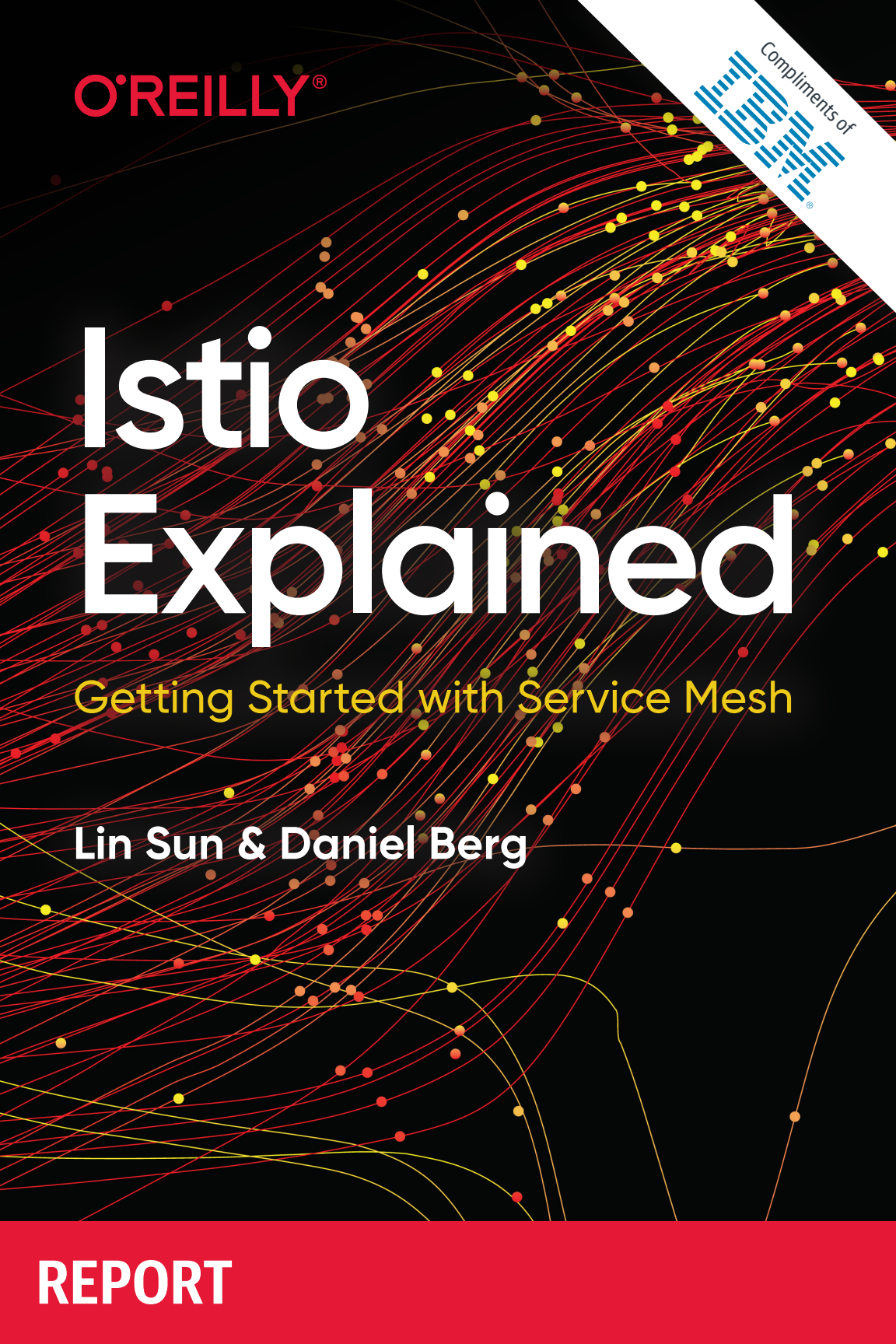
O'REILLY®
Compliments of
IBM
Istio
Explained
Getting Started with Service Mesh
Lin Sun & Daniel Berg
REPORT

Build Smart

Istio Explained

Getting Started with Service Mesh

Lin Sun and Daniel Berg

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Istio Explained
by Lin Sun and Daniel Berg

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: John Devins
Development Editor: Virginia Wilson
Production Editor: Christopher Faucher
Copyeditor: Octal Publishing, LLC
Proofreader: Kim Wimpsett
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

November 2019: First Edition

Revision History for the First Edition
2019-11-27: First Release

978-1-492-07393-2

[LSI]

Table of Contents

Foreword

Organizations are adopting cloud infrastructure and microservices to improve their ability to deliver new capabilities for their respective businesses. Building applications on this infrastructure and with this architecture forces a new perspective for solving problems that aren't all that new. For example, we've known for a while that the network causes challenges like latency and security that application teams need to consider, but in a cloud environment in which an organization does not own the network and where services can be autoscaled or fail unexpectedly, previous-generation solutions for this problem might not be adequate.

A service mesh, like Istio, provides a fresh look at solving application-networking challenges such as reliably connecting applications, securing traffic, and observing/controlling the traffic as it passes through the system between the services. Istio is an open source implementation of a service mesh that uses the Envoy proxy, the architecture for which has its origins in solving application-networking challenges in large distributed systems like those at Google and IBM. Istio has a large, flourishing, and vibrant community, with contributors from IBM, Google, Red Hat, VMware, Pivotal, among others. Istio's predictable quarterly releases produce regular feature enhancements, performance gains, and usability improvements. Istio has gained much hype as any new, interesting technology does, but has evolved in the last two years to become more stable and production ready.

If you're new to service mesh or to Istio, this report is for you. Burr Sutter and I wrote the first report on Istio for O'Reilly, and this report by Lin and Dan is a much-needed, in-depth, updated

complement to that original work. I had the pleasure of reviewing this report, and Dan and Lin's approach is notable in taking the time to help you understand not just what Istio provides in terms of functionality, but also how it works and things to watch out for. Istio provides a wealth of functionality, but in this report Dan and Lin introduce you to its core capabilities, using consumable examples to help you get up to speed. I hope you enjoy the report, and on behalf of the Istio community, welcome to the world of service mesh!

— Christian Posta
Global Field CTO, Solo.io

Preface

Microservices can be complicated and difficult to manage. The increased use of containers and the cloud have expanded the distributed nature of applications and further complicated the ability of development teams to understand and control interactions among services within these environments. These complexities have given rise to a new solution called a *service mesh*, which helps teams manage the interactions between microservices.

Who This Book Is For

We wrote this brief book for developers and operators; however, anyone responsible for the delivery of microservices will find the material here valuable. We assume you are just learning about Istio and service mesh, or you have been recently introduced to Istio and are looking for a "getting started" guide.

What You Will Learn

In the pages that follow, we provide you with a solid background on the challenges of microservices, explain what a service mesh is, describe how a service mesh works, and explore the current service mesh landscape. Starting with Chapter 2, we'll use Istio as our main service mesh implementation to explain how to set one up and use it. We'll describe the Istio architecture and explore Istio's observability, traffic management, and security capabilities.

You don't need to understand and consume all service mesh features at once. You can instead adopt features incrementally and still enjoy some of the benefits a service mesh offers. To that end, we take an

incremental approach to teaching you how to adopt a service mesh like Istio, our goal being to set you up for gradual adoption so that you can see benefits as quickly as possible.

Why Istio?

Though there are many service mesh options to choose from (as you'll see in Chapter 1), because we are most familiar with it, we chose Istio to illustrate the benefits a service mesh can offer through its features. We encourage you to use the information we provide in this book to evaluate the available options and choose the solution best suited for your needs.

Prerequisites

The working examples in this book build on Kubernetes for managing the sample's containers, and Kubernetes also serves as the platform for Istio itself. Thus, to get the most from our examples, it would be helpful for you to have a basic understanding of Kubernetes. To quickly get up to speed, we recommend that you check out this Kubernetes overview and its related links: "Kubernetes: A Simple Overview." As with the adoption process for any new technology, you could likely to run into difficulties with configuration or setup. In Chapter 2, we provide an introduction to techniques and commands to help you troubleshoot the issues you may encounter on your journey to adopting service mesh.

Acknowledgments

We would like to thank the entire Istio community for its passion, dedication, and tremendous commitment to the Istio project. Without the project maintainers and contributors to the project over the years, Istio would not have the rich feature set, diverse code base, and large ecosystem that it has today.

We also extend our thanks to John Alcorn and Ryan Claussen, the original authors of the example Kubernetes application we use as an exemplar in this book. Also, we would like to thank Christian Posta, Burr Sutter, and Virginia Wilson for their reviews, feedback, and overall wisdom that was provided during the creation of the book. A very special thanks to Peter Wassel and Jason McGee for all of their support and encouragement during this endeavor.

CHAPTER 1

Introduction to Service Mesh

In this chapter, we explore the notion of a service mesh and the vast ecosystem that has emerged in support of service mesh solutions. Organizations face many challenges when managing services, especially in a cloud native environment. We are introducing service mesh as a key solution on your cloud native journey because we believe that a service mesh should be a serious consideration for managing complex interactions between services. An understanding of service mesh and its ecosystem will help you choose an appropriate implementation for your cloud solution.

Challenges in Managing Microservices

Microservices are an architecture and development approach that breaks down business functions into individually deployable and managed services. We view microservices as loosely coupled components of an application that communicate with each other using well-defined APIs. A key characteristic of microservices is that you should be able to update them independent of one another, which enables smaller and more frequent deployments. Having many loosely coupled services that are independently and frequently changing does promote agility, but it also presents a number of management challenges:

- Observing interactions between services can be complex when you have many distributed, loosely coupled components.

- Traffic management at each service endpoint becomes more important to enable specialized routing for A/B testing or canary deployments without impacting clients within the system.
- Securing communication by encrypting the data flows is more complicated when the services are decoupled with different binary processes and possibly written in different languages.
- Managing timeouts and communication failures between the services can lead to cascading failures and is more difficult to do correctly when the services are distributed.

Many of these challenges can be resolved directly in the services code. However, adjusting the service code puts a massive burden on you to properly code solutions to these problems, and it requires each microservice owner to agree on the same solution approach to assure consistency. Solutions to these types of problems are complex, and it is extremely error prone to rely on application code changes to provide the solutions. Removing the burden of codifying solutions to these problems and reducing the operational costs of managing microservices are primary reasons we have seen the introduction of the service mesh.

What Is a Service Mesh Anyway?

A service mesh is a programmable framework that allows you to observe, secure, and connect microservices. It doesn't establish connectivity between microservices, but instead has policies and controls that are applied on top of an existing network to govern how microservices interact. Generally a service mesh is agnostic to the language of the application and can be applied to existing applications usually with little to no code changes.

A service mesh, ultimately, shifts implementation responsibilities out of the application and moves them into the network. This is accomplished by injecting behavior and controls within the application that are then applied to the network. This is how you can accomplish things such as metrics collecting, communication tracing, and secure communication without changing the applications themselves. As stated earlier, a service mesh is a programmable framework. This means that you can declare your intentions, and the mesh will ensure that your declared intentions are applied to the services and network. A service mesh simplifies the application

code, making it easier to write, support, and develop by removing complex logic that normally must be bundled in the application itself. Ultimately, a service mesh allows you to innovate faster.

How Did We Get Here?

Netflix pioneered the development of early frameworks for managing microservices, as part of Netflix OSS ("OSS" stands for "open source software"). This framework used components such as Asgard (control plane), Eureka (service registry), Zuul (load balancer gateway), and Ribbon (client-side load balancer). These early frameworks were implemented as a series of libraries written in Java. To make use of their features, you had to modify your Java application by adding the necessary Netflix OSS libraries and adjusting the application logic to make use of the types and methods within these libraries. This approach worked well if you were developing Java applications and were willing to adjust the code.

Recent years have seen a rise in the use of containers and Kubernetes (container orchestration) as the basis for microservices. Using containers has made it simpler to develop applications using multiple languages (e.g., Python, Golang, Java, etc.). Containers also provided opportunities for standardizing operational capabilities, as more features were being abstracted out of the application code and moved into the container platform. These conditions helped usher in the modern service management system that we now call a service mesh.

How Does a Service Mesh Work?

Many service mesh implementations have the same general reference architecture (see Figure 1-1). A service mesh will have a *control plane* to program the mesh, and client-side proxies in the *data plane* (shown below the dashed line) that are within the request path and serve as the control point for securing, observing, and routing decisions between services. The control plane transfers configurations to the proxies in their native format. The client-side proxies are attached to each application within the mesh. Each proxy intercepts all inbound and outbound traffic to and from its associated application. By intercepting traffic, the proxies have the ability to inject behavior on the communication flows between services. Following

is a list of behaviors commonly found in a service mesh implementation:

- Traffic shaping with dynamic routing controls between services
- Resiliency support for service communication such as circuit breakers, timeouts, and retries
- Observability of traffic between services
- Tracing of communication flows
- Secure communication between services

In Figure 1-1, you can see that the communication between two applications such as App1 and App2 is executed via the proxies versus directly between the applications themselves, as indicated by the red arrows. By having communication routed between the proxies, the proxies serve as a key control point for performing complicated tasks such as initiating transport layer security (TLS) handshakes for encrypted communication (shown on the red line with the lock in Figure 1-1). Since the communication is performed between the proxies, there is no need to embed complex networking logic in the applications themselves. Each service mesh implementation option has various features, but they all share this general approach.

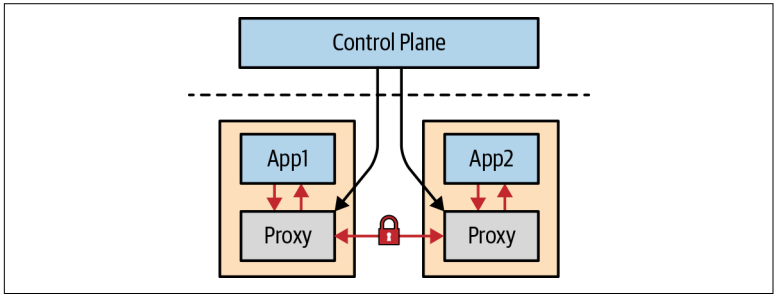

Figure 1-1. Anatomy of a service mesh

The Service Mesh Ecosystem

You might find navigating the service mesh ecosystem a bit daunting because there are many different implementation choices. While most choices share the same reference architecture shown in Figure 1-1, there are variations in approach and project structure you should consider when making your service mesh selection.

Here are some questions to ask yourself when selecting a service mesh implementation:

- Is it an open source project governed by a diverse contributor base?
- Does it use a proprietary proxy?
- Is the project part of a foundation?
- Does it contain the feature set that you need and want?

The fact that there are many service mesh options validates the interest of service mesh, and it shows that the community has not selected a de facto standard as we have seen with other projects such as Kubernetes for container orchestration. Your answers to these questions will have an impact on the type of service mesh that you prefer, whether it is a single vendor-controlled or a multivendor, open source project. Let's take a moment to review the service mesh ecosystem and describe each implementation so that you have a better understanding of what is available.

Envoy

The Envoy proxy is an open source project originally created by the folks at Lyft. The Envoy proxy is an edge and service proxy that was custom built to deal with the complexities and challenges of cloud native applications. While Envoy itself does not constitute a service mesh, it is definitely a key component of the service mesh ecosystem. What you will see from exploring the service mesh implementations is that the client-side proxy from the reference architecture in Figure 1-1 is often implemented using an Envoy proxy.

Envoy is one of the six *graduated projects* in the Cloud Native Computing Foundation (CNCF). The CNCF is part of the Linux foundation, and it hosts a number of open source projects that are used to manage modern cloud native solutions. The fact that Envoy is a CNCF graduated project is an indicator that it has a strong community with adopters using an Envoy proxy in production settings. Although Envoy was originally created by Lyft, the open source project has grown into a diverse community, as shown by the company contributions in the CNCF DevStats graph shown in Figure 1-2.

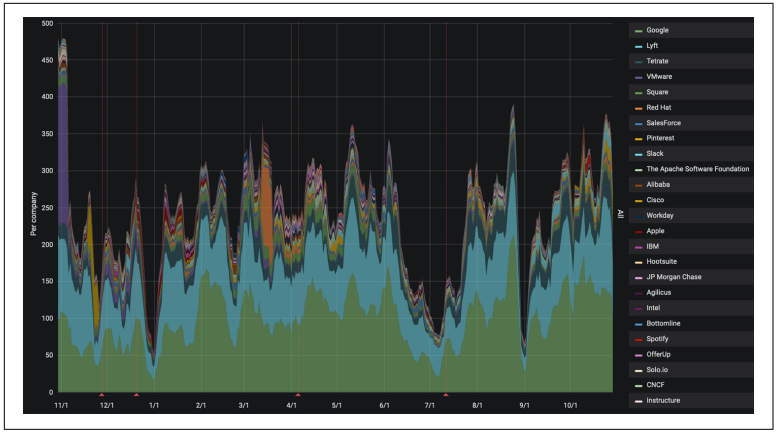

Figure 1-2. Envoy 12-month contribution distribution

Istio

The Istio project is an open source project cofounded by IBM, Google, and Lyft in 2017. Istio makes it possible to connect, secure, and observe your microservices while being language agnostic. Istio has grown to include contributions from companies beyond its original cofounders, companies such as VMware, Cisco, and Huawei, among others. Figure 1-3 shows company contributions over the past 12 months using the CNCF DevStats tool. As of this writing, the open source project Knative also builds upon the Istio project, providing tools and capabilities to build, deploy, and manage serverless workloads. Istio itself builds upon many other open source projects such as Envoy, Kubernetes, Jaeger, and Prometheus. Istio is listed as part of the CNCF Cloud Native Landscape, under the Service Mesh category.

