O'REILLY®

Compliments of RED HAT® DEVELOPER

Introducing Istio Service Mesh for Microservices

Build and Deploy Resilient, Fault-Tolerant Cloud Native Applications

Burr Sutter & Christian Posta

TOOLS FOR A MICROSERVICES FUTURE

Learn how you can build truly scalable, adaptive, complex systems that help your business adjust to rapidly changing competitive markets.

Drive your career with expert insights, plus no-cost access to Red Hat's software library.

- Red Hat products for development purposes
- App development tutorials, cheat sheets and ebooks
- More than 100,000 technical articles

Sign up at
https://developers.redhat.com/

SECOND EDITION

Introducing Istio Service Mesh for Microservices

Build and Deploy Resilient, Fault-Tolerant Cloud Native Applications

Burr Sutter and Christian Posta

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

Introducing Istio Service Mesh for Microservices
by Burr Sutter and Christian Posta

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*.

Editors: Chris Guzikowski and Eleanor Bru
Production Editor: Deborah Baker
Copyeditor: Kim Cofer
Proofreader: Matthew Burgoyne
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

March 2019: Second Edition

Revision History for the Second Edition
2019-03-19: First Release

978-1-492-05260-9

[LSI]

Table of Contents

CHAPTER 1

Introduction

If you are looking for an introduction into the world of Istio, the service mesh platform, with detailed examples, this is the book for you. This book is for the hands-on application architect and development team lead focused on cloud native applications based on the microservices architectural style. This book assumes that you have had hands-on experience with Docker, and while Istio will be available on multiple Linux container orchestration solutions, the focus of this book is specifically targeted at Istio on Kubernetes/OpenShift. Throughout this book, we will use the terms *Kubernetes* and *OpenShift* interchangeably. (OpenShift is Red Hat's supported distribution of Kubernetes.)

If you need an introduction to Java microservices covering Spring Boot and Thorntail (formerly known as WildFly Swarm), check out *Microservices for Java Developers* (O'Reilly), by Christian Posta.

Also, if you are interested in reactive microservices, an excellent place to start is *Building Reactive Microservices in Java* (O'Reilly), by Clement Escoffier, as it is focused on Vert.x, a reactive toolkit for the Java Virtual Machine.

In addition, this book assumes that you have a comfort level with Kubernetes/OpenShift; if that is not the case, *OpenShift for Developers* (O'Reilly), by Grant Shipley and Graham Dumpleton, is an excellent ebook on that very topic. We will be deploying, interacting, and configuring Istio through the lens of OpenShift; however, the commands we'll use are mostly portable to vanilla Kubernetes as well.

To begin, we discuss the challenges that Istio can help developers solve and then describe Istio's primary components.

The Challenge of Going Faster

The software development community, in the era of digital transformation, has embarked on a relentless pursuit of better serving customers and users. Today's digital creators—application programmers—have not only evolved into faster development cycles based on Agile principles, but are also in pursuit of vastly faster deployment times. Although the monolithic code base and resulting application might be deployable at the rapid clip of once a month or even once a week, it is possible to achieve even greater "to production" velocity by breaking up the application into smaller units with smaller team sizes, each with its independent workflow, governance model, and deployment pipeline. The industry has defined this approach as *microservices architecture*.

Much has been written about the various challenges associated with microservices as it introduces many teams, for the first time, to the fallacies of distributed computing. The number one fallacy is that the "network is reliable." Microservices communicate significantly over the network—the connection between your microservices. This is a fundamental change to how most enterprise software has been crafted over the past few decades. When you add a network dependency to your application logic, you have invited in a whole host of potential hazards that grow proportionally if not exponentially with the number of connections your application depends on.

Understandably, new challenges arise in moving from a single deployment every few months to (potentially) dozens of software deployments every week or even every day.

Some of the big web companies had to develop special frameworks and libraries to help alleviate some of the challenges of an unreliable network, ephemeral cloud hosts, and many code deployments per day. For example, companies like Netflix created projects like Ribbon, Hystrix, and Eureka to solve these types of problems. Others such as Twitter and Google ended up doing similar things. These frameworks that they created were very language and platform specific and, in some cases, made it difficult to bring in new application services written in programming languages that didn't have support from these resilience frameworks. Whenever these frameworks were updated, the applications also needed to be updated to stay in lock step. Finally, even if they created an implementation of these frameworks for every possible permutation of language runtime, they'd have massive overhead in trying to apply the functionality consistently. At least in the Netflix example, these libraries were created in a time when the virtual machine (VM) was the main deployable unit and they were able to standardize on a single cloud platform plus a single application runtime, the Java Virtual Machine. Most companies cannot and will not do this.

