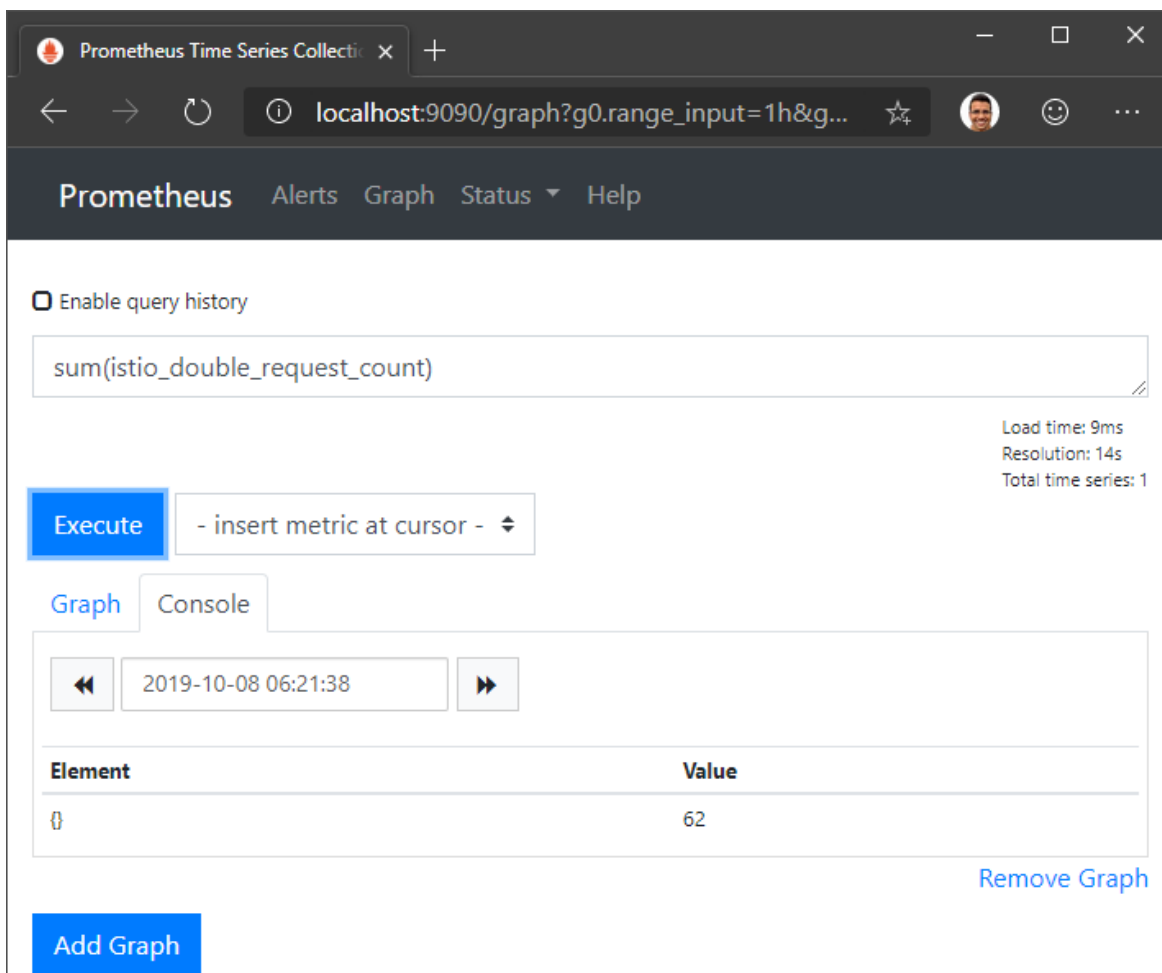


Figure 18: Prometheus UI

Prometheus is a time-series database and visualization tool. For an overall view of the mesh, Istio supports the Grafana visualization add-on. Grafana is a popular open-source metrics visualization tool that can be used to query, analyze, and alert on metrics. The Istio deployment of Grafana consists of some common dashboards. Grafana is dependent on Prometheus for metrics.

To enable Grafana, set the parameter `--set grafana.enabled=true` during the Helm installation of Istio. Next, send some traffic to the mesh and query for the dashboard endpoint by executing the following endpoint. Note that any dashboard endpoint that we discuss in this book will remain the same across clusters (yours or ours) unless configured otherwise, but you should know how you can retrieve the service ports and not remember a bunch of magic numbers.

Code Listing 141: Get Grafana service

```
$ kubectl -n istio-system get svc grafana
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
grafana	ClusterIP	10.108.18.200	<none>	3000/TCP	5m50s

The next operation to perform is to simply forward port 3000 of the Grafana service to a port on the localhost by executing the following command.

Code Listing 142: Port-forward Grafana