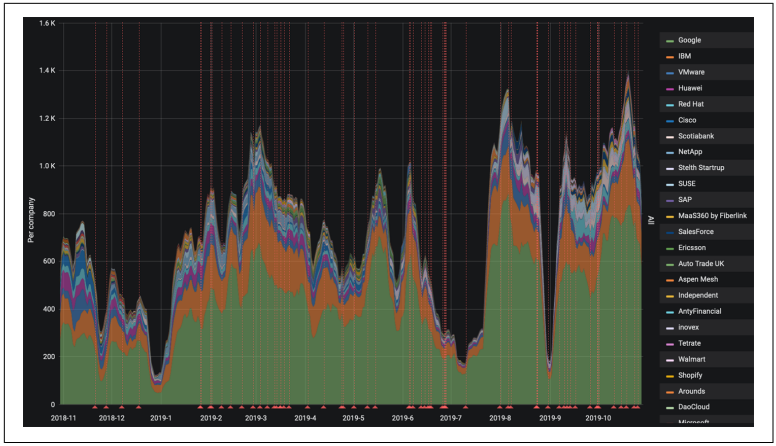

Figure 1-3. Istio 12-month contribution distribution

The Istio control plane extends the Kubernetes API server and utilizes the popular Envoy proxy for its client-side proxies. Istio supports mutual TLS authentication (mTLS) communication between services, traffic shifting, mesh gateways, monitoring and metrics with Prometheus and Grafana, as well as custom policy injection. Istio has installation profiles such as demo and production to make it easier to provision and configure the Istio control plane for specific use cases.

Consul Connect

Consul Connect is a service mesh developed by HashiCorp. Consul Connect extends HashiCorp's existing Consul offering, which has service discovery as a primary feature as well as other built-in features such as a key-value store, health checking, and service segmentation for secure TLS communication between services. Consul Connect is available as an open source project with HashiCorp itself being the predominant contributor. HashiCorp has an enterprise offering for Consul Connect for purchase with support. At the time of writing, Consul Connect was not contributed to the CNCF or another foundation. Consul is listed as part of the CNCF Cloud Native Landscape under the Service Mesh category.

Consul Connect uses Envoy as the sidecar proxy and the Consul server as the control plane for programming the sidecars. Consul Connect includes secure mTLS support between microservices and observability using Prometheus and Grafana projects. The secure

connectivity support uses the HashiCorp Vault product for managing the security certificates. Recently, HashiCorp has introduced Layer 7 (L7) traffic management and mesh gateways into Consul Connect as beta features.

Linkerd

The Linkerd service mesh project is an open source project as well as a CNCF incubating project focusing on providing an ultralight weight mesh implementation with a minimalist design. The predominant contributors to the Linkerd project are from Buoyant, as shown in the 12-month CNCF DevStats company contribution graph in Figure 1-4. Linkerd has the key capabilities of a service mesh, including observability using Prometheus and Grafana, secure mTLS communication, and—recently added—support for service traffic shifting. The client-side proxy used with Linkerd was developed specifically for and within the Linkerd project itself, and written in Rust. Linkerd provides an injector to inject proxies during a Kubernetes pod deployment based on an annotation to the Kubernetes pod specification. Linkerd also includes a user interface (UI) dashboard for viewing and configuring the mesh settings.

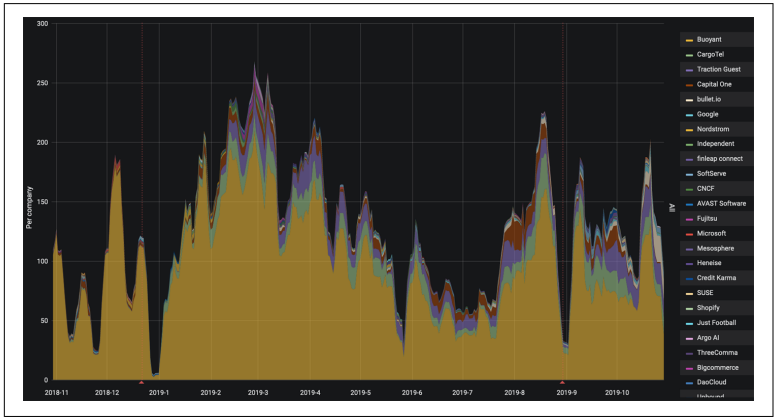

Figure 1-4. Linkerd one-year contribution distribution

App Mesh

App Mesh is a cloud service hosted by Amazon Web Services (AWS) to provide a service mesh with application-level networking support for compute services within AWS such as Amazon ECS, AWS Fargate, Amazon EKS, and Amazon EC2. As the project URL suggests, AWS App Mesh is a closed sourced managed control plane that is proprietary to AWS. App Mesh utilizes the Envoy proxy for the sidecar proxies within the mesh; this has the benefit that it may be compatible with other AWS partners and open source tools. It appears that App Mesh's API has similar routing concepts as the Istio control plane, unsurprising since Istio serves as a control plane for the Envoy proxy. As of this writing, secure mTLS communication support between services is not implemented but is a road map item. The focus of App Mesh appears to be primarily traffic routing and observability.

Kong

Kong's service mesh builds upon the Kong edge capabilities for managing APIs and has delivered these capabilities throughout the entire mesh. Though Kong is an open source project, it appears that its contributions are heavily dominated by Kong members. Kong is not a member of a foundation, but is listed as part of the CNCF Cloud Native Landscape under the API Gateway category. Kong does provide Kong Enterprise, which is a paid product with support.

Much like all the other service mesh implementations, Kong has both a control plane to program and manage the mesh as well as a client-side proxy. In Kong's case the client-side proxy is unique to the Kong project. Kong includes support for end-to-end mTLS encryption between services. Kong promotes its extensibility feature as a key advantage. You can extend the Kong proxy using Lua plugins to inject custom behavior at the proxies.

AspenMesh

AspenMesh is unlike the other service mesh implementations in being a supported distribution of the Istio project. AspenMesh does have many open source projects on GitHub, but its primary direction is not to build a new service mesh implementation but to harden and support an open source service mesh implementation through a paid offering. AspenMesh hosts components of Istio such

as Prometheus and Jaeger, making it easier to get started and use over time. It has features above and beyond the Istio base project, such as a UI and dashboard for viewing and managing Istio resources. AspenMesh has introduced additional tools such as Istio Vet, which is used to detect and resolve misconfigurations within an instance of Istio. AspenMesh is an example where there are new markets emerging to offer support and build upon a source service mesh implementation such as Istio.

Service Mesh Interface

Service Mesh Interface (SMI) is a relatively new specification that was announced at KubeCon EU 2019. SMI is spearheaded by Microsoft with a number of backing partners such as Linkerd, HashiCorp, Solo.io, and VMware. SMI is not a service mesh implementation; however, SMI is attempting to be a common interface or abstraction for other service mesh implementations. If you are familiar with Kubernetes, SMI is similar in concept to what Kubernetes has with Container Runtime Interface (CRI), which provides an abstraction for the container runtime in Kubernetes with implementations such as Docker and containerd. While SMI may not be immediately applicable for you, it is an area worth watching to see where the community may head as far as finding a common ground for service mesh implementations.

Conclusion

The service mesh ecosystem is vibrant, and you have learned that there are many open source projects as well as vendor-specific projects that provide implementations for a service mesh. As we continue to explore service mesh more deeply, we will turn our attention to the Istio project. We have selected the Istio project because it uses the Envoy proxy, it is rich in features, it has a diverse open source community, and, most important, we both have experience with the project.

CHAPTER 2

Introducing Istio

Now we turn our attention to our prime running example: the Istio service mesh. After reading this chapter, you should have a good understanding of Istio's architecture, how it operates, and the key tenants of the project. Because we want to provide you with a hands-on cloud journey, we will walk you through getting Istio installed so you can use it for tasks in later chapters.

Why Do We Love Istio?

We choose to focus our attention on Istio because we are founding members of the project, and we have deep knowledge and experience with the project implementation. We have some familiarity with other service mesh implementations, but with Istio we have a greater depth of knowledge and working experience. Istio is a mature service mesh implementation that allows you to break down the complexity of distributed cloud native deployments by taking complex functionality out of the application code and moving it into the network. We find Istio to be the most feature rich, and it's also built to serve the enterprise use cases most like those that we see in our day-to-day jobs.

Istio provides features that enable you to manage a network for deployed services with secure communication, monitoring, version-based load balancing, and much more. Istio works with modern cloud native applications because it requires little to no code changes with automatic sidecar proxy injection (shown in Figure 1-1) that intercepts all network traffic between services.

Individually managing hundreds if not thousands of sidecar proxies would be unwieldy. The Istio control plane provides you with a declarative API for defining your service configurations and policies, which are then propagated to the sidecar proxies with the proper configurations by the Istio control plane. Ultimately, Istio enables you to focus on solving business problems because error-prone logic is removed from the application code.

In Chapter 1, we introduced you to challenges that you may experience as you move to use the cloud and microservices. The Istio service mesh has capabilities that simplify your ability to solve these challenges, including the following:

- Automatic metrics and network tracing collected between services within the mesh, as well as inbound and outbound network communication with external clients and services
- Advanced rule-based traffic routing and control with automatic load balancing for HTTP, gRPC, WebSocket, and TCP traffic across service versions
- Automatic mTLS management for secure service-to-service communication using strong identity-based authentication and authorization
- Built-in service communication resiliency with retries, circuit breaking failover, and fault injection

Now, let's dive deeper into the key features of Istio and provide you with a base understanding for how it's architected. Having a base understanding of the Istio architecture will help you to understand how your interactions with the Istio control plane affect behavior within the mesh.

Istio Features

Istio solves the challenges of managing microservices by using a core set of features that allow you to observe, connect, and secure your services. These features can be broken down into three main categories: *observability*, *traffic management*, and *security*.

Observability

Simply by installing Istio and adding services to the mesh, you will begin to get rich tracing, monitoring, and logging for your services.

This level of information quickly gives you insights into the communication flows between your services, as well as potential performance, security, and other issues that could affect that communication. You can view these insights with customized Grafana dashboards, as well as the other useful tools Istio provides, such as Jaeger for trace flows, and Kiali for rich views of traffic flows.

Istio collects this information in an automated and nondisruptive manner from your service flows. The insights inform you of areas within the mesh where you may need to apply policies using the Istio control-plane APIs. The Istio control plane provides abstractions over the platform, allowing you to define policies so you can gain fine-grained control over the interactions of your services and immediately see the effects within the Grafana and Kiali dashboards.

Traffic Management

By using easy-to-configure rules, you have fine-grained control over how traffic flows between services at both the application layer (Layer 7) as well as the network IP address level (Layer 4). For example, in Kubernetes you have simple round-robin load balancing across all service endpoints. With Istio you can organize service endpoints by version and declare policies with the control plane to control load balancing. You can then also make determinations as to which service to use, based on a plethora of conditions, including the source client identity, the client input type, percentage distribution, geography, and more. Using such rules to control the traffic flow, you can easily adjust traffic as conditions within your application change.

Using Istio's traffic-shaping features allows you to deliver changes more rapidly because you can reduce delivery risks. Istio enables controlled rollout of changes using various deployment patterns such as percentage based, canary, A/B testing, and more. Istio's traffic-shaping support also includes features that increase the resiliency of your application without having to change the code. For example, with distributed microservices, it is more likely that you could see network failures between service calls, or disjointed timeouts between service calls, which result in a poor user experience. Istio allows you to set conditions that control how services recover from service call failures such as circuit breakers, timeouts, and retries. The Istio traffic-shaping features coupled with automatic

insights make it far simpler for you to program higher resiliency and control flows directly into the network of your service mesh.

Security

One of the most difficult features to enable in a distributed cloud application is secure communication between services where you have data encryption and authentication between the services. This is challenging because coding the logic in each service is complicated, and it takes only one improper configuration to expose a security threat. Istio provides a feature that automatically establishes a secure channel between services by managing service identities, certificates, and mTLS handshaking. Istio uses first-class service identity such as a Kubernetes service account to determine the identity of the service, which we covered in detail in "Istio Identities" on page 41 of Chapter 4. This means you can ensure that secure channels exist between your services with certificates that are generated and constantly rotated, dramatically reducing possible security threats between services.

As with all of the features in Istio, managing security between services is also declarative using the APIs available in the Istio control plane. Enabling secure communication within the mesh is not an all-or-nothing setting. Istio has settings that allow permissive secure channels between services. Selective permissive channels make it convenient for you to incrementally add services to the mesh without causing failures. This feature greatly simplifies your journey to the cloud.

Istio Architecture

At a high level, Istio consists of a data plane and a control plane, as shown in the service mesh reference architecture in Figure 1-1. Figure 2-1 depicts the Istio components used to implement the service mesh reference architecture. The Istio data plane is composed of Envoy sidecar proxies running in the same network space as each service to control all network communication between services, as well as Mixer, to provide extensible policy evaluation between services. The Istio control plane is responsible for the APIs used to configure the proxies and Mixer as part of the data plane. The key components of the control plane are Pilot, Citadel, Mixer, and Galley.

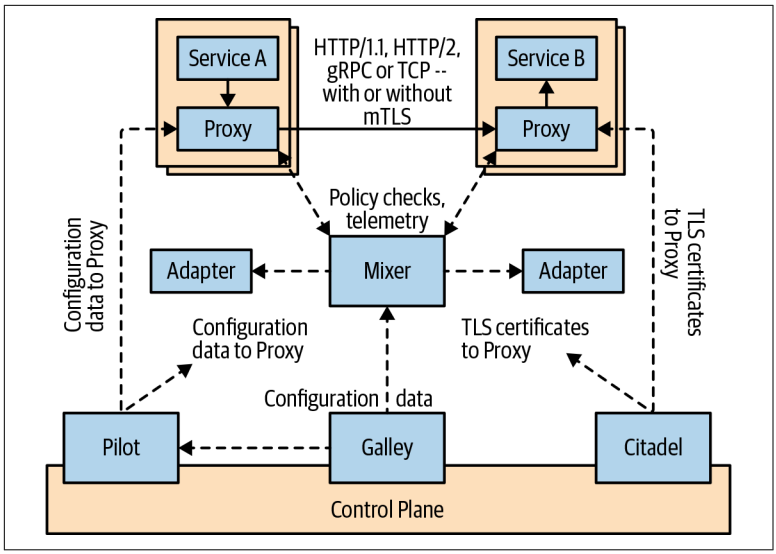

Figure 2-1. Istio architecture diagram

Envoy

Istio uses the Envoy proxy for the sidecars as well as gateways. You will learn more about gateways in Chapters 4 and 5. The Envoy proxy was developed to be extensible, and Istio uses an extended version of the Envoy proxy to provide the features and capabilities needed to work with the Istio control plane. Envoy is deployed as a sidecar to each service endpoint. Within a Kubernetes environment, Envoy is injected into each Kubernetes pod as a separate container. Ingress and egress network traffic in and out of the pod is configured to flow through the sidecar Envoy proxy. Flowing all traffic through the Envoy sidecar provides a control point to allow Istio to gather metrics, control traffic, evaluate policies, and encrypt data transfer. Istio adds to many of Envoy's built-in features, such as load balancing, TLS termination, circuit breakers, health checking, HTTP/2, gRPC, and much more.

Pilot

Pilot is the essential component that programs the Envoy sidecars: it converts Istio-defined APIs into Envoy-specific configurations, which are propagated to Envoy proxy sidecars. Responsible for service discovery within the service mesh, Pilot is also primarily

responsible for traffic-management capabilities as well as resiliency features such as circuit breakers and retry logic. To support service discovery, Pilot abstracts platform-specific service discovery implementations and converts them into a standard format used by sidecars that conform to the Envoy data plane APIs. The abstraction provided by Pilot allows Istio to be used with multiple environments, including Kubernetes, Consul, or Nomad, and provides you with a common interface. We explore more of the details about Pilot's traffic management capabilities in Chapter 5.

Citadel

Citadel provides critical security capabilities within the Istio service mesh. Citadel's primary responsibility is to manage certificates and provide strong service identities to enable strong service-to-service as well as end-user authentication. With the use of Citadel, you can upgrade communication between your microservices from sending plain text to having data sent fully encrypted using mTLS authentication and authorization. We'll get into how Citadel is used to secure communication between services in Chapter 4.

Mixer

Mixer has a dual role within Istio. It enforces access control and usage policies across the service mesh and collects telemetry data from the sidecar proxies as well as other Istio control-plane services. Mixer has been designed to be extensible by allowing you to inject your own specialized policies to be executed by Envoy proxies when communicating between services. This same extensibility framework enables Istio to work with multiple host environments and backends. Request-level telemetry metrics are extracted by the proxies and forwarded to Mixer for evaluation. We get into more of the details about collecting and viewing telemetry data in Chapter 3.

Galley

Galley manages Istio's configuration. It validates, ingests, processes, and distributes Istio's configuration to the other control-plane services. Galley ultimately insulates the other Istio components from the details of obtaining data from the underlying platform, such as Kubernetes.