The advent of the Linux container (e.g., Docker) and Kubernetes/OpenShift have been fundamental enablers for DevOps teams to achieve vastly higher velocities by focusing on the immutable image that flows quickly through each stage of a well-automated pipeline. How development teams manage their pipeline is now independent of the language or framework that runs inside the container. OpenShift has enabled us to provide better elasticity and overall management of a complex set of distributed, polyglot workloads. OpenShift ensures that developers can easily deploy and manage hundreds, if not thousands, of individual services. Those services are packaged as containers running in Kubernetes pods complete with their respective language runtime (e.g., Java Virtual Machine, CPython, and V8) and all their necessary dependencies, typically in the form of language-specific frameworks (e.g., Spring or Express) and libraries (e.g., jars or npms). However, OpenShift does not get involved with how each of the application components, running in their individual pods, interact with one another. This is the crossroads where architects and developers find ourselves. The tooling and infrastructure to quickly deploy and manage polyglot services is becoming mature, but we're missing similar capabilities when we talk about how those services interact. This is where the capabilities of a service mesh such as Istio allow you, the application developer, to build better software and deliver it faster than ever before.

Meet Istio

Istio is an implementation of a *service mesh*. A service mesh is the connective tissue between your services that adds additional capabilities like traffic control, service discovery, load balancing, resilience, observability, security, and so on. A service mesh allows applications to offload these capabilities from application-level libraries and allows developers to focus on differentiating business logic.

Istio has been designed from the ground up to work across deployment platforms, but it has first-class integration and support for Kubernetes.

Like many complementary open source projects within the Kubernetes ecosystem, *Istio* is a Greek nautical term that means "sail"—much like *Kubernetes* itself is the Greek term for "helmsman" or "ship's pilot". With Istio, there has been an explosion of interest in the concept of the service mesh—where Kubernetes/OpenShift has left off is where Istio begins. Istio provides developers and architects with vastly richer and declarative service discovery and routing capabilities. Where Kubernetes/OpenShift itself gives you default round-robin load balancing behind its service construct, Istio allows you to introduce unique and finely grained routing rules among all services within the mesh. Istio also provides us with greater observability, that ability to drill down deeper into the network topology of various distributed microservices, understanding the flows (tracing) between them and being able to see key metrics immediately.

If the network is in fact not always reliable, that critical link between and among our microservices needs to not only be subjected to greater scrutiny but also applied with greater rigor. Istio provides us with network-level resiliency capabilities such as retry, timeout, and implementing various circuit-breaker capabilities.

Istio also gives developers and architects the foundation to delve into a basic exploration of chaos engineering. In Chapter 5, we describe Istio's ability to drive chaos injection so that you can see how resilient and robust your overall application and its potentially dozens of interdependent microservices actually are.

Before we begin that discussion, we want to ensure that you have a basic understanding of Istio. The following section will provide you with an overview of Istio's essential components.

Understanding Istio Components

The Istio service mesh is primarily composed of two major areas: the *data plane* and the *control plane*, depicted in Figure 1-1.

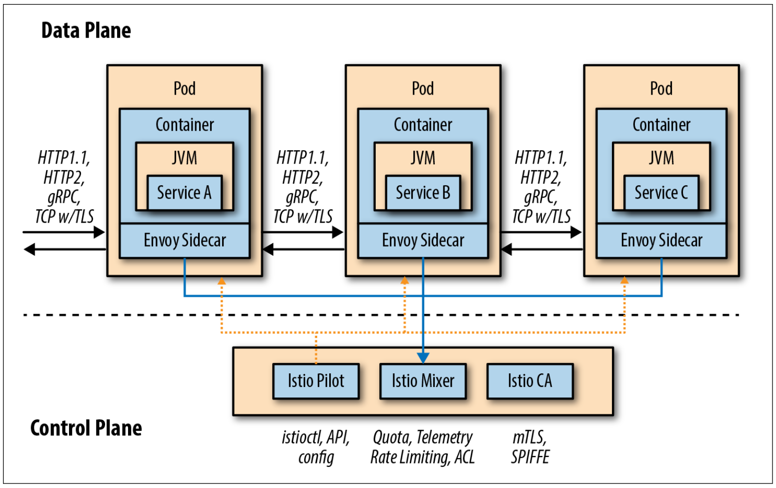


Figure 1-1. Data plane versus control plane

Data Plane

The data plane is implemented in such a way that it intercepts all inbound (ingress) and outbound (egress) network traffic. Your business logic, your app, your microservice is blissfully unaware of this fact. Your microservice can use simple framework capabilities to invoke a remote HTTP endpoint (e.g., Spring RestTemplate or JAX-RS client) across the network and mostly remain ignorant of the fact that a lot of interesting cross-cutting concerns are now being applied automatically. Figure 1-2 describes your typical microservice before the advent of Istio.