```
$ kubectl port-forward -n istio-system svc/grafana 3000:3000
Forwarding from 127.0.0.1:3000 -> 3000
Forwarding from [::1]:3000 -> 3000
```

This command will make the Grafana dashboard available on <http://localhost:3000>. Navigate to the portal and select **Istio Service Dashboard** from the list of dashboards, which will allow you to monitor incoming traffic in real time.

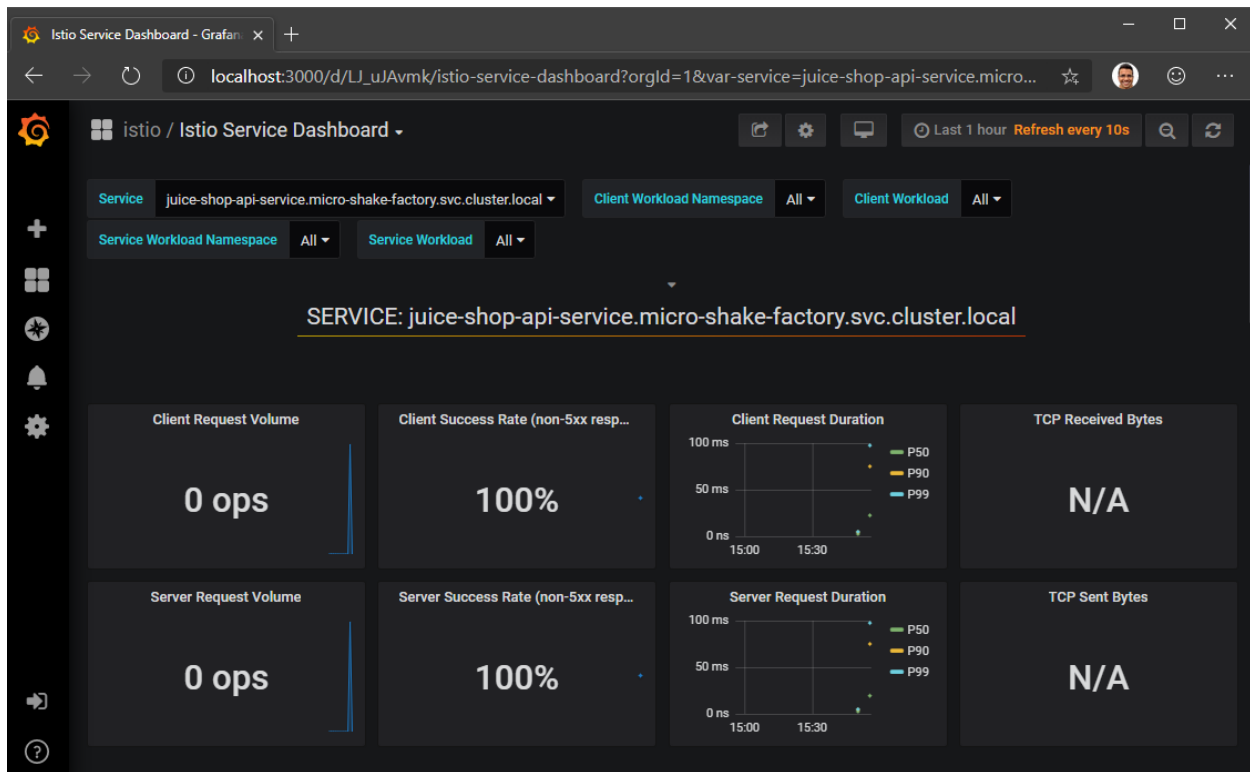


Figure 19: Grafana UI

Some of the other pre-provisioned dashboards allow you to visualize the overall mesh, individual workloads, and individual services.

Traces

Distributed tracing is key to getting deep insight from the service mesh. Two popular open-source tracing systems, [Jaeger](#) and [Zipkin](#), are bundled in the Istio release package. Both solutions follow the OpenTracing standards. To enable tracing, you only have to set the parameter `--set tracing.enabled=true` in the Istio Helm install options. To enable Zipkin, you also need to set the parameter `--set tracing.provider=zipkin` during installation; otherwise, Jaeger will be installed by default.

Here is a brief overview of how distributed OpenTracing works. The span is the primary building block of a distributed trace, which represents an individual unit of work in a distributed system. The application is responsible for creating spans and sharing them with the OpenTracing engine. A span contains the beginning and end time of the operation, the name of the operation, and the logs generated within the operation. If the service invokes any other service, it will be responsible for propagating the trace context to the other service, which will create a new span and repeat the process. With spans and trace context, the OpenTracing engine generates a trace. A trace can show you the complete path of a request along with the time, duration, and other information. Each span has its own ID and a parent ID, which is the ID of the trace. These identifiers should be propagated between the services for correlation.

Istio manages communication with the OpenTracing engine and generates request tracing headers if they don't exist. If it receives a request with the headers populated, it doesn't generate them again and treats the trace as an in-progress trace. Istio relies on the following headers for distributed tracing:

- **x-request-id**
- **x-b3-traceid**
- **x-b3-spanid**
- **x-b3-parentspanid**
- **x-b3-sampled**
- **x-b3-flags**
- **x-ot-span-context**

If the service invokes another service, it must propagate the trace headers with the request so that Istio is able to correlate the upstream request with the incoming request to the service. Our demo application APIs are instrumented to propagate the headers across the services, and therefore, we will be able to trace a request end to end. If you list the services running under the **istio-system** namespace, you will find a service named **tracing** that is responsible for interfacing with the tracing provider Jaeger or Zipkin and fetching logs from the application. Let's execute the following operations with our service to generate logs.

Code Listing 143: Access juice-shop API

```
$ curl -H "Host: juice-shop.istio-succinctly.io" -H "Content-Type: application/json" -H "country: au" -d "{\"fruits\": [\"kiwi\", \"mandarin\"]}" http://localhost/api/juice-shop/blender

$ curl -H "Host: juice-shop.istio-succinctly.io" -H "Content-Type: application/json" -H "country: au" -d "{\"fruits\": [\"kiwi\", \"mango\"]}" http://localhost/api/juice-shop/blender

$ curl http://localhost/api/juice-shop/hello -H 'Host: juice-shop.istio-succinctly.io'

$ curl http://localhost/api/juice-shop/exoticFruits -H 'Host: juice-shop.istio-succinctly.io'
```

To find out the port on which the telemetry service exposes the telemetry data, execute the following command, which returns all the ports exposed by the service.

Code Listing 144: Get tracing port name

```
$ kubectl get svc/tracing -n istio-system -o jsonpath='{.spec.ports}'

'[map[name:http-query port:80 protocol:TCP targetPort:16686]]'
```

We know what we should do next. Forward the port 80 (80 is mapped to 16686 on the container) of the service to the localhost using the following command.

Code Listing 145: Port-forward tracing