Installing Istio

Now that you have an overview of the Istio architecture, let's turn our attention to hands-on examples that will help you better understand how Istio works. One of our goals for the rest of this book is to provide you with information and examples that would help you on your cloud journey with an incremental adoption of service mesh such as Istio. To kick things off, you will need to install the Istio control plane.

There are several different ways to install Istio including a Helm chart, Kubernetes YAML files, and, soon to be available, an Istio Operator. For now, let's stick with the most straightforward approach and use the Kubernetes YAML files. We follow the three steps from the Getting Started section on istio.io.

Getting a Kubernetes Cluster

You will need a Kubernetes cluster to be able to follow along with the examples in the rest of this book. We have chosen to use the IBM Cloud Kubernetes Service (IKS) to provision a Kubernetes cluster. You can find more options for obtaining a Kubernetes cluster on *istio.io* in the Platform Setup section. You must use a Kubernetes cluster that is version 1.13 or greater.

Downloading the Istio Release

You begin by downloading the latest Istio release. You will find the current Istio release on the Istio releases page. The *istio.io* site includes the instructions on downloading the Istio release. For example, to download the v1.3.0 release (the current release as of the writing), you would enter the following command in your terminal:

```
$ curl -L https://git.io/getLatestIstio | ISTIO_VERSION=1.3.0 sh -
$ cd istio-1.3.0
```

Setting Up the Istio Command-Line Interface

You need to set up the Istio command-line interface (CLI) before proceeding. The Istio CLI, `istioctl`, is an executable that you set up on your local development environment. You install `istioctl` from the release download by adding it to your `PATH` environment variable:

```
$ export PATH=$PWD/bin:$PATH
```

Confirming Kubernetes Cluster

Istio has requirements against Kubernetes features that are available only in certain Kubernetes versions. You can use the following utility from `istioctl` to verify that your Kubernetes cluster meets the requirements for Istio:

```
$ istioctl verify-install
```

You will get an output similar to the following. The `istioctl` utility checks the Kubernetes API Server, Kubernetes version, whether Istio is already installed, and whether you have permission to create the required Kubernetes resources:

```
Checking the cluster to make sure it is ready for Istio installation...

Kubernetes-api
-----------------------
Can initialize the Kubernetes client.
Can query the Kubernetes API Server.

Kubernetes-version
-----------------------
Istio is compatible with Kubernetes: v1.13.10+IKS.

Istio-existence
-----------------------
Istio will be installed in the istio-system namespace.

Kubernetes-setup
-----------------------
Can create necessary Kubernetes configurations: Namespace,ClusterRole,ClusterRole-
Binding,CustomResourceDefinition,Role,ServiceAccount,Service,Deployments,Config-
Map.

SideCar-Injector
-----------------------
This Kubernetes cluster supports automatic sidecar injection. To enable automatic
sidecar injection see https://istio.io/docs/setup/kubernetes/additional-setup/
sidecar-injection/#deploying-an-app

-----------------------
Install Pre-Check passed! The cluster is ready for Istio installation.
```

Installing Istio Control Plane

Following the quick start evaluation install guide, you can quickly install Istio to be used for evaluation purposes. The evaluation installation will install the Istio custom resource definitions (CRDs) and the Istio demo profile with permissive mTLS. In Chapter 4, we'll explore adopting strict mTLS in an incremental fashion to simplify your onboarding experience.

First, you need to install all required Istio CRDs before installing the Istio control plane, using the following command:

```
$ for i in install/kubernetes/helm/istio-init/files/crd*yaml; \
do kubectl apply -f $i; done
```

When the CRDs are installed, you can then install the Istio demo profile using the `istio-demo.yaml` Kubernetes resources:

```
$ kubectl apply -f install/kubernetes/istio-demo.yaml
```

Wait a minute or two, and then run the `verify-install` command to verify that the installation is successful:

```
$ istioctl verify-install -f install/kubernetes/istio-demo.yaml
```

If you get an error message indicating that the deployment is still in progress, wait a bit longer and rerun the `verify-install` command. You should see an output similar to what is shown here indicating that you have a successful installation:

```
...
Checked 28 crds
Checked 9 Istio Deployments
Istio is installed successfully
```

Conclusion

You should now have a firm understanding of the key features that the Istio service mesh offers for managing, securing, and observing microservices, as well as the core components involved in the implementation of these features. With this basic understanding, you are now ready to continue your cloud journey. We'll get deeper into each of the key features with hands-on tasks that will provide you with the information you'll need to incrementally adopt a service mesh such as Istio for managing your microservices.

CHAPTER 3

Adding Services to the Mesh

The objective of this chapter is to show you how to incrementally add services to the mesh where the services are part of a brownfield application. As part of adding services to the mesh, the mesh is actually integrated as part of the services themselves to make the mesh mostly transparent to the service implementation. A *brownfield application* is one that already exists versus creating a *greenfield application*, which is new and designed specifically for the environment it will run in. With a greenfield application, new code is developed allowing platform features to be added into the code. With a brownfield application, the code already exists and, in an ideal case, you don't want to modify the code when moving into a new environment such as cloud. This chapter introduces you to the brownfield guiding application that will be used to demonstrate how the services of the application can be defined and introduced into the control of an Istio service mesh.

Introducing the Guiding Application

We have selected the IBM Stock Trader application as the guiding application for your cloud native journey experience because it is readily available and is representative of a simple brownfield application that would be taken to the cloud. The Stock Trader application works well for illustrative purposes because it has multiple interacting services as well as a persistent service; this will be important as we take you through the features of Istio.

The Stock Trader application shown in Figure 3-1 is a simple stock trading sample with which you can create various stock portfolios and add shares of stock with a commission. It keeps track of each portfolio's total value and detailed stock holdings. You can find the source code of the application in repositories found in the istio-explained GitHub organization.

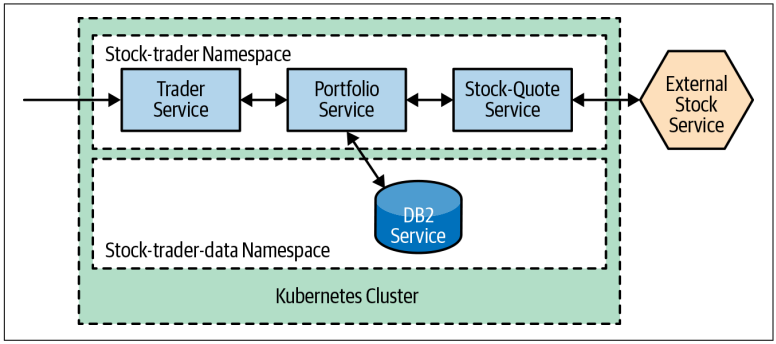

Figure 3-1. Stock Trader application

One strategy for incrementally adding services into the mesh would be to separate additional services by namespace. For example, services added to the mesh will be in a separate namespace from services that will continue to remain outside the mesh.

Deploying the Data Components

In this example, you will use the `stock-trader` namespace for services that will participate in the mesh, and the data services will remain outside the mesh in a separate `stock-trader-data` namespace. You will need to follow these instructions to deploy the DB2 service components in the `stock-trader-data` namespace.

Sidecar Injection

Adding services to the mesh requires that the client-side proxies be associated with the service components and registered with the control plane. With Istio, you have two methods to inject the Envoy proxy sidecar into the microservice Kubernetes pods:

- Automatic sidecar injection
- Manual sidecar injection

Automatic sidecar injection is available within a Kubernetes environment. The automatic sidecar injection uses a Kubernetes mutating admission controller that is invoked during pod scheduling to determine whether it should add the Istio proxy to the deployment specification. The `istio-injected` label on the namespace of the pod is used by the Istio mutating admission controller to determine whether the Istio proxy should be injected into the pod deployment specification.

Manual sidecar injection, on the other hand, allows you to examine the injected deployment YAML prior to deployment. Using the manual sidecar-injection approach gives you fine-grained control over which services are added to the mesh. A benefit of separating services by namespace, it is straightforward to use the automatic sidecar-injection approach configured for all services within a namespace.

Sidecar-Injection Approaches

You can find more information about the different approaches for injecting the Istio Envoy sidecar from the istio.io documentation site.

Using the code snippets that follow, you can create the `stock-trader` namespace enabled with automatic sidecar injection by adding the `istio-injection` label to the namespace. By executing these steps, you have identified the namespace that will be used to add services from the Stock Trader application into the mesh:

```
# create the stock-trader namespace
$ kubectl create namespace stock-trader

# enable the stocker-trader namespace for Istio automatic sidecar injection
$ kubectl label namespace stock-trader istio-injection=enabled

# validate the namespace is annotated with istio-injection
$ kubectl get namespace -L istio-injection
```

Now that you have a namespace with automatic sidecar injection enabled, you are ready to start adding services into the mesh.

Reviewing Service Requirements

Before you add Kubernetes services to the mesh, you need to be aware of the pod and services requirements to ensure that your Kubernetes services meet the minimum requirements.

Service descriptors:

- Each service port name must start with the protocol name, for example, `name: http`.

Deployment descriptors:

- The pods must be associated with a Kubernetes service.
- The pods must not run as a user with UID 1337.
- It is recommended that app and version labels are added to provide contextual information for metrics and telemetry.

You will need to confirm that each of your Kubernetes services meets these requirements and make adjustments as necessary. For the Stock Trader application, you will investigate each service to validate that each has a Kubernetes service association and is not running any container as a user with UID 1337. If you don't have NET_ADMIN security rights, you would need to use the Istio CNI plugin to remove the NET_ADMIN requirement.

Let's explore adjustments to a service configuration based on the Istio mesh service requirements. Open the `portfolio` service deploy YAML file to review both the service and deployment descriptors for validating the service requirements. The `portfolio` service has two ports, one for HTTP traffic on port 9080 and the other for HTTPS traffic on port 9443. Notice that the names of the ports `http` and `https`, respectively, start with the required protocol name as highlighted here:

```
ports:
  - name: http
    protocol: TCP
    port: 9080
    targetPort: 9080
  - name: https
    protocol: TCP
    port: 9443
    targetPort: 9443
```

Also notice that the deployment descriptor declares the `containerPorts` for `9080` and `9443`:

```
ports:
  - containerPort: 9080
  - containerPort: 9443
imagePullPolicy: Always
```

Transparent Service Mesh

The recently released Istio 1.3 introduces the ability to automatically discover the pod container ports, eliminating the requirement to declare `containerPorts`, as needed in previous versions of Istio.

To ensure that more context is provided to metrics and telemetry, you will see that both the app and version labels are included in the `portfolio` deployment descriptor, as shown:

```
labels:
  app: portfolio
  version: v1
  solution: stock-trader
```

Onboarding Portfolio Service

After you follow the instruction to deploy the DB2 data service via its helm chart and populate the DB2 table for the Stock Trader application, you are ready to deploy and onboard your first service to the mesh. You can find the source code of the portfolio service in the `portfolio` repository. You will deploy the `portfolio` service with the following CLI:

```
# create the required JWT secret
$ kubectl create secret generic jwt -n stock-trader \
  --from-literal=audience=stock-trader \
  --from-literal=issuer=http://stock-trader.ibm.com

$ git clone https://github.com/istio-explained/portfolio.git
$ kubectl apply -f portfolio/manifests/deploy.yaml -n stock-trader

# validate pod has reached running status with sidecar injected
$ kubectl get pods -l app=portfolio -n stock-trader
NAME                         READY   STATUS    RESTARTS   AGE
portfolio-9d4c5576f-8qrlc    2/2     Running   0          30s

# view logs to validate no errors
$ kubectl logs -c portfolio --namespace=stock-trader \
  --selector="app=portfolio,solution=stock-trader"
```

The steps to deploy the `portfolio` service work exactly the same with or without Istio. There is nothing in the steps or the configura-

tion that have a strong dependency with Istio. Since you added the `istio-injection` label to the `stock-trader` namespace, the Istio mutating admission controller automatically injects the Envoy proxy sidecar during the deployment of the pod. You can see this by inspecting the detail of the `portfolio` pod. Use the following command to describe the details of the `portfolio` pod that has been deployed:

```
$ kubectl describe pod -l app=portfolio -n stock-trader
```

You should focus your attention on the containers in the output. You will notice that the first container is an init container called `istio-init`, which is started prior to the other containers in the pod. You can see the containers including the `istio-init` container in the output here as a result of running the `kubectl describe` command:

```
Name:               portfolio-66555d8c88-z2nt9
Namespace:          stock-trader
Labels:             app=portfolio
                    pod-template-hash=66555d8c88
                    solution=stock-trader
                    version=v1
Status:             Running
IP:                 172.30.115.89
Controlled By:      ReplicaSet/portfolio-66555d8c88
Init Containers:
  istio-init:
    Container ID:  containerd://
a306e75aa92be8c55ff3dae814c39c5a7701548aa965f03757594a7998f09404
    Image:         docker.io/istio/proxy_init:1.3.0
    Image ID:      docker.io/istio/
proxy_init@sha256:aede2a1e5e810e5c0515261320d007ad192a90a6982cf6be8442cf1671475b8a
    Port:          <none>
    Host Port:     <none>
    Args:
      -p
      15001
      -z
      15006
      -u
      1337
      -m
      REDIRECT
      -i
      *
      -x

      -b
      *
      -d
      15020
    State:          Terminated
      Reason:       Completed
      Exit Code:    0
      Started:      Tue, 17 Sep 2019 21:52:02 -0400
```

```
      Finished:     Tue, 17 Sep 2019 21:52:03 -0400
    Ready:          True
    Restart Count:  0
    Limits:
      cpu:     100m
      memory:  50Mi
    Requests:
      cpu:        10m
      memory:     10Mi
    Environment:  <none>
    Mounts:       <none>
```

The Istio mutating admission controller was responsible for injecting the `istio-init` container. You can see from the *Dockerfile.proxy_init* that the entry point of the container is *istio-iptables.sh*, which is responsible for configuring the pod iptable rules. If you inspect the *istio-iptables.sh* file, you will notice that all inbound ports are redirected to the Envoy proxy container within the pod. You can also see that port 15020 is excluded from redirection (you'll soon learn why this is the case).

When you continue looking through the list of containers in the pod, you will see the `portfolio` container, as expected, as well as the injected `istio-proxy` container. The proxyv2 image is used for the image of the `istio-proxy` container. Notice that the entry point of the image is the pilot-agent executable, which is responsible for bootstrapping the `istio-proxy` container with the default Envoy bootstrap configuration file provided by the image, along with these arguments passed into the `istio-proxy` container shown in the code block below. This is what we call the "dummy proxy." If you envision the `portfolio` application container as a room, you can think of the istio-proxy sidecar as a storage box attached next to the room. See the description of the `istio-proxy` container here, which is shown as part of the `kubectl describe` output.

```
Containers:
  portfolio:
    Container ID:   containerd://
d4b327c3f42c093803c3503fcc5e466eef231f90cdfb6690b5ba595ca492f601
    Image:          docker.io/linsun/portfolio:latest
    ...
  istio-proxy:
    Container ID:  containerd://
ba2a9df2d956cb51c1bf2c47aff1369df0370325ae5f88390fe0dd5e7d858fbe
    Image:         docker.io/istio/proxyv2:1.3.0
    Image ID:      docker.io/istio/
proxyv2@sha256:b41903d0c4e3e218930144f986af18eae4a063a9efdace7b7decd0ec189c7cb9
    Port:          15090/TCP
    Host Port:     0/TCP
    Args:
      proxy
      sidecar
```

```
      --domain
      $(POD_NAMESPACE).svc.cluster.local
      --configPath
      /etc/istio/proxy
      --binaryPath
      /usr/local/bin/envoy
      --serviceCluster
      portfolio.$(POD_NAMESPACE)
      --drainDuration
      45s
      --parentShutdownDuration
      1m0s
      --discoveryAddress
      istio-pilot.istio-system:15010
      --zipkinAddress
      zipkin.istio-system:9411
      --dnsRefreshRate
      300s
      --connectTimeout
      10s
      --proxyAdminPort
      15000
      --concurrency
      2
      --controlPlaneAuthPolicy
      NONE
      --statusPort
      15020
      --applicationPorts
      9080,9443
    State:          Running
      Started:      Tue, 17 Sep 2019 21:52:04 -0400
    Ready:          True
    Restart Count:  0
    Limits:
      cpu:     2
      memory:  1Gi
    Requests:
      cpu:      10m
      memory:   40Mi
    Readiness:  http-get http://:15020/healthz/ready delay=1s timeout=1s
period=2s #success=1 #failure=30
    Mounts:
      /etc/certs/ from istio-certs (ro)
      /etc/istio/proxy from istio-envoy (rw)
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-b7q9c (ro)
```

You will notice that the `istio-proxy` container has requested 0.01 CPU and 40 MB memory to start with as well as 2 CPU and 1 GB memory for limits. You will need to budget for these settings when managing the capacity of the cluster. Further, the requested resource and resource limit may vary per installation profile. Also notice that the `istio-certs` are mounted to the pod for the purpose of implementing mTLS, which we explore in Chapter 4.

Onboarding Remaining Stock Trader Services

Now that you have a good understanding for how a single service is onboarded to the mesh, you can deploy the remaining services into the mesh. You can validate that the `trader` and `stock-quote` services meet the Istio mesh requirements. Following the service installation steps, you install the `trader` and `stock-quote` services into the `stock-trader` namespace.