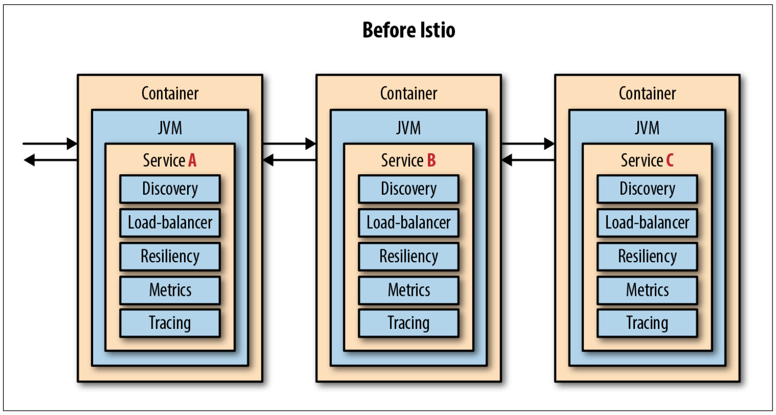


Figure 1-2. Before Istio

The data plane for Istio service mesh is made up the *istio-proxy* running as a *sidecar container*, as shown in Figure 1-3.

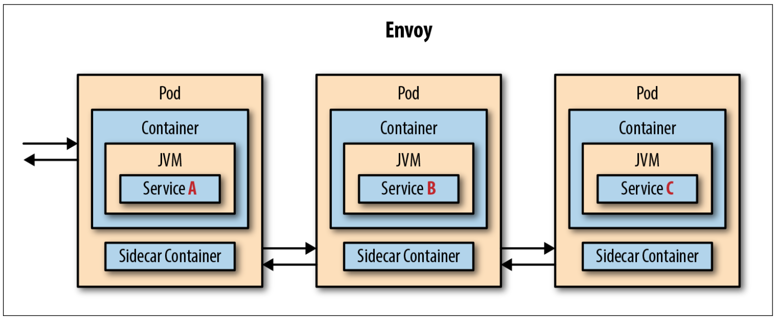


Figure 1-3. With Envoy sidecar (istio-proxy)

Let's explore each concept.

Service proxy

A service proxy augments an application service. The application service calls through the service proxy any time it needs to communicate over the network. The service proxy acts as an intermediary or interceptor that can add capabilities like automatic retries, circuit breaker, service discovery, security, and more. The default service proxy for Istio is based on Envoy proxy.

Envoy proxy is a layer 7 (L7) proxy (see the OSI model on Wikipedia) developed by Lyft, the ridesharing company, which currently uses it in production to handle millions of requests per second. Written in C++, it is battle-tested, highly performant, and lightweight. It provides features like load balancing for HTTP1.1, HTTP2, and gRPC. It has the ability to collect request-level metrics, trace spans, provide for service discovery, inject faults, and much more. You might notice that some of the capabilities of Istio overlap with Envoy. This fact is simply explained as Istio uses Envoy for its implementation of these capabilities.

But how does Istio deploy Envoy as a service proxy? Istio brings the service proxy capabilities as close as possible to the application code through a deployment technique known as the *sidecar*.

Sidecar

When Kubernetes/OpenShift were born, they did not refer to a Linux container as the runnable/deployable unit as you might expect. Instead, the name *pod* was born, and it is the primary thing to manage in a Kubernetes/OpenShift world. Why *pod*? Some think it is an obscure reference to the 1956 film *Invasion of the Body Snatchers*, but it is actually based on the concept of a family or group of whales. The whale was the early image associated with the Docker open source project—the most popular Linux container solution of its era. So, a pod can be a group of Linux containers. The sidecar is yet another Linux container that lives directly alongside your business logic application or microservice container. Unlike a real-world sidecar that bolts onto the side of a motorcycle and is essentially a simple add-on feature, this sidecar can take over the handlebars and throttle.

With Istio, a second Linux container called "istio-proxy" (aka the Envoy service proxy) is manually or automatically injected into the pod that houses your application or microservice. This sidecar is responsible for intercepting all inbound (ingress) and outbound (egress) network traffic from your business logic container, which means new policies can be applied that reroute the traffic (in or out), perhaps apply policies such as access control lists (ACLs) or rate limits, also snatch monitoring and tracing data (Mixer), and even introduce a little chaos such as network delays or HTTP errors.