```
$ kubectl port-forward svc/tracing -n istio-system 8081:80
```

Forwarding from 127.0.0.1:8081 -> 16686
Forwarding from [::1]:8081 -> 16686

After executing the previous command, you can navigate to <http://localhost:8081> to view the Jaeger dashboard. At this time, you might or might not see all the requests in the logs, since to save processing overheads, Istio records just 1 percent of requests by default. There are two approaches that you can take to adjust this value:

- **Update the PILOT_TRACE_SAMPLING environment variable:** Execute the command `kubectl -n istio-system edit deploy istio-pilot` and update the value of the `PILOT_TRACING_SAMPLING` variable to a number between 1 and 100. This change will affect all the services in the mesh.
- **Set x-envoy-force-trace request header:** Set the `x-envoy-force-trace` request header for all the requests that need to be traced.

Execute the previous requests to the `juice-shop` API service again after setting the request header, and then visit the Jaeger dashboard. In the dashboard, you will now be able to look up traces generated by each service as follows.

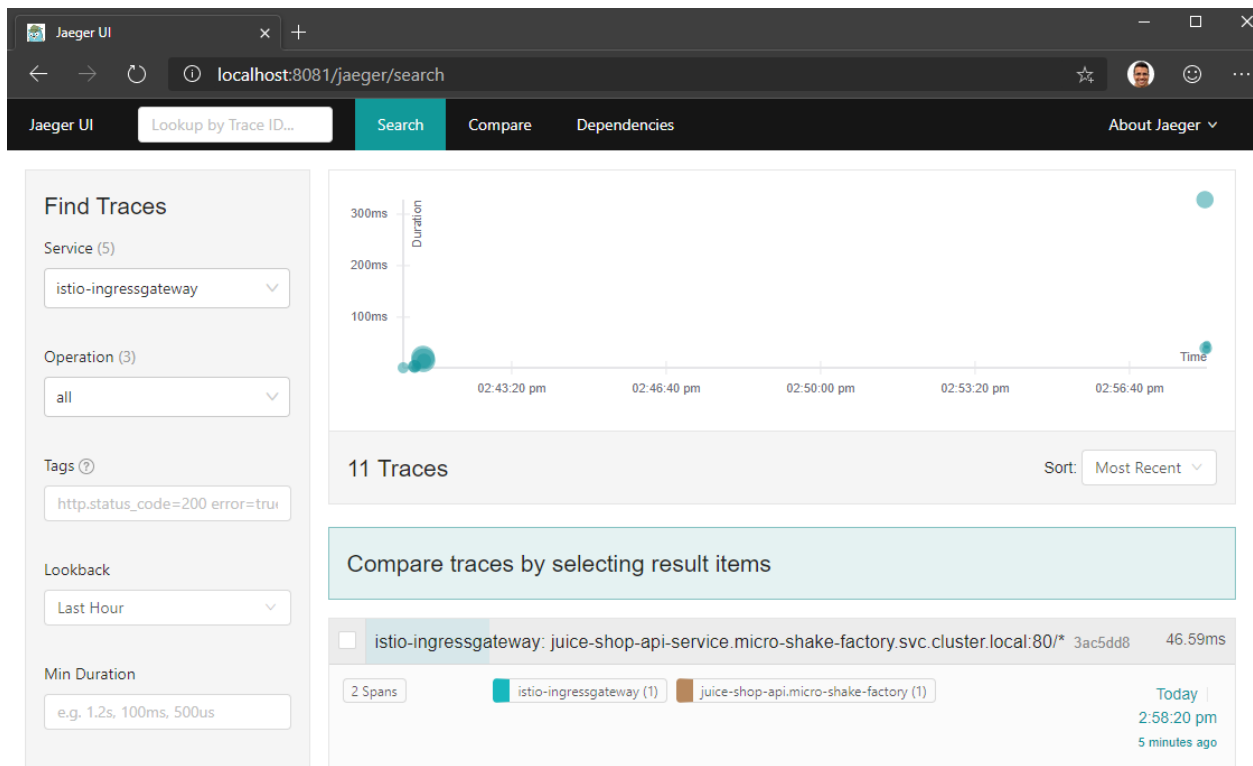


Figure 20: Jaeger dashboard

If you click on a trace, you will be able to see the call graph of the request. The following screenshot illustrates one such trace where the request is tracked across the services.

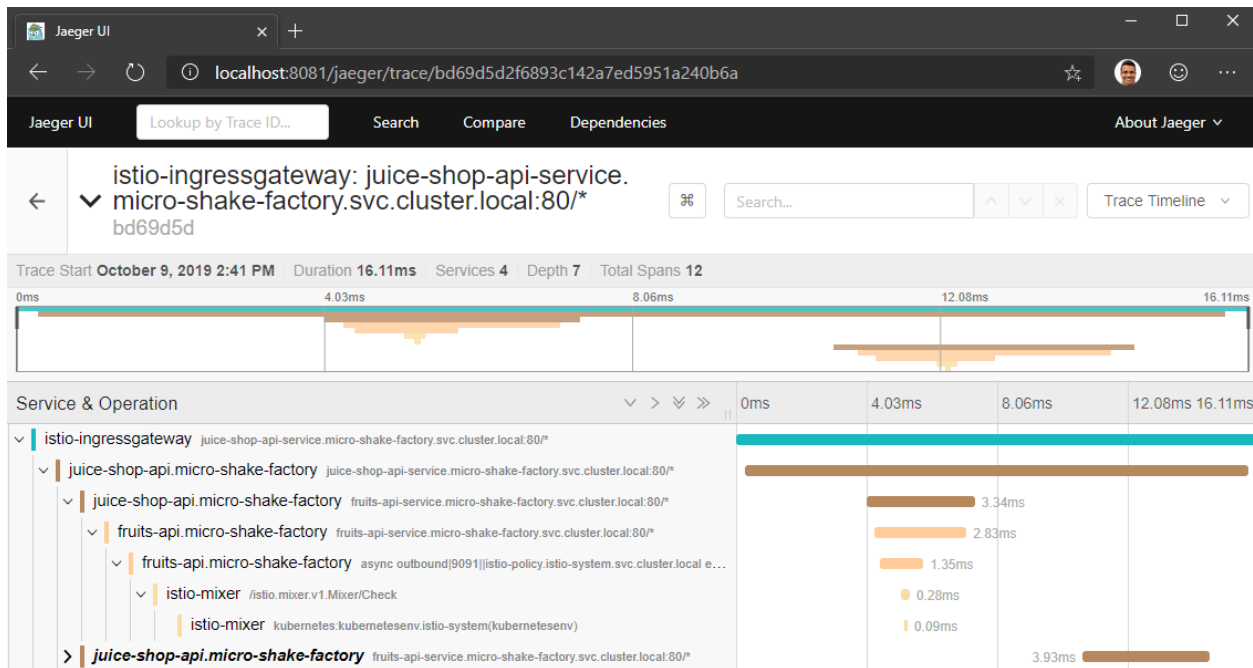


Figure 21: Jaeger trace results

By clicking on the **Dependencies** tab on the dashboard, you can also see the dependency graph of the services whose traces are available to Jaeger.

Logs

Logs are crucial for recording the application state details when an error occurs. Arguably the most popular choice for recording logs in a distributed environment is [Fluentd](#), which can be enabled in the mesh through the Fluentd Mixer adapter. Mixer has a built-in adapter named **logentry** that generates a stream of logs. Fluentd, and many other logging backends rely on **logentry** to gather logs. Out of the many approaches for configuring Fluentd, let's look at one of the easiest approaches using the following workflow:

1. Create a **logentry** instance to generate a stream of logs.
2. Configure a **Fluentd** handler to collect the logs and pass them to a Fluentd daemon running on the cluster.
3. Create a rule that binds the **logentry** instance to the **Fluentd** handler and configures the logging level.

The following is the specification for a log entry instance, which will generate a log stream for logs of severity level *information* and above.

Code Listing 146: Log entry specification