Use the following commands to install and onboard the `trader` service:

```
$ git clone https://github.com/istio-explained/trader.git
$ kubectl apply -f trader/manifests/deploy.yaml -n stock-trader
```

Then, use these commands to install and onboard the `stock-quote` service:

```
$ git clone https://github.com/istio-explained/stock-quote.git
$ kubectl apply -f stock-quote/manifests/deploy.yaml -n stock-trader
```

Run commands that follow to validate that the pods are running and that the Envoy sidecar has been injected by inspecting the number of containers that are running. You should see `2/2`, since the deployment descriptors have a container for each service plus the newly injected `istio-proxy` container for each service.

```
# validate pod has reached running status with sidecar injected
$ kubectl get pods -l app=trader -n stock-trader
NAME                      READY   STATUS    RESTARTS   AGE
trader-7fc498f64-wkn58    2/2     Running   0          13d

# view logs to validate no errors
$ kubectl logs -c trader --namespace=stock-trader \
  --selector="app=trader,solution=stock-trader"

# validate pod has reached running status with sidecar injected
$ kubectl get pods -l app=stock-quote -n stock-trader
NAME                           READY   STATUS    RESTARTS   AGE
stock-quote-57cc4c4d4f-tntjc   2/2     Running   0          3d16h

# view logs to validate no errors
$ kubectl logs -c stock-quote --namespace=stock-trader \
  --selector="app=stock-quote,solution=stock-trader"
```

At this point, you have deployed all of the services in the Stock Trader example with some services associated with the mesh and the data service not included in the mesh.

Accessing the Stock Trader Application

The `trader` service is deployed as a Kubernetes `nodeport` service, which means that the service will be assigned a cluster-unique port. This means that you can access the port using an IP address from any of the worker nodes within the cluster. First, you need to determine an IP address to access the service. You will need to determine a public IP address for one of the worker nodes. If you are using the IBM Cloud Kubernetes Service (IKS), you can find your cluster using this command:

```
$ ibmcloud ks clusters
```

Use the cluster name from the output to view the list of worker nodes in your cluster using this command:

```
$ ibmcloud ks workers --cluster $CLUSTER_NAME
$ export STOCK_TRADER_IP=<public IP of one of the worker nodes>
```

The output of the worker nodes will show the public IP addresses of the worker nodes. If you are using another Kubernetes environment other than IKS, you can try this next command to obtain the IP address from Kubernetes:

```
$ export STOCK_TRADER_IP=$(kubectl get po -l istio=ingressgateway \
-n istio-system -o jsonpath='{.items[0].status.hostIP}')
```

If you are using Minikube, you can try this command to obtain the IP address from Minikube:

```
$ export STOCK_TRADER_IP=$(minikube ip)
```

Confirm that you can access the application via its node-port. Open a web browser and enter the following in the URL field, replacing `$STOCK_TRADER_IP` with the environment variable that you have set previously. Log in using username `stock` and password `trader`.

```
http://$STOCK_TRADER_IP:32388/trader/login
```

Statically Defined Nodeport

Generally, Kubernetes nodeport services are automatically assigned a node port within the cluster, but you may statically define the node port to be used. The key requirement is that the node port must be unique across the cluster. In this case, the `trader` service has a statically defined node port of 32388 for HTTP.

If everything is working properly, the application will load as shown in Figure 3-2.

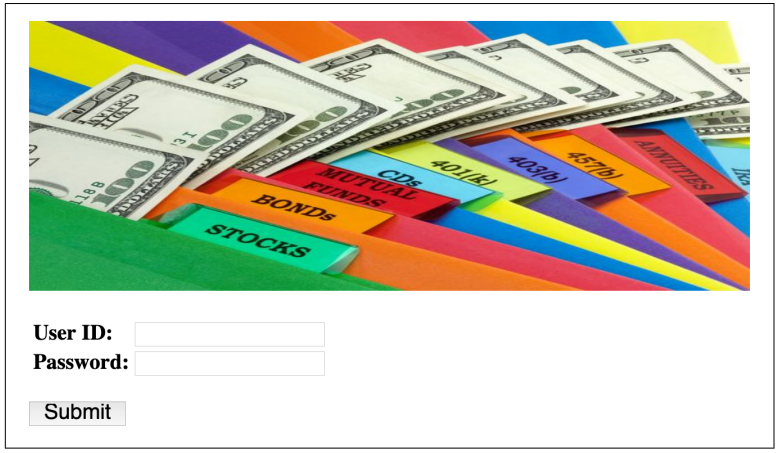

Figure 3-2. Trade application up and running

What Have You Gained?

One of the values of using a service mesh is that you can gain immediate insights into the behaviors and interactions of your services. Istio in particular delivers a set of dashboards that provide you access to important telemetry data that is available just by adding services into the mesh. If you run istioctl dashboard, you will see various monitoring, tracing, and graphical network topology dashboards such as Grafana, Jaeger, Kiali, and Prometheus, which you can easily launch directly from the `istioctl` CLI tool. These dashboard utilities avoid the need to figure out the exact syntax for the Kubernetes `port-forward` command, because they do all the hard work for you. For example, you can launch the Kiali UI using `istioctl dashboard kiali` and log in using the default username `admin` and password `admin` as shown in Figure 3-3.

Note that the external stock service isn't shown in the Kiali workload graph. Due to an Istio issue that the community is actively working to correct, the istio-proxy sidecar that interacts with the external service isn't currently captured in the telemetry data. We expect this issue to be fixed in a future Istio release, and a workaround is provided in Chapter 5 by creating a service entry resource.

NOTE

If You're Not Using the Demo Profile

The demo installation profile is provided as a convenient way to view the breadth of Istio's features. Thus Kiali is configured with a default user ID and password. If you are not using the demo profile, you will need to install the Kiali secret following these instructions, and log into the Kiali UI using the username and password you specified while creating the secret.

You can view distributed tracing information using the Jaeger dashboard, which you can launch using `istioctl dashboard jaeger`, as shown in Figure 3-4. Select the `trader.stock-trader` service to view all traces related to the service.

Click each trace to view the detailed trace spans among the microservices, as shown in Figure 3-5.

When there are errors for some traces, you can click one of the traces that contains errors to investigate the problem. In this case, there is a 500 error return code when the `trader` service called the `portfolio` service, as shown in Figure 3-6. These traces can help you quickly pin down which service(s) to troubleshoot further.

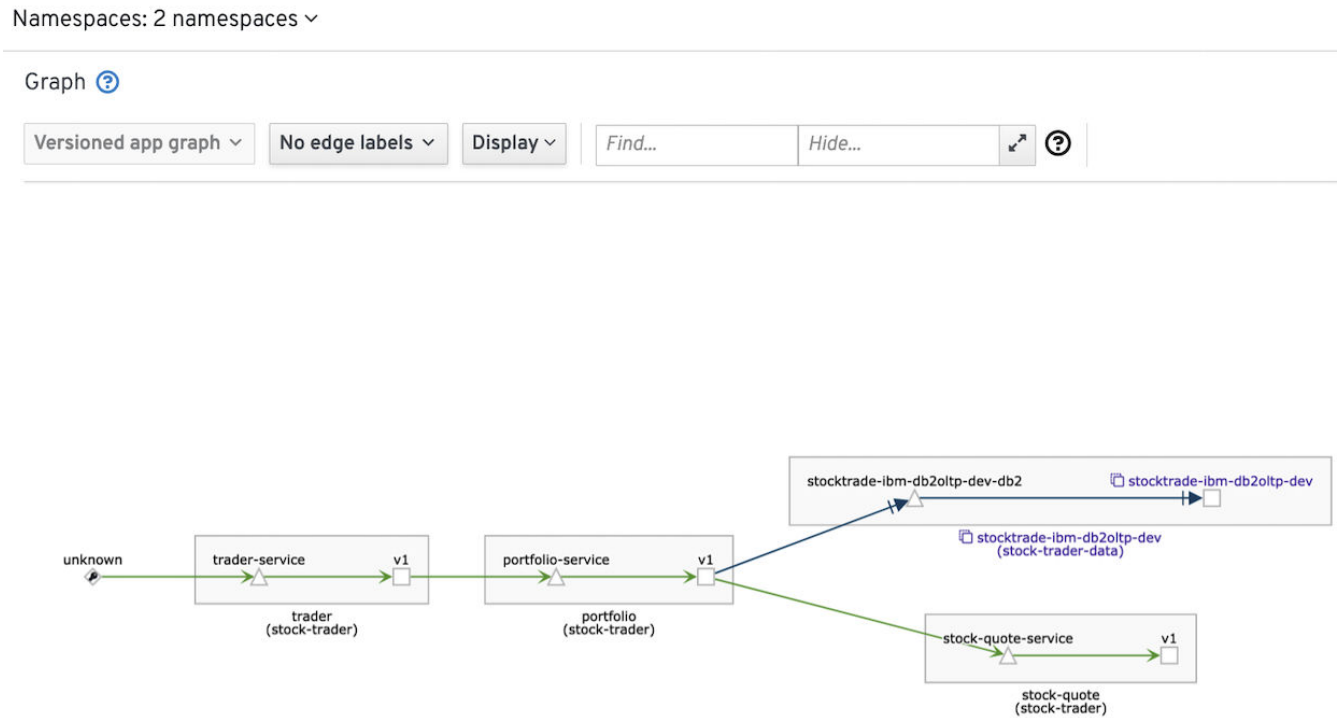

Figure 3-3. Kiali UI: workload graph

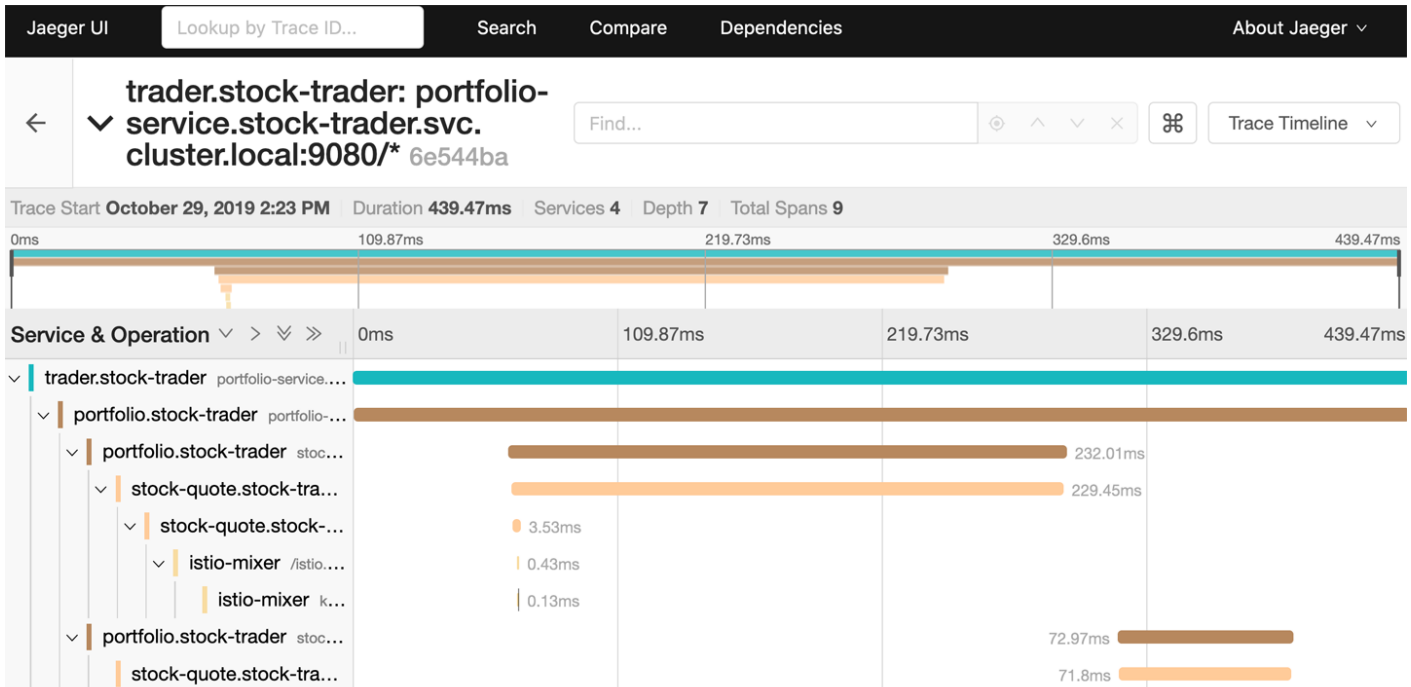

Figure 3-4. Jaeger UI: latest traces

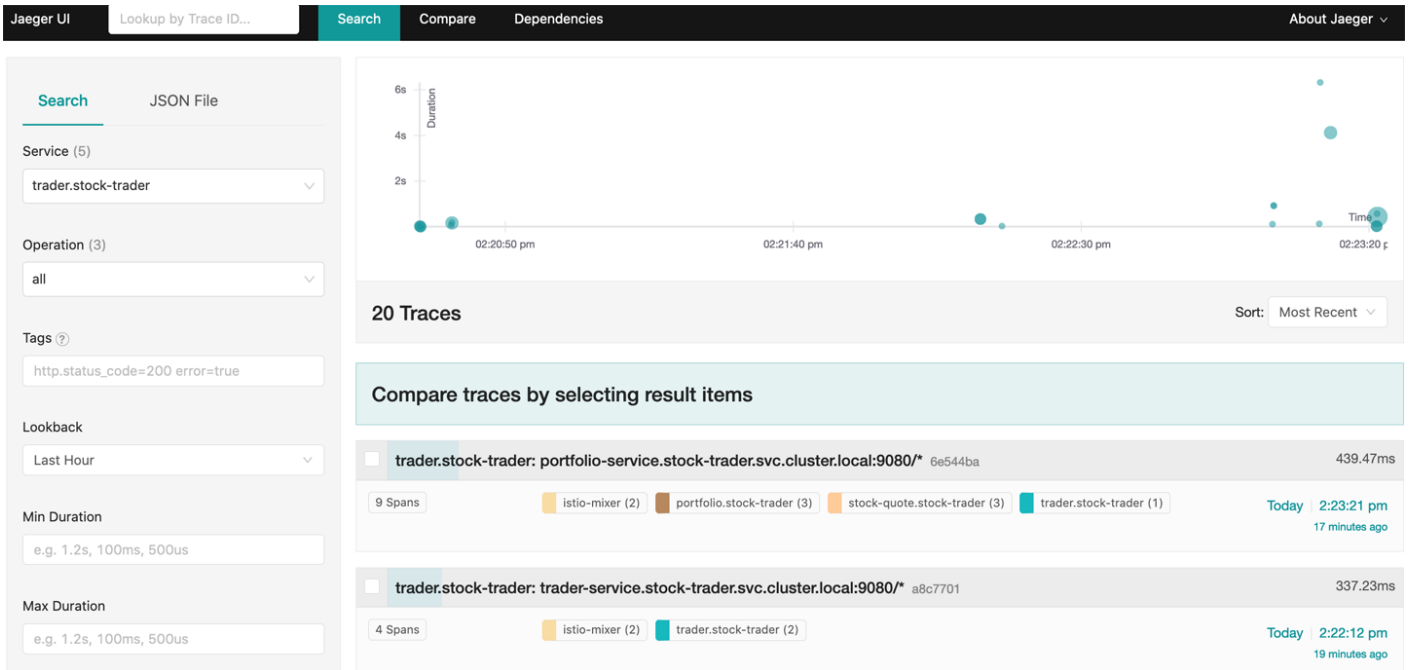

Figure 3-5. Jaeger UI: single request trace spans

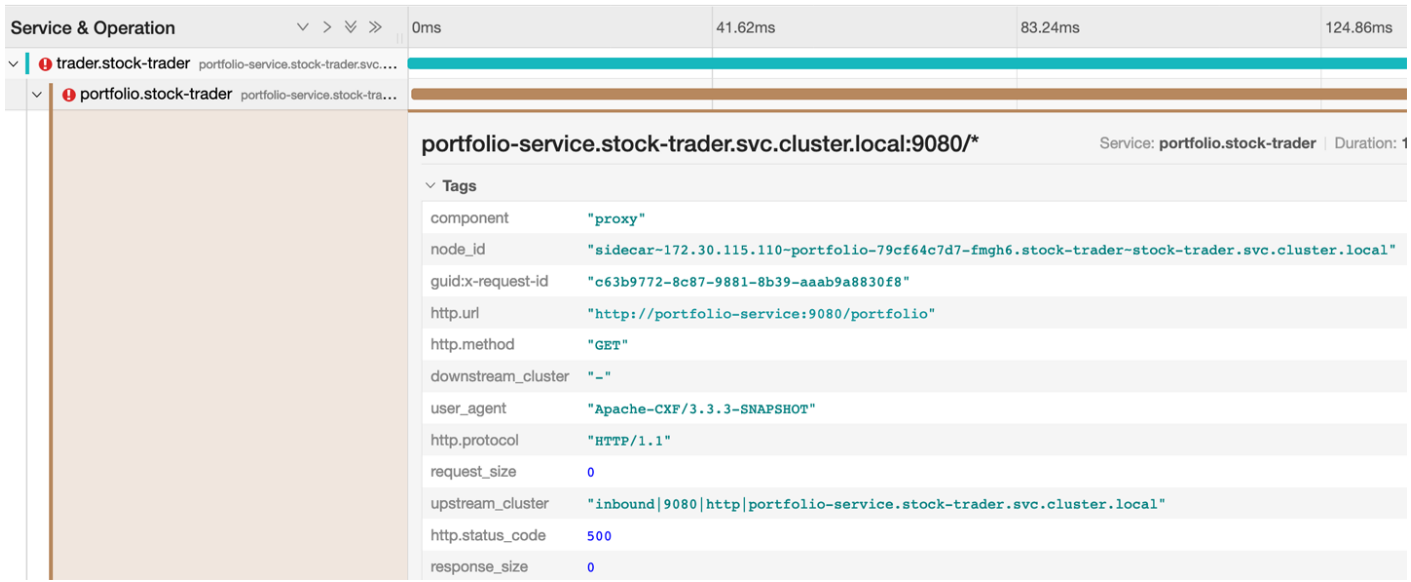

Figure 3-6. Jaeger UI: error trace

Once the issue is fixed, you can validate it in either Kiali or Jaeger. Kiali displays the health status for each service based on the namespaces you choose, which in this case are the `stock-trader` and `stock-trader-data` namespaces (see Figure 3-7).