```
apiVersion: config.istio.io/v1alpha2
kind: logentry
metadata:
  name: istiolog
```

```

  namespace: istio-system
spec:
  severity: '"info"'
  timestamp: request.time
  variables:
    source: source.labels["app"] | source.service | "unknown"
    user: source.user | "unknown"
    destination: destination.labels["app"] | destination.service | "unknown"
    responseCode: response.code | 0
    responseSize: response.size | 0
    latency: response.duration | "0ms"
  monitored_resource_type: '"UNSPECIFIED"'

```

Next, we will create the **Fluentd** handler, which will pass the logs to the **Fluentd** daemon running at **localhost:24224** in the cluster.

Code Listing 147: Fluentd specification

```

apiVersion: config.istio.io/v1alpha2
kind: fluentd
metadata:
  name: handler
  namespace: istio-system
spec:
  address: "localhost:24224"
  integerDuration: n

```

Finally, the following rule will bind the previous two objects together.

Code Listing 148: Rule specification

```

apiVersion: config.istio.io/v1alpha2
kind: rule
metadata:
  name: istiologtofluentd
  namespace: istio-system
spec:
  match: "true" # Match for all requests
  actions:
    - handler: handler.fluentd
      instances:
        - istiolog.logentry

```

You can tweak the severity level and the match rules to suit your needs. The application of this policy is dependent on how Fluentd is deployed on your cluster, and therefore, you should refer to the official documentation on the [topic here](#) for further guidance.

Mesh visualization

Visualizing how your mesh is performing at runtime can help give you absolute control over your mesh. Kiali (a Greek word that means “spyglass”) is an open-source and versatile visualization tool that pulls data from Prometheus and the host Kubernetes to generate a communication graph of the mesh that shows service-to-service interactions. With Kiali, since the entire communication stack (Istio and Kubernetes) is available to you, rather than just that of Istio from Grafana, you get much better visibility of the system with Kiali.

For installing Kiali in the mesh, set the parameter `--set kiali.enabled=true` in the Helm installation options of Istio. Since Kiali requires a username and password for configuration, we will create a secret that Kiali will read by default with both username and password set as `admin` (Base64 encoded).

Code Listing 149: Kiali secret

```
apiVersion: v1
kind: Secret
metadata:
  name: kiali
  namespace: istio-system
  labels:
    app: kiali
type: Opaque
data:
  username: YWRtaW4=
  passphrase: YWRtaW4=
```

The policy that we just provisioned is for demo only. You should never leave such policies in the source code repository. Let's apply this policy to our mesh.

Code Listing 150: Create Kiali secret

```
$ kubectl apply -f kiali-secret.yml
secret/kiali created
```

For the secrets to take effect, we will delete the Kiali pod so that it gets recreated. Use the command `kubectl get pods -n istio-system` to find the name of the pod running the Kiali service, and execute the following command by substituting the name of the pod that your cluster has.

Code Listing 151: Delete existing pod

```
$ kubectl delete pod kiali-7d749f9dcb-lbkth -n istio-system
pod "kiali-7d749f9dcb-lbkth" deleted
```

Let's repeat the same exercise as before to forward the port of the Kiali service to a port on the localhost. Use the same step as before to find out the required port of the Kiali service.

Code Listing 152: Port forward Kiali

```
$ kubectl port-forward -n istio-system svc/kiali 8080:20001
```

Forwarding from 127.0.0.1:8080 -> 20001
Forwarding from [::1]:8080 -> 20001

Navigate to <http://localhost:8080> to visit the Kiali dashboard. You will be asked to enter a username and password, which would be **admin** for both the fields. On the **Overview** dashboard, you will see all the applications that are executing in your mesh. You can click on any application to view the health of the services in that application.

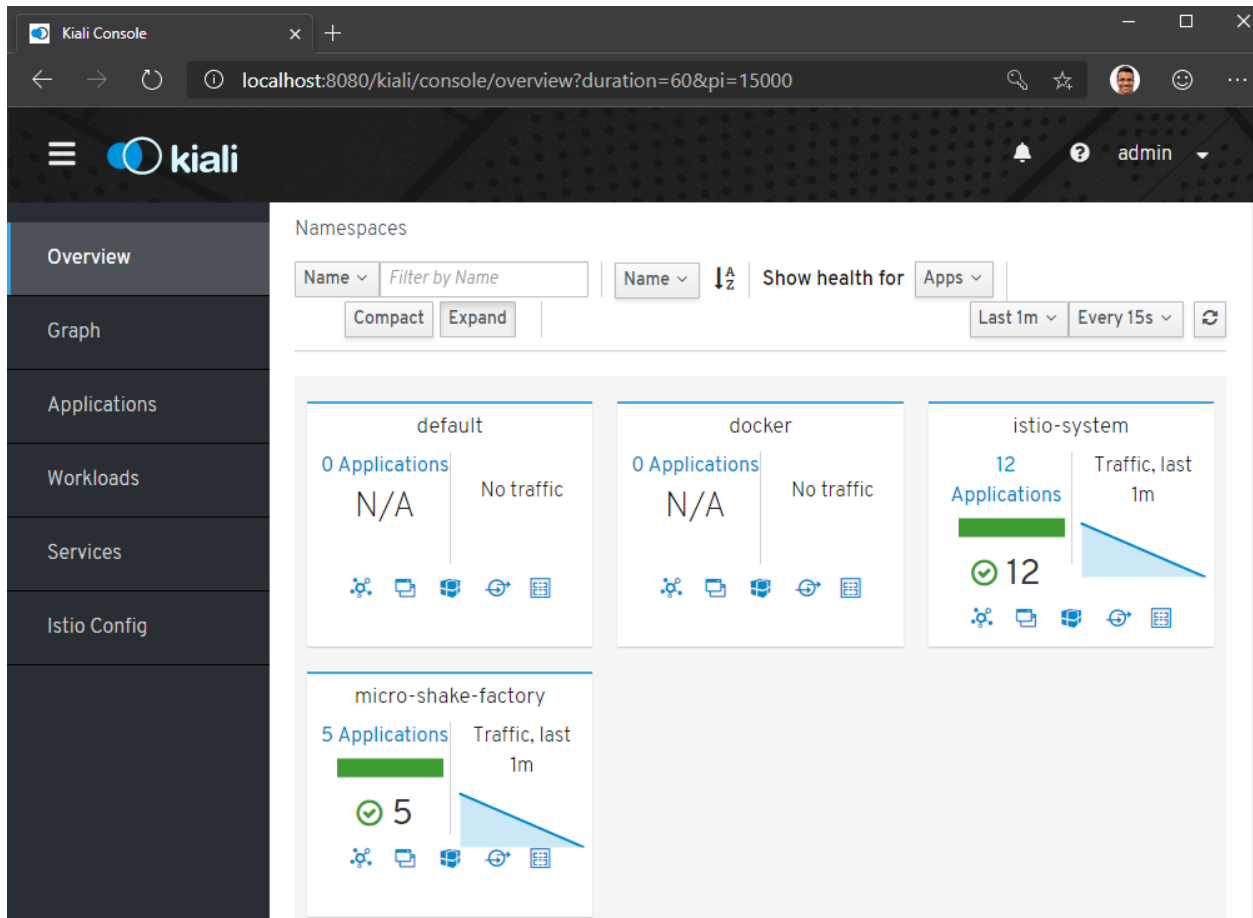


Figure 22: Kiali dashboard

For each application, the dashboard also shows incoming and outgoing traffic metrics. You can bring up this dashboard and send some traffic to the service in the background to light it up.

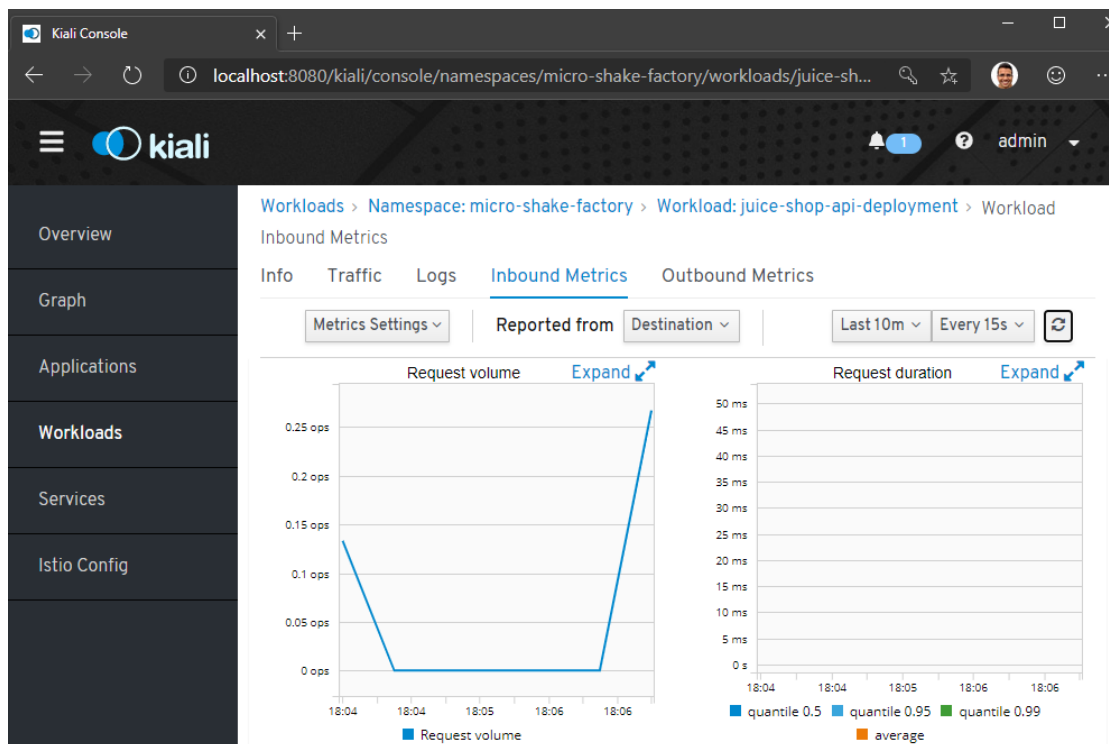


Figure 23: Kiali mesh visualization

Finally, you can also get a visual representation of the services in your namespace by clicking on the **Graph** tab and selecting your namespace from the drop-down list.

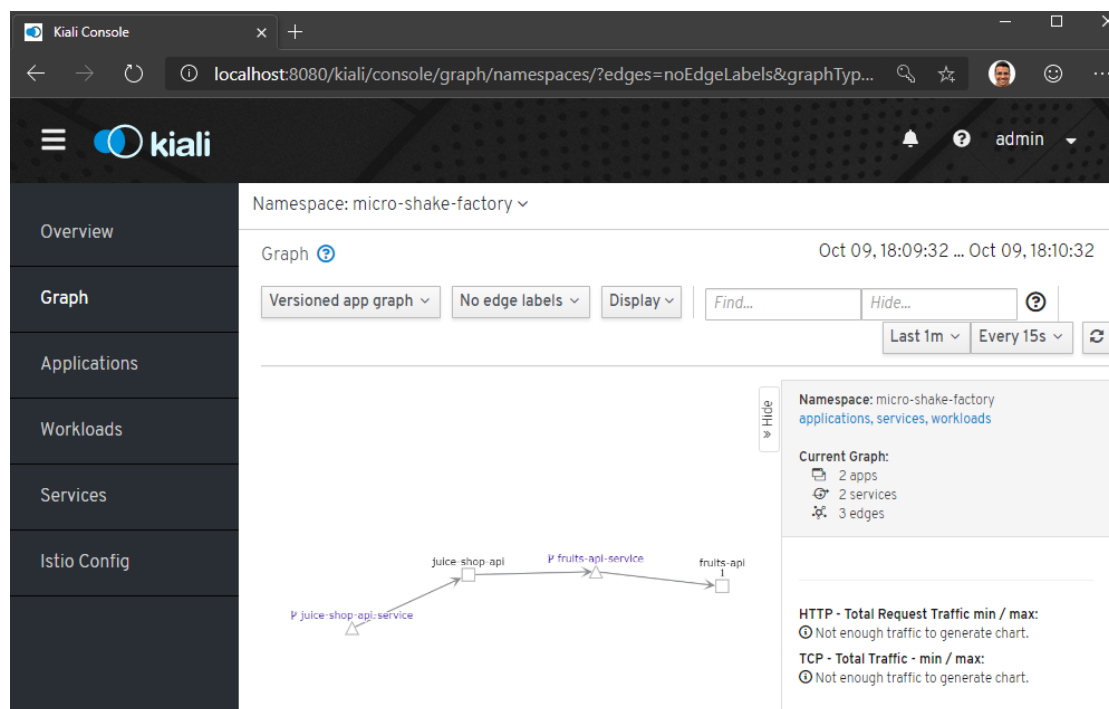


Figure 24: Kiali namespace visualization

The **Istio Config** tab can show you any misconfigurations in your mesh. This feature can surface issues such as virtual services listening to a gateway that does not exist, routes that do not exist, multiple virtual services for the same host, and non-existent service subsets.

Istioctl observability utilities

The **istioctl dashboard** command is a built-in wrapper over the port-forward commands that we discussed for the various dashboards. The command **istioctl dashboard <dashboard-name>** will launch a browser to bring up one of the available dashboards: **controlz**, **envoy**, **grafana**, **jaeger**, **kiali**, **prometheus**, and **zipkin**.

Summary

In this chapter, we discussed the elements of observability: metrics, traces, and logs; and how Istio supports all of them. We saw the observability visualization tools that are packaged with the Istio binary, and that it requires minimal effort from us to set up. However, there are costs to telemetry, and indiscriminate logging and tracing may hamper the performance of the application. We will discuss the aspect of performance and some recent service mesh initiatives in the next chapter.

Chapter 9 Next Steps

By now, you understand most of the nuances of Istio and its ability to offload the east-west network traffic concerns of applications to the platform. With widespread adoption of service mesh, we now need to address challenges with respect to performance, interoperability, and cross-cluster deployment. Let's explore some of these concepts succinctly to continue our learning journey beyond this book.

Service Mesh Interface