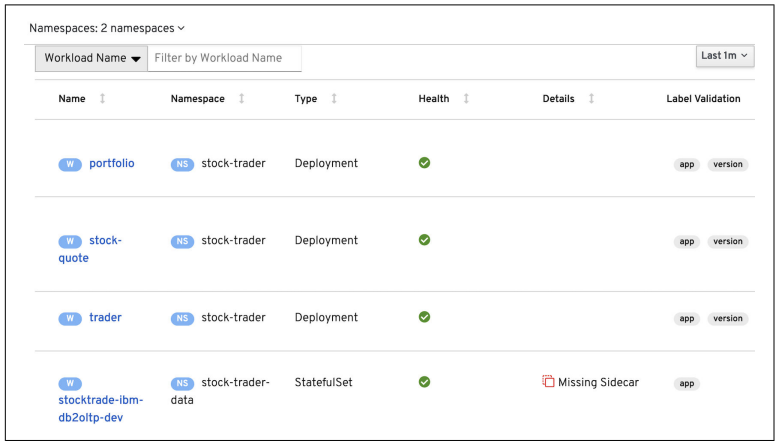

Figure 3-7. Kiali UI: workloads for namespaces

Getting More Out of Traces

Adding services to a service mesh enables rich features such as telemetry and tracing by default without any code changes. However, there are limits to what is possible when the services are participating in a mesh. For example, traces are useful only if the context of the trace can be preserved between services. This is possible only if the context is propagated between the services. Within Istio, you propagate the context by adjusting the header information.

You will need to propagate header context information within your code. This is the one area where it is necessary to adjust the code to ensure that traces are linked together with a common context; otherwise, each trace would be independent, which provides no value when you're trying to debug problems across services. For example, you can modify the code in the *Portfolio.java* file to propagate the context in the headers so that individual trace spans are tied to each request when viewed in the Jaeger dashboard. You can see the highlighted changes in the *Portfolio.java* file in Example 3-1 which show how the headers are copied and forwarded from the request. The `copyFromRequest` method was added to copy the header information from the request over to the response. You can

use the same approach in your own code to forward header information to keep trace spans connected.

Example 3-1. Portfolio.java propagate headers

```
    private static JsonObject invokeREST(HttpServletRequest request, String verb,
String uri, String payload, String user, String password) throws IOException {
        logger.info("Preparing call to "+verb+" "+uri);
        URL url = new URL(uri);
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();

        //forward headers (including cookies) from inbound request
        if (request!=null) copyFromRequest(conn, request);
        ...
    }

    //forward headers (including cookies) from inbound request
    private static void copyFromRequest(HttpURLConnection conn, HttpServletRe-
quest request) {
        logger.info("Copying headers (and cookies) from request to response");
        Enumeration<String> headers = request.getHeaderNames();
        if (headers != null) {
            int count = 0;
            while (headers.hasMoreElements()) {
                String headerName = headers.nextElement(); //"Authorization" and
"Cookie" are especially important headers
                String headerValue = request.getHeader(headerName);
                logger.fine(headerName+": "+headerValue);
                conn.setRequestProperty(headerName, headerValue); //odd it's
called request property here, rather than header...
                count++;
            }
            if (count == 0) logger.warning("headers is empty");
        } else {
            logger.warning("headers is null");
        }
    }
```

Conclusion

In this chapter, you learned that adding services to a service mesh requires little effort and requires no code changes to get valuable telemetry support out of the box. Istio makes it even easier to add services to the mesh by enabling automatic sidecar injection per Kubernetes namespace. More important is that you have the ability to control which services are added to the mesh to enable incremental adoption of services, which is especially important for existing applications. In the next chapter, we explore enabling secure communication between your services, taking into account that not all of the services have been added to the mesh all at once.

CHAPTER 4

Securing Communication Within Istio

A key requirement for many cloud native applications is the ability to provide secure communication paths between services. Traditionally, applications would deploy a "secure at the edge" architecture. Although this approach was sufficient in a monolithic architecture, it has security exposures in a distributed, microservices architecture.

In the previous chapter, we explored including services into a mesh; however, our installation of Istio from Chapter 2 configured a *permissive* security mode. Recall that the Istio permissive security setting is useful when you have services that are being moved into the service mesh incrementally by allowing both plain text and mTLS traffic. A *strict* security setting would force all communication to be secure, which can cause endless headaches if you must incrementally move services of your application into the mesh. In this chapter, we explore how Istio manages secure communication between services, and we investigate enabling strict security communication between some of the services in our sample application.

Istio Security

It is always imperative to secure communication to your application by ensuring that only trusted identities can call your services. In traditional applications, we often see that communication to services is secured at the *edge* of the application, or, to be more explicit, a network gateway (appliance or software) is configured on the network

in which the application is deployed. In these topologies the first line of defense—and often the only line of defense—is at the edge of the network prior to getting into the application. Such a deployment topology exhibits faults when moving to a highly distributed, cloud native solution.

Istio aims to provide *security in depth* to ensure that an application can be secured even on an untrusted network. Security at depth places security controls at every endpoint of the mesh and not simply at the edge. Placing security controls at each endpoint ensures defense against man-in-the-middle attacks by enabling mTLS, encrypted traffic flow with secure service identities. Traditionally, application code would be modified using common libraries and approaches to establish TLS communication to other services in the application. The traditional approach is complex, varies between languages, and relies on the developers to follow development guidelines to enable TLS communication between services. The traditional approach is fraught with errors, and a single error to secure a connection can compromise the entire application. Istio, on the other hand, establishes and manages mTLS connections within the mesh itself and not within the application code. Thus mTLS communication can be enabled without changing code, and it can be done with a high degree of consistency and control ensuring far less opportunity for error. Figure 4-1 shows the key Istio components involved in providing mTLS communication between services in the mesh, including the following:

Citadel
: Manages keys and certificates including generation and rotation.

Istio (Envoy) Proxy
: Implements secure communication between clients and servers.

Pilot
: Distributes secure naming, mapping, and authentication policies to the proxies.

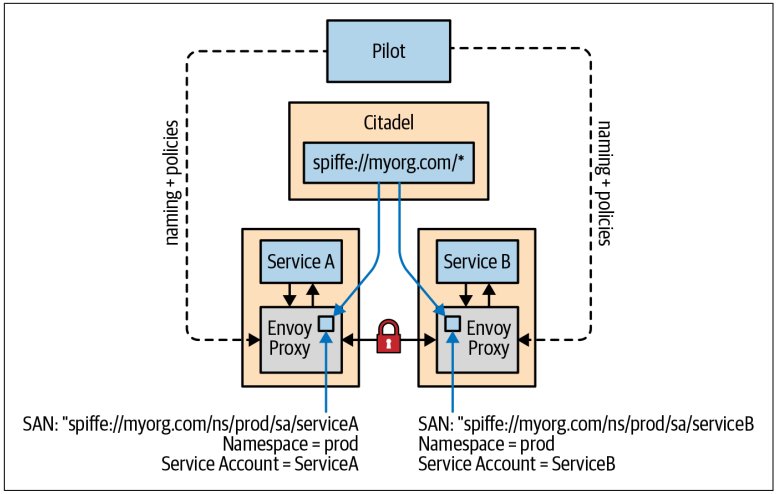

Figure 4-1. Istio secure identity architecture

Istio Identities

A critical aspect of being able to secure communication between services requires a consistent approach to defining the service identities. For mutual authentication between two services, the services must exchange credentials encoded with their identity. In Kubernetes, service accounts are used to provide service identities. Istio uses *secure naming* information on the client side of a service invocation to determine whether the client is allowed to call the server-side service. On the server side, the server is able to determine how the client can access and what information can be accessed on the service using *authorization policies.*

Along with service identities being encoded in certificates, secure naming in Istio will map the service identities to the service names that have been discovered. In simple Kubernetes terms this means a mapping of service account (i.e., the service identity) *X* to a service named *Z* indicates that "service account *X* is authorized to run service *Z*." What this means in practice is that when a client attempts to call service *Z*, Istio will check whether the identity running the service is actually authorized to run the service before allowing the client to use the service. As you learned earlier, Istio Pilot is responsible for configuring the Envoy proxies. In a Kubernetes environment, Pilot will watch the Kubernetes api-server for services being added or removed and generates the secure naming mapping

information, which is then securely distributed to all of the Envoy proxies in the mesh. Secure naming prevents DNS spoofing attacks with the mapping of service identities (service accounts) to service names.

Authorization policies are modeled after Kubernetes Role-Based Access Control (RBAC), which defines roles with actions used within a Kubernetes cluster and role bindings to associate roles to identities, either user or service. Authorization policies are defined using a *ServiceRole* and *ServiceRoleBinding*. A ServiceRole is used to define permissions for accessing services, and a ServiceRoleBinding grants a ServiceRole to subjects that can be a user, a group, or a service. This combination defines *who* is allowed to do *what* under *which* conditions. Here is a simple example of a ServiceRole that provides read access to all services under the `/quotes` path in the `trader` namespace:

```
apiVersion: "rbac.istio.io/v1alpha1"
kind: ServiceRole
metadata:
  name: quotes-viewer
  namespace: trader
spec:
  rules:
  - services: ["*"]
    paths: ["*/quotes"]
    methods: ["GET"]
```

A ServiceRole uses a combination of namespace, services, paths, and methods to define how a service or services are accessed.

A ServiceRoleBinding has two parts, a *roleRef* that refers to a ServiceRole within the namespace, and a *list of subjects* to be assigned to the role. For example, you can define a `ServiceRoleBinding` shown here, to allow only authenticated users and services to view quotes:

```
apiVersion: "rbac.istio.io/v1alpha1"
kind: ServiceRoleBinding
metadata:
  name: binding-view-quotes-all-authenticated
  namespace: trader
spec:
  subjects:
  - properties:
      source.principal: "*"
  roleRef:
    kind: ServiceRole
    name: "quotes-viewer"
```

Citadel

To provide secure communication between services it is necessary to have a public key infrastructure (PKI). Istio's PKI is built on top of a component named Citadel. A PKI is responsible for securing communication between a *client* and a *server* by using public and private cryptographic keys. The PKI creates, distributes, and revokes digital certificates as well as manages public key encryption. Thus, Citadel is responsible for managing keys and certificates across the mesh. A PKI will bind public keys with an identity of a given entity. In the case of Citadel, the entities are the services within the mesh.

Citadel uses the SPIFFE format to construct strong identities for every service encoded in x.509 certificates. SPIFFE (stands for "Secure Production Identity Framework for Everyone") is an open source project that removes the need for application-level authentication and network ACLs by encoding workload identities in specially crafted x.509 certificates. Istio uses SPIFFE Verifiable Identity documents (SVIDs) for the identity documents. The SVID certificate URI field in a Kubernetes environment uses the following format:

```
spiffe://\<domain\>/ns/\<namespace\>/sa/\<serviceaccount>\>
```

Figure 4-1 shows an example of an encoded service identity using SPIFFE in the Subject Alternative Name (SAN) extension field. Using SVID allows Istio to accept connections with other SPIFFE-compliant systems.

Enable mTLS Communication Between Services

Now that you have a basic understanding for how secure communication works in Istio, let's get hands-on to explore enabling mTLS communication between services in our Stock Trader reference application. We start by enabling mTLS communication between the `trader` service and the `portfolio` service. Before getting started, you can validate that the cluster is installed with the global `PERMISSIVE` mesh policy to allow both plain-text and mTLS connections using the following command:

```
$ kubectl describe meshpolicy default
```

Confirm that `PERMISSIVE` is in the mTLS mode:

```
Spec:
  Peers:
    Mtls:
      Mode:  PERMISSIVE
```

You can use the `istioctl authn` command to validate the existing TLS settings both client side and server side for accessing the `portfolio` service from the point of view of a `trader` service pod. Use these commands to check the TLS settings for a `trader` pod (the client) using the `portfolio` service (the server):

```
$ TRADER_POD=$(kubectl get pod -l app=trader -o jsonpath={.items..metadata.name} -
n stock-trader)
$ istioctl authn tls-check ${TRADER_POD}.stock-trader portfolio-service.stock-
trader.svc.cluster.local
```

You should see an output that states that the `portfolio-service` supports both plain-text and mTLS connections (the SERVER column has HTTP/mTLS) defined by the global mesh policy (the AUTHN POLICY column has default/) since there is no destination rule defined:

```
HOST:PORT                                                    STATUS
portfolio-service.stock-trader.svc.cluster.local:9080        OK
SERVER     CLIENT     AUTHN POLICY   DESTINATION RULE
HTTP/mTLS  HTTP       default/       -
```

You can now begin enabling mTLS communication between the services for the Stock Trader application.

Using Kiali

For information about using Kiali, refer to Chapter 3.

We use Kiali to show how communication is visualized between Istio services. Open the Kiali console using the following command, and log in with admin/admin as the default credentials for the Istio demo profile:

```
$ istioctl dashboard kiali
```

You will change some settings to enable security visualization. Start by switching to the Graph tab on the left sidebar. Change the namespace selection to include `stock-trader` and `stock-trader-data` at a minimum, as shown in Figure 4-2.

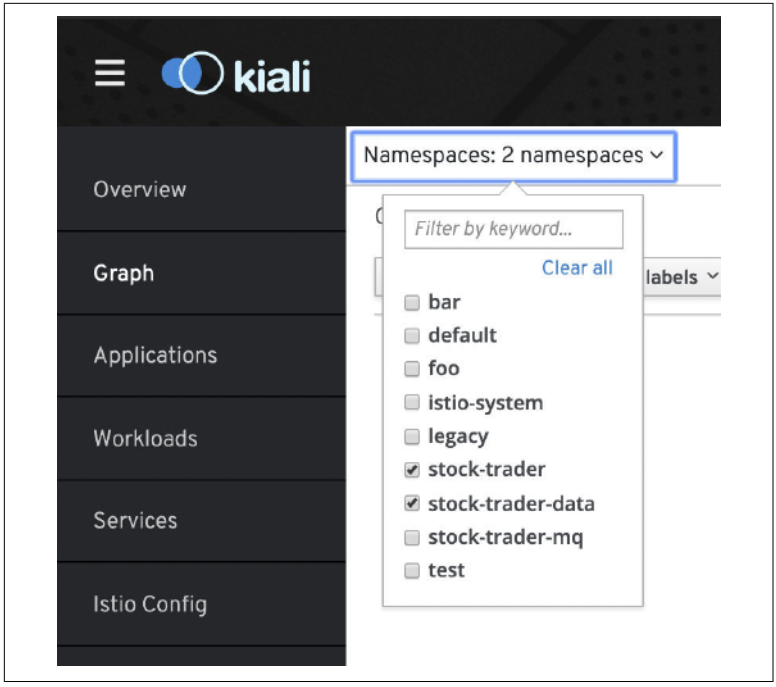

Figure 4-2. Kiali namespace selection

Adjust the Kiali display settings to show Traffic Animation and Security as illustrated in Figure 4-3. You can adjust the type of graph shown to see more or less detail.

At this point Kiali will only show information based on the Istio mesh registration. You will now generate a little load so that you can see how Kiali captures and visualizes traffic between services. In a terminal window, enter the following commands:

```
$ ibmcloud ks workers --cluster $CLUSTER_NAME
$ export STOCK_TRADER_IP=<public IP of one of the worker nodes>
```

NOTE

Remember to Set the STOCK_TRADER_IP Variable

Recall from Chapter 3 that the `STOCK_TRADER_IP` environment variable is set using these commands.