Popular service mesh implementations such as Istio, Linkerd, Kong, and Cilium expose their own APIs that are unique to the implementation. Custom APIs lead to vendors building tooling for a handful of popular service meshes at a cost to other offerings. Moreover, customers who deploy an implementation of the service mesh to their infrastructure lose the ability to migrate to another implementation without incurring huge costs, which leads to vendor lock-in.

Companies like Microsoft, Pivotal, Red Hat, Linkerd, and many others recognized the challenges of different service mesh APIs and pioneered building a baseline of common APIs for service mesh that can be implemented by different providers. A common API will bring standardization to customers and space for innovation to providers and tooling vendors.

[Service Mesh Interface \(SMI\) specification](#) consists of four API objects, with each object defined as a Kubernetes CRD. The following are the objects declared in the SMI spec:

- **Traffic spec:** The types defined in this spec control the shape of traffic based on the protocol (HTTP and TCP). These resources work with the access control spec to control traffic at a protocol level. The two types defined in this spec are **HTTPRouteGroup** and **TCPRoute**.
- **Traffic access control spec:** The types in this spec restrict the audience of the services on the mesh. By default, the spec dictates that no traffic can reach any service, and access control is used to explicitly grant access to any service. This spec only controls request authorization, leaving the authentication aspect to the implementation, such as Istio. The only type defined in this spec is **TrafficTarget**, which defines the source, destination, and route of the traffic.
- **TrafficSplit spec:** This type determines the amount of traffic that should land at a version of a service on a per-client basis. Declaring a traffic split requires three elements: root service, which is the point of origin of traffic; backend service, which is a subset of root service; and weights, which is the ratio in which the traffic should be split between the various backend services.
- **TrafficMetrics spec:** This spec defines the types that surface metrics related to HTTP traffic from the service mesh. These metrics can be consumed by tools such as CLI, HPA scalers, automated canary propagation, and visualization.

To support easy adoption, the community is working on creating adapters known as SMI adapters that form a bridge between SMI specs and the underlying platform. Istio already has [an adapter](#) that deploys as another CRD in the cluster in the `istio-system` namespace. The adapter regularly polls SMI objects and creates Istio configurations from the objects. The following is a simple example that shows the difference between specifications of traffic split using SMI and using virtual service in Istio.

Code Listing 153: SMI traffic split specification

```
# SMI
apiVersion: split.smi-spec.io/v1alpha1
kind: TrafficSplit
metadata:
  name: split-sample
spec:
  service: web
  backends:
    - service: web-v1
      weight: 100
    - service: web-v2
      weight: 900
```

The following is the same policy specified using the native Istio configuration model.

Code Listing 154: Istio traffic split specification

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: split-sample
spec:
  http:
    - route:
        - destination:
            host: web
            subset: v1
            weight: 10
        - destination:
            host: web
            subset: v2
            weight: 90
```

SMI enables developers to experiment with multiple service meshes without making any changes to the application. As more and more service mesh implementations onboard SMI, customers will have the flexibility to use a unified API, and the tooling vendors will be able to use their existing investments across all the service meshes.

Knative

[Knative](#) is a serverless framework from Google, Pivotal, and other industry players to build serverless-style functions in Kubernetes. Knative is built upon Istio and Kubernetes, which provide it a runtime environment and advanced networking capabilities, respectively. Knative provides Kubernetes Custom Resource Definitions (CRDs), from which you can create custom objects to provision serverless applications in a cluster. The following diagram shows the various personas involved in delivering a serverless application with Knative.

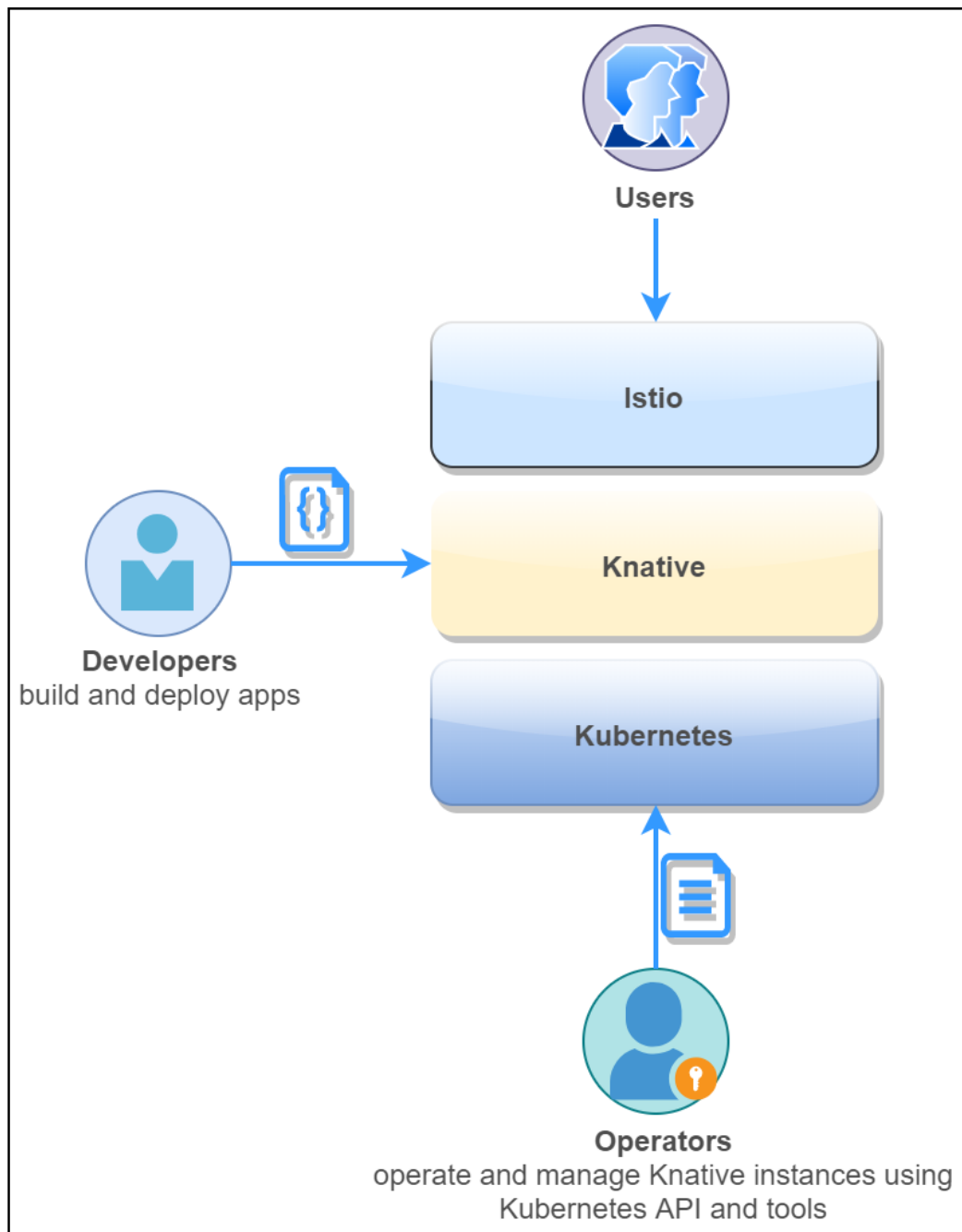


Figure 25: Knative personas

Knative consists of three primary components (**Build**, **Serve**, and **Event**) that are used to deliver serverless applications. Each of the components are installed using a CRD in Kubernetes. Let's take a brief look at these components.