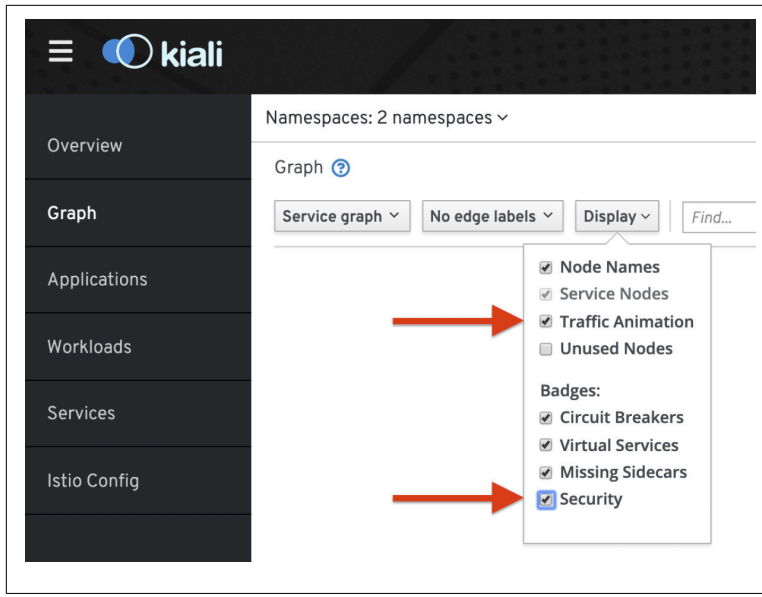

Figure 4-3. Kiali display settings

First, log in to the sample application site, which will be used to generate load against the Stock Trader application. You can use the following cURL command to log in to the Stock Trader application from a terminal to obtain an authentication cookie:

```
$ curl -X POST \
  http://$STOCK_TRADER_IP:32388/trader/login \
  -H 'Content-Type: application/x-www-form-urlencoded' \
  -H 'Referer: http://$STOCK_TRADER_IP:32388/trader/login' \
  -H 'cache-control: no-cache' \
  -d 'id=admin&password=admin&submit=Submit' \
  --insecure --cookie-jar stock-trader-cookie
```

Then, you can generate load against the summary page using the cached cookie:

```
$ while sleep 2.0; do curl -L http://$STOCK_TRADER_IP:32388/trader/summary --
insecure --cookie stock-trader-cookie; done
```

After a couple of minutes, return to the Kiali console, where you should see traffic flowing between the services as shown in Figure 4-4. A green line indicates that traffic is successfully flowing between the services.

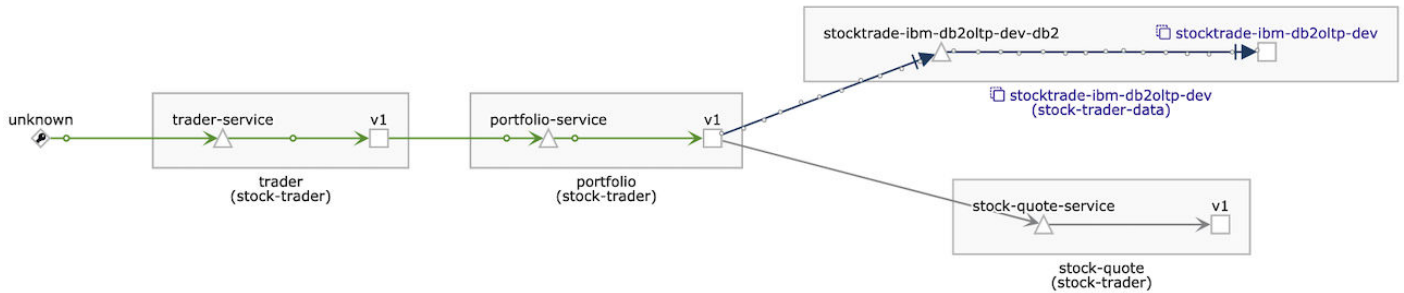

Figure 4-4. Kiali traffic flow

As you can see in both the TLS settings and the Kiali console, you do not have secure communication between the services in the mesh. This means that the clients are sending data in plain text and the servers are accepting the clear text. Next, you will configure the services in the `stock-trader` namespace so that they require client-side mTLS, but they will still tolerate plain text, which is useful for incremental onboarding services into the mesh. You can accomplish this by executing the following command to create a default `DestinationRule` that has a TLS traffic policy set to `ISTIO_MUTUAL` for the `stock-trader` namespace:

```
$ kubectl apply -f - <<EOF
apiVersion: "networking.istio.io/v1alpha3"
kind: "DestinationRule"
metadata:
  name: "default"
  namespace: "stock-trader"
spec:
  host: "*.stock-trader.svc.cluster.local"
  trafficPolicy:
    tls:
      mode: ISTIO_MUTUAL
EOF
```

Default DestinationRule

There can be only one default `DestinationRule` in a given namespace, and it must be named "default." The default `DestinationRule` will override the global setting and provide a default setting for all services defined in the namespace.

Use the following command to execute the `tls-check` command again:

```
$ TRADER_POD=$(kubectl get pod -l app=trader -o jsonpath={.items..metadata.name} -
n stock-trader)
$ istioctl authn tls-check ${TRADER_POD}.stock-trader portfolio-service.stock-
trader.svc.cluster.local
```

You will see that you now have a default namespace scoped `DestinationRule` (default/stock-trader) that applies to the pod being examined. Notice that the results show that the client-side authentication requires mTLS as defined in the default `DestinationRule` in the `stock-trader` namespace, meaning that mesh clients will send encrypted messages. The server-side authentication remains `PERMISSIVE`, accepting both plain text and mTLS from the client due to the mesh-wide default policy:

```
HOST:PORT                                          STATUS   SERVER
trader-service.stock-trader.svc.cluster.local:9080 OK       HTTP/mTLS
CLIENT      AUTHN POLICY     DESTINATION RULE
mTLS        default/         default/stock-trader
```

You should still be generating load in a terminal window from earlier. Switching back to the Kiali console, you should notice that a padlock icon now appears on the traffic being sent between the services, indicating that the messages are encrypted (see Figure 4-5).

You can further lock down the secure access to require both mTLS from the client and server in the `stock-trader` namespace using an authentication `Policy` to remove the `PERMISSIVE` support. Execute this command to define a default policy in the `stock-trader` namespace that updates all of the servers to accept only mTLS traffic:

```
$ kubectl apply -f - <<EOF
apiVersion: "authentication.istio.io/v1alpha1"
kind: "Policy"
metadata:
  name: "default"
  namespace: "stock-trader"
spec:
  peers:
  - mtls: {}
EOF
```

Executing the `tls-check` once again, you see that both client-side and server-side authentication requires mTLS:

```
$ TRADER_POD=$(kubectl get pod -l app=trader -o jsonpath={.items..metadata.name} -
n stock-trader)
$ istioctl authn tls-check ${TRADER_POD}.stock-trader portfolio-service.stock-
trader.svc.cluster.local

HOST:PORT                                          STATUS     SERVER
trader-service.stock-trader.svc.cluster.local:9080 OK         mTLS
CLIENT     AUTHN POLICY            DESTINATION RULE
mTLS       default/stock-trader    default/stock-trader
```

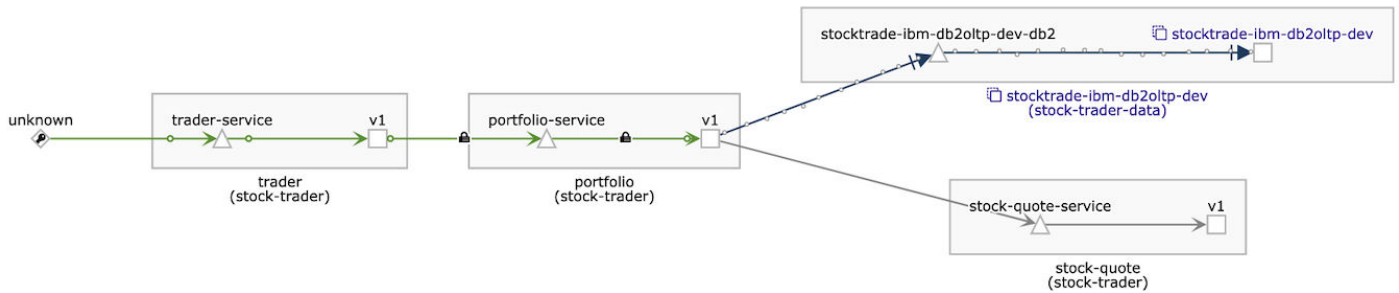

Figure 4-5. Kiali with mTLS communication

NOTE

Managing HTTP Health Checks

The `trader` and `stock-quote` deployments both have Kubernetes HTTP probes configured for checking the health of the containers. Since you enabled only mTLS communication to the servers, this means the Kubernetes probes initiated by the Kubelet on the nodes will fail since the Kubelet isn't using mTLS to communicate with the server. If the Kubernetes probes fail, then the pods will begin to fail. This problem was avoided because the `sidecar.istio.io/rewriteAppHTTPProbers: "true"` pod annotation was already defined in the corresponding *deploy.yaml* files. This annotation enables the rewrite of the HTTP probes without requiring changes to the services. We expect this feature will be enabled by default in a future Istio release.

You will see failures now in your cURL calls from earlier since the server side is requiring clients to send mTLS traffic, which also includes clients from the internet. It will be necessary to secure inbound traffic to the service mesh as well to remove these errors:

```
curl: (56) Recv failure: Connection reset by peer
```

Securing Inbound Traffic

Now that you have walked through the configuration of secure mTLS communication within the mesh, we want to turn your attention to securing communication into the mesh from a client outside the mesh. For example, clients using a web browser want to access a service from the public internet. As you saw in the last section, setting the default `stock-trader` namespace `Policy` to enforce mTLS from clients caused client requests from the internet (i.e., from a web browser) to fail TLS handshake. This is because `PERMISSIVE` support was removed on the microservices within the Istio mesh. To solve this problem, use an Istio Gateway configured with TLS.

Inbound and outbound traffic for the mesh is controlled with Istio gateways. These gateways are implemented as Envoy proxies, which allow or block traffic from entering or leaving the mesh. A mesh can have multiple gateway configurations; for example, you may want one set of gateways for public internet inbound and outbound traffic, while having a separate set of gateways for private network traffic. Istio Gateways are primarily used to provide secure inbound

access to the mesh, but the Istio egress (outbound) gateway also provides critical control over outbound traffic. For example, you can configure an Istio egress gateway with policies to restrict which destinations can be reached by specific services within the mesh. This level of traffic control is quite difficult with Kubernetes itself. Let's start by ensuring there is secure inbound communication by configuring an Istio ingress (inbound) gateway for secure TLS communication to the `trade` service within the mesh.

You can see that a default Istio ingress gateway has already been deployed into our Kubernetes cluster during installation. Using the following command, you can see that the ingress gateway is deployed as a `LoadBalancer` service with an external public IP address:

```
$ kubectl get svc -n istio-system -l app=istio-ingressgateway

NAME                   TYPE           CLUSTER-IP     EXTERNAL-IP
istio-ingressgateway   LoadBalancer   172.21.39.61   169.63.159.157
PORT(S)
15020:31382/TCP,80:31380/TCP,443:31390/TCP,31400:31400/TCP,15029:31133/
TCP,15030:30832/TCP,15031:31732/TCP,15032:32263/TCP,15443:31348/TCP
AGE
15d
```

To configure secure communication via the gateway, you will need a signed certificate. For this example, we'll use a DNS entry with a signed wildcard certificate from IBM Cloud Kubernetes Service (IKS). Refer to your cloud provider's documentation to determine how to configure a DNS entry for your Istio ingress gateway service.

You can use IKS to generate a DNS entry and certificate for the Istio ingress gateway used in the example Stock Trader application. Using the `ibmcloud` CLI, you will register a new DNS entry for the Istio ingress gateway using the external IP of the `istio-ingressgateway` service. You need to set the name of your cluster in an environment variable to be used in later commands. In the following command, make sure that you replace `<YOUR_CLUSTER_NAME>` with the name of the cluster that you created in IKS:

```
$ export CLUSTER_NAME=<YOUR_CLUSTER_NAME>
```

If you do not recall the name of your cluster, you can use the following command to list all the clusters that belong to you:

```
$ ibmcloud ks clusters
```

Now you can register a DNS entry using the IP address of the Istio ingress gateway using these commands:

```
$ export INGRESS_IP=$(kubectl -n istio-system get service istio-ingressgateway -o
jsonpath='{.status.loadBalancer.ingress[0].ip}')
$ ibmcloud ks nlb-dns-create --cluster $CLUSTER_NAME --ip $INGRESS_IP
```

You should see a result similar to the following:

```
OK
Hostname subdomain is created as <YOUR_CLUSTER_NAME>-
f0a5715bb2873122b708ede2bf765701-0001.us-east.containers.appdomain.cloud
```

Check the status of the DNS entry using the IKS `nlb-dns` command, which will have a result similar to this:

```
$ ibmcloud ks nlb-dns ls --cluster $CLUSTER_NAME
Retrieving hostnames, certificates, IPs, and health check monitors for network
load balancer (NLB) pods in cluster <YOUR_CLUSTER_NAME>...
OK
Hostname
istio-book-f0a5715bb2873122b708ede2bf765701-0001.us-
east.containers.appdomain.cloud
IP(s)            Health Monitor   SSL Cert Status
169.63.159.157   None             created
SSL Cert Secret Name
istio-book-f0a5715bb2873122b708ede2bf765701-0001
```

The SSL certificate is encoded in the `SSL Cert Secret Name` Kubernetes secret stored in the default namespace. You will need to copy the secret into the `istio-system` namespace where the gateway service is deployed, and you will need to name the secret `istio-ingressgateway-certs`. The name is a reserved name and will automatically get loaded by the ingress gateway when a secret with the `istio-ingressgateway-certs` is found.

Copy the SSL secret generated by your cloud provider into the `istio-ingressgateway-certs` secret. To do this you'll need to export the secret name using this command where you change `<YOUR_SSL_SECRET_NAME>` to the name generated by your cloud provider:

```
$ export SSL_SECRET_NAME=<YOUR_SSL_SECRET_NAME>
```

Now you can create the `istio-ingressgateway-certs` secret with this command:

```
$ kubectl get secret -n default $SSL_SECRET_NAME --export -o json | jq '.meta-
data.name |= "istio-ingressgateway-certs"' | kubectl -n istio-system create -f -
```

Validate that the secret was created using the following command:

```
$ kubectl get secret istio-ingressgateway-certs -n istio-system
NAME                          TYPE     DATA   AGE
istio-ingressgateway-certs    Opaque    2     3h5m
```

You will need to force the gateway pod(s) to be restarted to pick up the certificates. This is done by deleting the `istio-ingressgateway` pods with this `kubectl` command.

```
$ kubectl delete pod -n istio-system -l istio=ingressgateway
```

You're now ready to configure the gateway using the generated domain name and the signed certificate stored in the secret. You will need to use the generated hostname from the `nlb-dns` when configuring the ingress gateway. The following command provides you with the generated domain name within IKS:

```
$ ibmcloud ks nlb-dns ls --cluster $CLUSTER_NAME
```

The certificate that IKS has generated is a signed wildcard certificate. This means you can add segments to the front of the generated hostname if you want. You will use the following YAML file to configure the gateway with TLS. The `tls` section configures the gateway to use simple TLS authentication, and the certificate and private key need to have the exact paths specified. These paths will be automatically mounted using the `istio-ingressgateway-certs` secret that you created earlier. Apply your gateway configuration with this command:

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: trader-gateway
  namespace: stock-trader
spec:
  selector:
    istio: ingressgateway # use istio default ingress gateway
  servers:
  - port:
      number: 443
      name: https
      protocol: HTTPS
    tls:
      mode: SIMPLE
      serverCertificate: /etc/istio/ingressgateway-certs/tls.crt
      privateKey: /etc/istio/ingressgateway-certs/tls.key
    hosts:
    - "*"
EOF
```

At this point you have secured the ingress gateway, but you haven't defined any services to be accessed via the gateway. Exposing a service outside the mesh requires that a service is bound to the gateway using an Istio `VirtualService` resource. The VirtualService is bound to the gateway using the `gateways` section. In this case the VirtualService is bound to the newly configured `trader-gateway`.

Using this command, you will apply your virtual service configuration:

```
$ kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: virtual-service-trader
spec:
  hosts:
    - '*'
  gateways:
    - trader-gateway
  http:
  - match:
    - uri:
        prefix: /trader
    route:
    - destination:
        host: trader-service
        port:
          number: 9080
EOF
```

Specifying a Hostname

You can specify a hostname instead of using "*" for a given gateway resource and virtual service resource. For example, you can use `<YOUR_DNS_NLB_HOSTNAME>` as the hostname.

You should now be able to access the Stock Trader application in your web browser with a secure connection and no handshake failures. Enter `https://<YOUR_DNS_NLB_HOSTNAME>/trader` in your web browser (see Figure 4-6).

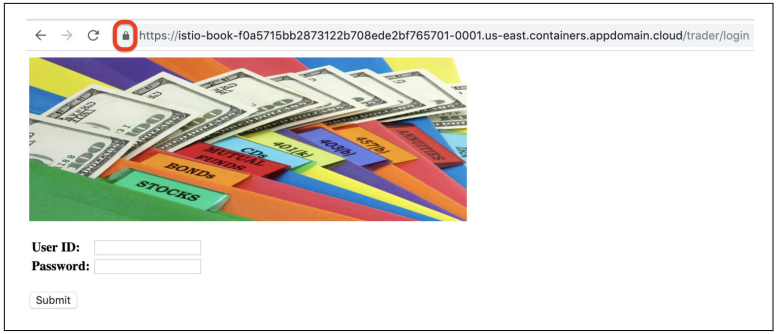

Figure 4-6. Secure access via web browser

When you go back to the Kiali dashboard, you can see that the gateway is now shown and there is a secure connection between the gateway and the `trader`, as shown in Figure 4-7. The gateway itself is configured with TLS ensuring that traffic is encrypted from the client outside the mesh all the way to the target service.

Using RBAC with Secure Communication

Istio also supports authorization policies that leverage ServiceRoles and ServiceRoleBindings to describe who is allowed to do what under which conditions. A ServiceRole describes a set of permissions or methods including paths for accessing a service. A ServiceRoleBinding grants a ServiceRole to a set of subjects, which can be a user or a service. The combination of ServiceRoles and ServiceRoleBindings provide you with fine-grained access controls to services.

Conclusion

In this chapter, you learned that Istio provides a simple yet powerful mechanism to manage mTLS communication between services using strong identities defined with the SPIFFE format. Citadel is the key Istio component responsible for the generation and rotation of keys and certificates used for secure communication between services within the mesh. You learned that Istio uses a declarative model to set security policies enabling the ability to incrementally onboard secure services into the mesh. By using a permissive model, it is possible to have services that support both plain text and mTLS communication, which makes it easier to incrementally move services into the mesh. Using Istio gateways ensures that there are secure, encrypted communication from clients outside of the mesh accessing services that are exposed from the mesh. Now that you have discovered how to secure services in the mesh, we turn your attention toward controlling traffic within the mesh.

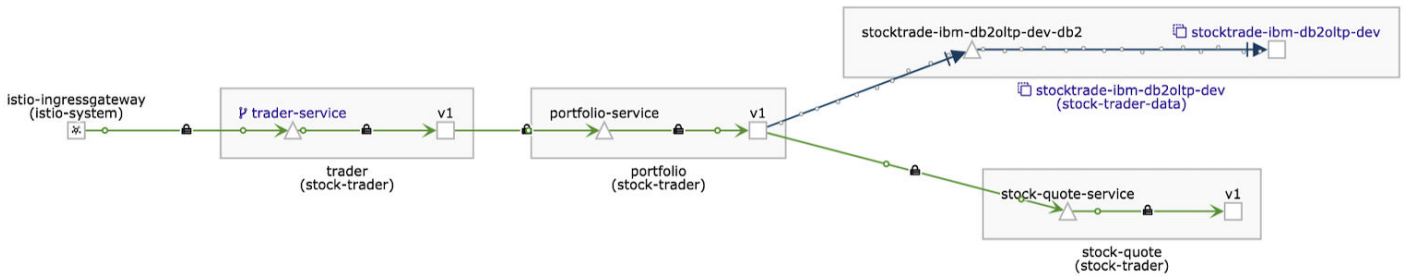

Figure 4-7. Secure gateway communication

CHAPTER 5

Control Traffic

You are now ready to take control of how traffic flows between services. In a Kubernetes environment, there is simple round-robin load balancing between service endpoints. While Kubernetes does support deployment strategies such as a rolling deployment, it is quite coarse grained and is limited to moving to a new version of the service. You may find it necessary to have more than one version of the service running and perform a dark launch or a canary test. A service mesh enables these types of traffic management patterns by controlling requests and resiliency between services and controlling the traffic entering and leaving the cluster. This chapter explores many of these types of features to control the traffic between services including increasing the resiliency between the services.

Dark Launch

Dark launch allows you to deploy a service or a new version of a service while minimizing the impact to users; in other words, you can keep the service in the dark. It is imperative that you can develop and deliver new versions of your application with agility and low risk. Using a dark-launch approach enables you to deliver new functions rapidly with reduced risk. Since Istio allows you to precisely control how new versions of services are rolled out and accessed by clients, you can use a dark-launch approach for delivering changes.

Introducing Changes as a New Version

For example, you may want to create a new version of the Stock Trader service with the loyalty information for each portfolio owner, starting with basic loyalty level with the possibility to generate the loyalty level based on the portfolio's total value. In the Trader GitHub repository, we have already created a v1 branch with the original version of the application and left the master branch for new development.

After you have updated the `trader` service in the master branch, you can update the version value in the deployment labels, selector match labels, and template labels in the *deploy.yaml* file to reflect the v2 version, as shown in the example that follows. Recall from Chapter 3 that version labels were added to the deployment descriptors to provide more context for metrics and telemetry:

```
$ cat trader/manifests/deploy.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: traderv2
  labels:
    app: trader
    solution: stock-trader
    version: v2
spec:
  replicas: 1
  selector:
    matchLabels:
      app: trader
      version: v2
  template:
    metadata:
      labels:
        app: trader
        version: v2
...
```

You can execute the next set of commands if you want to make the changes in your own fork of the GitHub repositories for the Stock Trader example. With these commands you can build a new image with the updated changes:

```
$ git checkout master
$ mvn package

$ docker build -t linsun/trader:v2 .
$ docker push linsun/trader:v2
```

NOTE

Docker Repository

In the preceding example, we used a Docker hub repository that we have made available for use with the example in this book. If you are following along, you'll need to use your own image repository replacing `linsun` in the previous commands and update the *manifests/deploy.yml* descriptor file to use your image repository.

Execute the following command to deploy the v2 changes for the `trader` service into the cluster without deleting the existing v1 deployment where the deployment descriptor changes for v2 were shown in the previous example:

```
# update manifests/deploy.yaml to use the new docker image if needed.
$ kubectl apply -f manifests/deploy.yaml -n stock-trader
deployment.extensions/traderv2 created
```

If you visit the `trader` service's endpoint via *https://<YOUR_DNS_NLB_HOSTNAME>/trader/*, you can see that roughly 50% of the traffic is routed to `trader` v1 deployment endpoints and 50% is routed to `trader` v2 deployment endpoints. Why is this happening? Both deployments have the same number of replicas, and they have the same labels used by the Kubernetes service selector. You can validate the endpoints of the `trader` service using the following Kubernetes endpoints command:

```
$ kubectl get endpoints trader-service -n stock-trader
NAME             ENDPOINTS                                      AGE
trader-service   172.30.244.107:9080,172.30.244.107:9443   7h5m
```

You should see endpoints from both the v1 deployment as well as the v2 deployment. Since Kubernetes deals with connections, not requests, and supports only round-robin load balancing, you will see traffic routed between both endpoints for the service. We could have defined a separate service for the v2 changes, but then clients would need to be aware of the version to use, which breaks the transparency required of the service provider to control how they introduce changes. This is not the behavior you would like to see since you haven't tested `trader` v2. You want to precisely control how changes are exposed to clients by managing how traffic is routed to the `trader` v2 deployment replicas.

Basic Traffic Routing

Istio provides fine-grained traffic routing controls for both the client and destination service using the Istio virtual services and destination rules. A virtual service provides you with the ability to configure a list of routing rules that control how the Envoy sidecar proxies route requests to a service within the service mesh. In this example, you can define a virtual service to define the routing rules that would be used when invoking the `trader` service. Since the `trader` service is an edge service (i.e., a service accessible from outside the mesh), you'll need to bind the virtual service to the `trader-gateway` to describe the route rules from the gateway to the `trader` service. Using this approach, you are able to dark launch the `trader-v2` deployment changes. Inspect the `virtual-service-trader` VirtualService using the next command to see the configured *route*, which sends 100% of the requests to the destination *trader-service* within the mesh. With this virtual service definition, none of the requests to the `trader-service` would be routed to the v2 deployment endpoints because all the requests are routed to the pods with the "v1" subset label:

```
$ cat manifests/trader-vs-100-v1.yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: virtual-service-trader
spec:
  hosts:
    - '*'
  gateways:
    - trader-gateway
  http:
  - match:
    - uri:
        prefix: /trader
    route:
    - destination:
        host: trader-service
        subset: "v1"
        port:
          number: 9080
      weight: 100
```

A destination rule allows you to define configurations of policies that are applied to a request after the routing rules are enforced as defined in the destination virtual service. The destination rule is also used to define the set of Kubernetes pods or VMs that belong to a subset grouping. In this example, the Istio ingress gateway is the client, and the `trader` service is the destination service. A destination

rule is used to identify multiple versions of a service, which are called "subsets" in Istio.

TLS Settings

In Chapter 4, you enabled a default destination rule within the namespace for the mTLS settings. When you define a destination rule for a destination virtual service, you need to specify the TLS settings because the declaration of the destination rule for the virtual service will override the default mTLS settings, either global or namespace scoped.

If you look at the *trader-dr.yaml* file from the example Stock Trader application that follows, you'll see that the *destination-rule-trader* resource exposes two subsets, "v1" and "v2," based on labels found in the destination *trader-service*. The `destination-rule-trader` shown in the example is an extremely simple case that uses the default round-robin load-balancing strategy. It is common to have other rules such as load balancing, connection pool size, and outlier detection settings to detect and evict unhealthy hosts from the load-balancing pool:

```
$ cat manifests/trader-dr.yaml
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
 name: destination-rule-trader
spec:
 host: trader-service
 trafficPolicy:
   tls:
     mode: ISTIO_MUTUAL
 subsets:
 - name: v1
   labels:
     version: "v1"
 - name: v2
   labels:
     version: "v2"
```

You can apply the virtual service and destination rule resources, which will program the Istio mesh and change the routing behavior. Execute the commands that follow to essentially instruct Pilot to program the Istio ingress gateway to route requests on port 443 via https and a URI path with */trader* to the v1 subset of the *trader-service*. Note, you do not need to redeploy either the `trader` v1 or v2 deployments or the `trader` service for the changes to take effect:

```
$ kubectl apply -f manifests/trader-vs-100-v1.yaml -n stock-trader
virtualservice.networking.istio.io/virtual-service-trader configured

$ kubectl apply -f manifests/trader-dr.yaml -n stock-trader
destinationrule.networking.istio.io/destination-rule-trader configured
```

At this point, you have successfully dark launched the v2 of the `trader` deployment without any impact to the `trader` service. If you visit the `trader` service's endpoint via *https://<YOUR_DNS_NLB_HOSTNAME>/trader/*, you will always see v1 of the `trader` deployment. The v2 `trader` deployment does not have any requests routed to it based on the programming rules and policies that you have just defined with an Istio virtual service and destination rule.

NOTE

Accessing from a Nodeport or Load Balancer

If you send a request to the Stock Trader application from a Kubernetes node port address like you did in Chapter 3, you will find that the routing rules are not applied. The same is true if a service is exposed via a load-balancer service. This is expected because the route rule is from the `istio-gateway` to the trade service, so the entry point has to go through the `istio-gateway` for the route rule to be effective.

Selectively Route Requests

Alternatively, you may want to launch the v2 of the `trader` deployment only for a specific client or user and route the other client requests to the v1 deployment. In this example, you use specific support in a virtual service to create HTTP routing rules, which gives you the ability to introspect and use features from the HTTP request such as header information. When you inspect the `virtual-service-trader` virtual service in the *trader-vs-test.yaml* file shown in the following example, you will see that an HTTP route rule has been defined to route requests from clients using a Firefox browser to the v2 subset of the `trader-service`. All other client requests will continue to use the v1 subset of the `trader-service`:

```
$ cat manifests/trader-vs-test.yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: virtual-service-trader
spec:
  hosts:
```

```
    - '*'
  gateways:
    - trader-gateway
  http:
  - match:
    - headers:
        user-agent:
          regex: '.*Firefox.*'
      uri:
        prefix: /trader
    route:
    - destination:
        host: trader-service
        subset: "v2"
        port:
          number: 9080
  - match:
    - uri:
        prefix: /trader
    route:
    - destination:
        host: trader-service
        subset: "v1"
        port:
          number: 9080
```

After reviewing the changes to the `virtual-service-trade` resource, you can apply the changes to your mesh using this command:

```
$ kubectl apply -f manifests/trader-vs-test.yaml -n stock-trader
virtualservice.networking.istio.io/virtual-service-trader configured
```

Namespace Scoped

Notice that you have deployed the `trader-gateway`, the `virtual-service-trader`, and the `destination-rule-trader` resources in the same `stock-trader` namespace. It isn't necessary to have the gateway resource in the same namespace as the virtual service resource. However, it is necessary in this case, because the `virtual-service-trader` resource is referring to the `trader-gateway` without a namespace. Therefore the references are all scoped to the local namespace.

With the new mesh configurations applied, you can test the effects with the sample application. Open a Firefox browser and visit the *https://<YOUR_DNS_NLB_HOSTNAME>/trader* URL. When using Firefox as the client browser to visit the application, you will see your requests are routed to the v2 deployment of the `trader` service, as demonstrated in Figure 5-1. Open another web browser client such as Chrome or Safari. Now visit the same *https://*

<YOUR_DNS_NLB_HOSTNAME>/trader URL. You should now see requests being routed to the v1 deployment of the `trader` service.

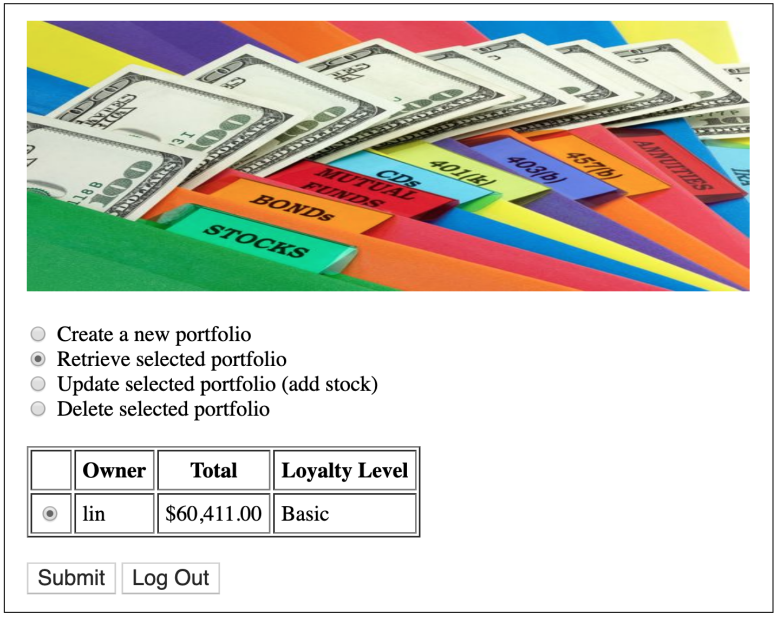

Figure 5-1. Trader v2 after login using Firefox

Canary Testing

A canary test[1] is when you deploy a new version (the canary) along with the previous version and route a percentage of requests to the new version to determine whether there are problems before routing all traffic to the new version. After satisfactorily testing a new feature with a selective set of requests, a canary test is often performed to ensure that the new version of the service not only functions properly, but also doesn't cause a degradation in performance or reliability. You may even place higher load on the canary deployment monitoring the effects over time. If there are no observed ill effects on the environment, you would adjust the routing rules to direct all of the traffic to the canary deployment.

1 The term comes from coal mining; miners took canary birds into the mine since the birds would be affected by carbon monoxide before the miners, thus giving crucial advance warning about the problem.

You can use Istio's virtual service with its weighted routing feature to configure the percentage of requests that are sent to the stable subset versus the new canary subset of the `trader` service. You can accomplish this weighted distribution by using the desired weight for the v1 subset, such as 80% of the requests and a weight for v2 canary subset to be the remaining 20% of requests. You can see the weighted distributions for the `trader` service VirtualService definition here:

```
$ cat manifests/trader-vs-80-20.yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: virtual-service-trader
spec:
  hosts:
    - '*'
  gateways:
    - trader-gateway
  http:
    - route:
        - destination:
            host: trader-service
            subset: "v1"
            port:
              number: 9080
          weight: 80
        - destination:
            host: trader-service
            subset: "v2"
            port:
              number: 9080
          weight: 20
```

You can now deploy the updated virtual service definition for the `trader` service using the following command. There is no need to redeploy either versions of the `trader` deployments to change to the desired traffic distribution because Istio is dynamically reconfiguring the Envoy sidecars within the mesh:

```
$ kubectl apply -f manifests/trader-vs-80-20.yaml -n stock-trader
virtualservice.networking.istio.io/virtual-service-trader configured
```

Visit the `trader` service in your favorite web browser using *https://<YOUR_DNS_NLB_HOSTNAME>/trader/login*. Validate that 20% of the requests go to Trade v2 while 80% of the requests continue to show the Trade v1 UI. You will not immediately see the 80/20 distribution mix until the Envoy sidecars have processed the configuration change and adjusted the routing distribution. You may need to refresh */trader/login* multiple times, perhaps 15 or more to see the proper distribution.

NOTE

Canary Without Istio

Container orchestration platforms such as Kubernetes use instance scaling to manage traffic routing between deployments as well as the number of replicas to control the weight between the deployment endpoints.

With Istio, you can have multiple versions of the `trader` service deployed at the same time and allow them to scale up and down independently, without affecting the traffic distribution between them. As a result, you can scale up or down either version of the `trader` service without worrying about causing an impact to the traffic distribution among the versions of the service. Istio allows you to decouple deployments from traffic routing.

Resiliency and Chaos Testing

When you build a distributed application designed for the cloud, it is critical to ensure that the services within your application are resilient to failures in the underlying platform as well as failures due to dependent services. Kubernetes provides capabilities that allow you to increase the resiliency of your container-based components against failures in the underlying infrastructure; however, it's up to you to ensure that your service implementations are resilient to failures from other services. To increase the resiliency of your services, it is common to set up retries, timeouts, and circuit breakers when calling dependent services. Luckily, you do not need to modify your existing code to have logic for retries and timeouts. Service meshes generally have constructs that allow you to program resiliency between your services. Istio has support for retries, timeouts, and circuit breakers, and even has capabilities that you can use to inject faults into your service calls to help test and tune your timeouts.

Retries

Istio has support to program retries for your services in the mesh without you specifying changes to your code. By default, client requests to each of your services in the mesh will be retried twice. When using Istio, you can configure the number of retries and the timeout for each retry from the point of view of a client that is calling the service. You can configure retries per service within the Istio

virtual service resource, and you can have a different set of retries per route for each virtual service.

Istio will execute two retries per request by default. You can adjust the number of retires or disable them altogether when automatic retries don't make sense for your microservices. If you look at the *trader-vs-retries.yaml* file from the example Stock Trader application shown in the following example, you'll see that you can adjust the route rule within the `virtual-service-trader` resource to control retries. Notice that a `retries` configuration section has been added to disable clients retry requests when accessing the `trader` service:

```
$ cat manifests/trader-vs-retries.yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: virtual-service-trader
spec:
  hosts:
    - '*'
  gateways:
    - trader-gateway
  http:
  - match:
    - uri:
        prefix: /trader
    route:
    - destination:
        host: trader-service
        port:
          number: 9080
    retries:
      attempts: 0
```

To set the new retries configuration, you can apply the *trader-vs-retries.yaml* file using this command:

```
$ kubectl apply -f manifests/trader-vs-retries.yaml -n stock-trader
virtualservice.networking.istio.io/virtual-service-trader configured
```

Timeouts

Istio has built-in support for timeouts with client requests to services within the mesh. The default timeout for HTTP requests is 15 seconds. You can override the default timeout setting of a service route within the route rule for a virtual service resource. For example, in the route rule within the `virtual-service-trader` resource, you can add the following `timeout` configuration to set the timeout of the `/trader` route to be 10 seconds, along with 3 retries with each retry timeout after 2 seconds:

```
$ cat manifests/trader-vs-retries-timeout.yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: virtual-service-trader
spec:
  hosts:
    - '*'
  gateways:
    - trader-gateway
  http:
  - match:
    - uri:
        prefix: /trader
    route:
    - destination:
        host: trader-service
        port:
          number: 9080
    retries:
      attempts: 3
      perTryTimeout: 2s
    timeout: 10s
```

To see the new timeouts and retries in action, you can use this command to apply the *trader-vs-retries-timeout.yaml*:

```
$ kubectl apply -f manifests/trader-vs-retries-timeout.yaml -n stock-trader
virtualservice.networking.istio.io/virtual-service-trader configured
```

Visit the `trader` service in a web browser using *https://<YOUR_DNS_NLB_HOSTNAME>/trader/login*. You might begin to notice upstream request timeout errors when you view an owner's portfolio if it takes more than 10 seconds to obtain the portfolio information. You can undo the custom `timeout: 10s` configuration and return to the default 15-second timeout by executing this command to apply the *trader-vs-retries.yaml*:

```
$ kubectl apply -f manifests/trader-vs-retries.yaml -n stock-trader
virtualservice.networking.istio.io/virtual-service-trader configured
```

Circuit Breakers

Circuit breaking is an important pattern for creating resilient microservice applications. Circuit breaking allows you to limit the impact of failures and network delays, which are often beyond your control when making requests to dependent services. Generally, it would be necessary to add logic directly within your code to handle situations when the calling service fails to provide the desirable result. You would code logic to capture the failure and make decisions on the proper course of action, which would provide a more desirable result to the client rather than an error message.

Istio allows you to apply circuit breaking configurations within a destination rule resource, without any need to modify your microservices code. You will use a `connectionPool` to define circuit-breaking conditions on each individual upstream host for either HTTP or TCP requests. For example, you can define the maximum number of pending requests and maximum requests per connection. The Istio outlier detection is a circuit breaking concept to configure failure settings for each upstream host, which is used to remove a failing upstream host from the load-balancing pool for the defined period of time before trying again.

Take a look at the *trader-dr-cb.yaml* file from the Stock Trader application shown in the example that follows. The file defines the destination rule for the `trader-service`. Within the traffic policy of the `destination-rule-trader`, you can specify the connection pool configuration to indicate the maximum number of TCP connections, specify the maximum number of HTTP requests per connection, and set the outlier ejection to be three minutes after a single error. These settings are used to configure the circuit-breaker behavior when a client accesses the `trader-service`:

```
$ cat manifests/trader-dr-cb.yaml
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
 name: destination-rule-trader
spec:
 host: trader-service
 subsets:
 - name: v1
   labels:
     version: "v1"
 - name: v2
   labels:
     version: "v2"
 trafficPolicy:
   tls:
     mode: ISTIO_MUTUAL
   connectionPool:
     tcp:
       maxConnections: 1
     http:
       http1MaxPendingRequests: 1
       maxRequestsPerConnection: 1
   outlierDetection:
     consecutiveErrors: 1
     interval: 1s
     baseEjectionTime: 3m
     maxEjectionPercent: 100
```

Fault Injection

It is difficult to get service timeouts and circuit-breaker configurations properly set in a distributed microservice application, as seen when you move to a cloud native application. Istio makes it easier to get these settings correct by enabling you to inject faults into your service requests without the need to modify your code. The fault injection support makes it possible to perform chaos testing of your application. You can see how fault injection affects your application by adding an HTTP delay fault into the `stock-quote` service for only a specific user's portfolio so that the injected fault doesn't affect all user flows.

First, you need to ensure that there is match condition we can leverage when the `portfolio` service calls the `stock-quote` service. One common match condition is header based. The *Portfolio.java* file in Example 5-1 has been modified to create a header called `portfolio_user` that is passed along with the HTTP header when calling the `stock-quote` service.

Example 5-1. Modify Portfolio.java to add the `portfolio_user` header

```
...
        if (request!=null) {
            //forward headers (including cookies)
            // from inbound request
            copyFromRequest(conn, request);

            // add a portfolio_user header
            addPortfolioUserHeader(conn, owner);
        }
...

    //add portfolio_user here to specify rules for the user
    private static void addPortfolioUserHeader(HttpURLConnection conn, String
user) {
        logger.info("Adding portfolio_user header for user " + user);
        conn.setRequestProperty("portfolio_user", user);
    }
```

Second, you will inject a 90-second fault delay for 100% of the client requests when the `portfolio_user` HTTP header value exactly matches the value `Jason`. Using a fault injection such as this allows you to minimize the impact to most client requests since you are injecting the failure only on a specific client request, like so:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: virtual-service-stock-quote
spec:
  hosts:
  - stock-quote-service
  http:
  - fault:
      delay:
        fixedDelay: 90s
        percent: 100
    match:
    - headers:
        portfolio_user:
          exact: Jason
    route:
    - destination:
        host: stock-quote-service
        port:
          number: 9080
  - route:
    - destination:
        host: stock-quote-service
        port:
          number: 9080
```

Apply the virtual service and destination rule changes using the following commands to view the effects of the fault injection:

```
$ cd ..
$ kubectl apply -f  stock-quote/manifests/stock-quote-vs-fault-match.yaml \
-n stock-trader
virtualservice.networking.istio.io/virtual-service-stock-quote created
```

To test the new fault-injection settings that you created using the preceding steps, you'll need to use the sample Stock Trader application and a user portfolio with an owner named "Jason." You can visit the Stock Trader application in a web browser using *https://<YOUR_DNS_NLB_HOSTNAME>/trader* and log in as user `stock` with password `trader`. Once you have logged in to the application in your web browser, complete the following steps to see the fault injection in action:

1. Create a new portfolio called Jason if one does not exist.
2. Select the portfolio with the owner named Jason and select the "Retrieve selected portfolio" radio box, as shown in Figure 5-2.
3. Click the Submit button.

- Create a new portfolio
- Retrieve selected portfolio
- Update selected portfolio (add stock)
- Delete selected portfolio

	Owner	Total	Loyalty Level
○	linsun	$67,985.00	Basic
◉	Jason	$76,970.00	Basic

Submit Log Out

Figure 5-2. Retrieving the selected portfolio for Jason

Since the HTTP delay was injected earlier, it will take 90 seconds to get a response. In this case, an error occurs that needs to be fixed in the `trader` or `portfolio` service to ensure that it can handle the network degradation failure properly and serve a useful message to the client.

Create another user portfolio with a name other than "Jason." Following the same steps as above, retrieve the new user's portfolio. You will find that the request succeeds because the fault injection is not applied to portfolios without the user name "Jason."

NOTE

No Destination Rule for stock-quote Service

In this example, you did not specify a corresponding destination rule for the `virtual-service-stock-quote` resource because there is only one version of the `stock-quote` deployment; thus, subsets do not need to be defined in a destination rule.

Controlling Outbound Traffic

When you use Kubernetes, any application pod can make calls to services that are outside the Kubernetes cluster unless there is a Network Policy that prevents calling the target service. However, Network Policies are restricted to Layer 4 rules, which means that they can allow or prevent access only to specific IP addresses. Often, you might want more control over how applications within the mesh can reach external services using Layer 7 (URL-based) policies and more fine-grained attribute policy evaluation.

By default, Istio allows all outbound traffic to ensure users have a smooth starting experience. If you choose to restrict all outbound traffic across the mesh, you can update the `global.outboundTrafficPolicy.mode` installation option setting to enable restricted outbound traffic access:

1. Validate that your Istio installation is configured with the `global.outboundTrafficPolicy.mode` option set to `ALLOW_ANY` using this command:

   ```
   $ kubectl get configmap istio -n istio-system -o yaml \
   | grep -o "mode: ALLOW_ANY"
   mode: ALLOW_ANY
   ```

   The string `mode: ALLOW_ANY` should appear in the output if it is enabled.

2. You can change Istio's setting to block all outbound traffic by default. Run the following command to change the `global.outboundTrafficPolicy.mode` option to `REGISTRY_ONLY`:

   ```
   $ kubectl get configmap istio -n istio-system -o yaml \
   | sed 's/mode: ALLOW_ANY/mode: REGISTRY_ONLY/g' | \
   kubectl replace -n istio-system -f -
   configmap "istio" replaced
   ```

The `stock-quote` service needs to reach out to the IEX Cloud external service to get the current quote of the stock. Go back to the Stock Trader application and add a new stock to one of the portfolios, using a different stock symbol to ensure that the `stock-quote` service must call the IEX Cloud external service to get the most recent stock price. Figure 5-3 shows an example view of using the Stock Trader application to purchase shares into a portfolio.

Figure 5-3. Add stock to portfolio

Because all outbound traffic is blocked by default, you will see the following exception thrown by application class:

```
org.apache.cxf.microprofile.client.DefaultResponseExceptionMapper.toThrowable:33

javax.ws.rs.WebApplicationException: HTTP 500 Internal Server Error
    at org.apache.cxf.microprofile.client.DefaultResponseExceptionMapper
        .toThrowable(DefaultResponseExceptionMapper.java:33)
    at [internal classes]
    at com.sun.proxy.$Proxy90.updatePortfolio(Unknown Source)
    at com.ibm.hybrid.cloud.sample.stocktrader.trader.AddStock.doPost
        .(AddStock.java:138)
    at javax.servlet.http.HttpServlet.service(HttpServlet.java:706)
    at [internal classes]
```

You can examine the `stock-quote` pod log to get more detailed information for the connection failure by entering the following in your terminal:

```
$ kubectl logs -c stock-quote --namespace=stock-trader \
  --selector="app=stock-quote,solution=stock-trader"
```

You should see in the log an entry similar to the example that follows, which indicates a connection problem with the IEX Cloud external service. This is expected because we have restricted any service in the mesh from accessing any other service that is external to the mesh:

```
{"type":"liberty_message","host":"stock-quote-78848589c6-cgqvh",
"ibm_userDir":"\/opt\/ol\/wlp\/usr\/","ibm_serverName":"defaultServer",
"message":"javax.ws.rs.ProcessingException:
javax.net.ssl.SSLHandshakeException: SSLHandshakeException
```

```
invoking https:\/\/cloud.iexapis.com\/stable\/stock\/VT\/quote:
Remote host closed connection during handshake
```

Istio has the ability to selectively access external services using a Service Entry. A Service Entry allows you to define a service that is external to the mesh and allows access by services within the mesh. Through service entries, you can bring external services as participants in the mesh. Create a service entry to ensure services can access the IEX Cloud external service while still preventing access to all other external services, like so:

```
$ cat stock-quote/manifests/se-iex.yaml
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: iex-service-entry
spec:
  hosts:
  - "cloud.iexapis.com"
  ports:
  - number: 443
    name: https
    protocol: https
  resolution: DNS
```

Apply the service entry resource into the `stock-trader` namespace:

```
$ kubectl apply -f stock-quote/manifests/se-iex.yaml -n stock-trader
serviceentry.networking.istio.io/iex-service-entry created
```

Repeat the same step as earlier to revisit the Stock Trader application to add a new stock to a portfolio. This time, you should see that the stock is successfully added given that the call to the IEX Cloud external service is no longer being blocked.

Similar to intercluster requests, Istio routing rules can be used with external services to define retries, timeouts, and fault injection policies. For example, you can set a timeout rule on calls to the *api.us.apiconnect.ibmcloud.com* service used in the Stock Trader application as shown here:

```
$ cat stock-quote/manifests/iex-vs.yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: iex-virtual-service
spec:
  hosts:
    - "cloud.iexapis.com"
  https:
  - timeout: 3s
    route:
      - destination:
          host: "cloud.iexapis.com"
        weight: 100
```

Run the following command to apply the virtual service:

```
$ kubectl apply -f stock-quote/manifests/iex-vs.yaml -n stock-trader
virtualservice.networking.istio.io/iex-virtual-service created
```

NOTE

Egress Gateway

Although service entries provide controlled access to external services, when combined with an Istio egress gateway, you can ensure that all external services are accessed through a single exit point. Having a single exit point allows you to provide specific security constraints on the nodes as well as the pods that all traffic leaving the mesh will pass. Refer to the official egress tasks for more details on using egress gateway.

Conclusion

When you move to developing a cloud native solution, the distributed nature of the services requires greater control over the flow of traffic between the services. Basic routing and load-balancing support in Kubernetes often falls short of what is needed to manage traffic in these highly distributed applications. A service mesh like Istio has the capabilities that provides you with the ability to manage traffic flows within the mesh as well as entering and leaving the mesh. These capabilities allow you to efficiently control rollout and access to new features, and they make it possible to build more resilient services within the mesh, all without having to make complicated changes to your application code.

CHAPTER 6
Wrap-Up

A key point that we have echoed throughout this book is that deploying and managing microservices is difficult, especially within a cloud environment. There is an increased use of containers when using microservices with the cloud, and the increased use of containers further complicates your ability to understand and manage the interactions between these containers. This is where a service mesh is critical to managing the interactions between microservices.

Takeaways

Above and beyond everything else, it is important to have a service mesh strategy when using microservices in the cloud in order to get control over the complexities introduced by the highly distributed nature of microservices and the cloud. Beyond this key point, you should have a better understanding of the following points:

- A service mesh allows you to observe, secure, and connect microservices.
- Istio is a mature, multivendor open source service mesh implementation that has an architecture that provides you with the most complete and feature-rich service mesh implementation available.
- You have seen firsthand how Istio provides rich metrics for observability and secure mTLS communication between your services with few configurations required.

- You can add services to the mesh using Istio's automatic sidecar injection support per Kubernetes namespace, making it easier to incrementally adopt a service mesh, which is important for brownfield applications.

Next Steps

If you read through this book but you did not run the steps outlined in each chapter, we recommend that you take another pass through and actually work through the steps with the Stock Trader application. You will retain more of the lessons if you try them out yourself.

If after reading this book we have piqued your interest in either service meshes or Istio itself and you would like to get more information, we recommend that you check out the following:

- The "What's a service mesh?" blog provides another view of what a service mesh is and why it is valuable for managing microservices.
- Visit Istio.io concepts to learn more about the features that Istio provides and to better understand the details of the Istio architecture.
- Istio.io tasks provides you with examples of the Istio features that you can easily try out on our own.
- Istio blog posts provides you with additional information about changes to the Istio architecture as well as helpful information for using Istio features.
- *Istio Up and Running* is a book by our friends Zack Butcher and Lee Calcote, which will provide you with a more in-depth look into using Istio.

When you are ready, the next logical step is to apply what you have learned on your own projects to truly see the value that you can achieve with a service mesh.

About the Authors

Lin Sun is a senior technical staff member and Master Inventor at IBM. She is a maintainer on the Istio project and also serves on the Istio Steering Committee and Technical Oversight Committee. She is passionate about new technologies and loves to play with them. She holds more than 150 patents issued with USPTO.

Daniel Berg is an IBM Distinguished Engineer responsible for the technical architecture and delivery of the IBM Cloud Kubernetes Service and Istio. Daniel has deep knowledge of container technologies including Docker and Kubernetes and has extensive experience building and operating highly available cloud native services. Daniel is a member of the Technical Oversight Committee for the Istio.io open source service mesh project, and he is responsible for driving the technical integration of Istio into IBM Cloud.