Build

The **Build** spec defines how the application code can be packaged in a container from its source code. This spec is useful if you are using the Google container building service. This component is optional if you are using CI tools such as Azure DevOps, Jenkins, and Chef to generate a container image. The image generated through this specification is pushed to a container registry, and then used in subsequent Knative CRD specifications.

Serve

The Knative service template extends Kubernetes to support the deployment and execution of serverless workloads. It supports scaling of application instances all the way down to zero. If requests arrive after the application has been scaled down to zero, then the requests are queued and Knative starts scaling out application instances to process the pending requests. The asynchronous nature of processing makes Knative unsuitable as a host of backend services for web applications, but suitable for hosting batch jobs and event-driven jobs.

Code Listing 155: Service specification

```
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: fruits-api-svc
  namespace: default
spec:
  runLatest:
    configuration:
      revisionTemplate:
        spec:
          container:
            image: istiosuccinctly/fruits-api:1.0.0
```

The previous specification will deploy the **fruits** API service as a serverless service, which will scale out and down as per the number of requests made to the service.

Events

Event provides a way for the services to produce and consume events. The events can be supplied by any pluggable event source, and the events produced by serverless services can be delivered through various pub/sub-broker services. Several event sources such as Kafka, Container, AWS SQS, and Kubernetes events are already supported by Knative.

Finally, serverless functions built for popular serverless managed services like AWS Lambda and Azure Functions already support containerization, and therefore, they can be easily migrated to Knative.

Istio performance

We have discussed how Istio abstracts network concerns from the application without impacting the application code. However, the data plane components and control plane components of Istio have performance implications, and they require a different mitigation strategy for each component. We will discuss the performance of the individual components of Istio next.

Control plane performance

The control plane of Istio manages services, virtual services, and other objects. The management overhead of the services increases with the number of services on the mesh. As a result, the CPU and memory requirements of Pilot (one of the control plane components) is directly proportional to the service configurations. The CPU consumption of Pilot depends upon the following factors:

- The rate of deployment changes.
- The rate of configuration changes.
- The total number of services deployed in the mesh.

In a cluster where Istio is deployed with namespace isolation, a single instance of Pilot can handle approximately 1,000 services and 2,000 sidecars with just one vCPU and 1.5 GB of memory. The performance of Pilot can be directly affected by scaling it out, which will reduce the time required for applying configuration changes in the mesh.

Data plane performance

We know that the data plane of Istio intercepts every request in the mesh, and it also takes care of networking concerns such as service discovery, routing, and load balancing. The networking features of Envoy directly affect its performance. For example, if the number of requests sent to the services in a mesh is high, then it will degrade the performance of the data plane. Similarly, factors such as the size of the request or response, the protocol used for the requests, and active client connections within the mesh affect the performance of the data plane. The operators of the mesh are required to balance the performance of the data plane with the expected volume of traffic to the services on the mesh.

We know that the sidecar proxy operates on the data path of the request, which is where it consumes CPU and memory. The total resource utilization of the proxy is dependent on the number of resources that you configure in it. For example, if you provision many listeners, policies, and routes, then the provisioned resources will increase the memory required by the proxy. Since Envoy does not buffer request data, the rate of requests does not affect the memory consumption of the proxy.

In Istio version 1.1, the concept of namespace isolation was introduced, which you can use to configure the CPU and memory quota at the namespace level. The namespace isolation feature is extremely useful for namespaces with many services since a proxy sharing the namespace with other services may eat into the quota of that namespace.

Mixer policies such as authentication and filters can also add to the latency in responses from Envoy, as these policies are evaluated (or looked up) for each request. Another function of Mixer is to aggregate telemetry, for which the sidecar proxy spends some time collecting telemetry from each request. During the time of telemetry collection, Envoy does not process another request, which adds to the request latency. Therefore, telemetry should only be configured for required values so that Envoy does not spend additional time aggregating unnecessary logs.

Multi-cluster mesh

A multi-cluster mesh spans many clusters, but it is administered through a single console. It can be implemented as meshes with a single control plane or multiple control planes. In a multi-cluster service mesh, two services with the same name and namespace in different clusters are considered the same.

Multi-cluster service meshes abstract the physical location of services from the consumers, which ensures that services will be available to the client even if a cluster stops functioning. There are two approaches to implementing a multi-cluster service mesh:

- Provision a single Istio control plane that can access and configure services in all the clusters. This approach is beneficial for staging and/or production environments where secondary clusters may be used for canary releases or act as a backup for disaster recovery.
- Provision multiple Istio control planes with replicated services and routing configurations. In this setup, the ingress gateways are responsible for establishing communication between clusters. The DNS configurations of Istio control planes manage the communication between services across the clusters.

You can read more about multi-cluster service mesh [here](#).

Summary

This concludes our journey of learning Istio. In this chapter, we discussed the service mesh standardization initiative called Service Mesh Interface (SMI). We also discussed how another upcoming project, named Knative, can help you build serverless applications on Istio and Kubernetes. We proceeded to discuss some of the important factors that affect the performance of Istio and discussed some mitigation strategies for them. Finally, we touched upon the subject of multi-cluster mesh deployments, which is an area that you should explore for building highly available Istio meshes.

We hope that you enjoyed learning Istio with us, and that we were able to ignite in you the desire to explore Istio. Thank you for being with us on this learning journey.