Control Plane

The control plane is responsible for being the authoritative source for configuration and policy and making the data plane usable in a cluster potentially consisting of hundreds of pods scattered across a number of nodes. Istio's control plane comprises three primary Istio services: Pilot, Mixer, and Citadel.

Pilot

The Pilot is responsible for managing the overall fleet—all of your microservices' sidecars running across your Kubernetes/OpenShift cluster. The Istio Pilot ensures that each of the independent microservices, wrapped as individual Linux containers and running inside their pods, has the current view of the overall topology and an up-to-date "routing table." Pilot provides capabilities like service discovery as well as support for `VirtualService`. The `VirtualService` is what gives you fine-grained request distribution, retries, timeouts, etc. We cover this in more detail in Chapter 3 and Chapter 4.

Mixer

As the name implies, Mixer is the Istio service that brings things together. Each of the distributed istio-proxies delivers its telemetry back to Mixer. Furthermore, Mixer maintains the canonical model of the usage and access policies for the overall suite of microservices. With Mixer, you can create policies, apply rate-limiting rules, and even capture custom metrics. Mixer has a pluggable backend architecture that is rapidly evolving with new plug-ins and partners that are extending Mixer's default capabilities in many new and interesting ways. Many of the capabilities of Mixer fall beyond the scope of this introductory book, but we do address observability in Chapter 6, and security in Chapter 7.

Citadel

The Istio Citadel component, formerly known as Istio CA or Auth, is responsible for certificate signing, certificate issuance, and revocation/rotation. Istio issues X.509 certificates to all your microservices, allowing for mutual Transport Layer Security (mTLS) between those services, encrypting all their traffic transparently. It uses identity built into the underlying deployment platform and builds that into the certificates. This identity allows you to enforce policy. An example of setting up mTLS is discussed in Chapter 7.

CHAPTER 2
Installation and Getting Started

In this section, we show you how to get started with Istio on Kubernetes. Istio is not tied to Kubernetes in any way, and in fact, it's intended to be agnostic of any deployment infrastructure. With that said, Kubernetes is a great place to run Istio with its native support of the sidecar-deployment concept. Feel free to use any distribution of Kubernetes you wish, but here we use Minishift, which is a developer-focused enterprise distribution of Kubernetes named OpenShift.

Command-Line Tools Installation

As a developer, you might already have some of these tools, but for the sake of clarity, here are the tools you will need:

Minishift
: Minishift is essentially Red Hat's distribution of minikube.

VirtualBox
: Alternative virtualization options are available at virtualization options.

Docker for Mac/Windows
: You will need the Docker client (e.g., `docker build -t example/myimage`).

kubectl
: We will focus on the usage of the *oc* CLI throughout this book, but it is mostly interchangeable with *kubectl*. You can switch back and forth between the two easily.

oc
: `minishift oc-env` will output the path to the oc client binary, no need to download separately.

OpenJDK
: You will need access to both *javac* and *java* command-line tools.

Maven
: For building the sample Java projects.

stern
: For easily viewing logs.

Siege
: For load testing the Istio resiliency options in Chapter 4.

Git
: For *git clone*, downloading the sample code.

istioctl
: Will be installed via the steps that follow.

curl and tar
: To use as part of your bash shell.

Kubernetes/OpenShift Installation

Keep in mind when bootstrapping Minishift that you'll be creating a lot of services. You'll be installing the Istio control plane, some supporting metrics and visualization applications, and your sample application services. To accomplish this, the virtual machine (VM) that you use to run Kubernetes will need to have enough resources. Although we recommend 8 GB of RAM and 3 CPUs for the VM, we have seen the examples contained in this book run successfully on 4 GB of RAM and 2 CPUs.

After you've installed Minishift, you can bootstrap the environment by using this script:

```
#!/bin/bash
```

```
export MINISHIFT_HOME=~/minishift_1.27.0
export PATH=$MINISHIFT_HOME:$PATH

minishift profile set tutorial
minishift config set memory 8GB
minishift config set cpus 3
minishift config set vm-driver virtualbox
minishift config set image-caching true
minishift addon enable admin-user
minishift addon enable anyuid
minishift config set openshift-version v3.11.0

minishift start
```

When things have launched correctly, you should be able to set up your environment to have access to Minishift's included docker daemon and also log in to the Kubernetes cluster:

```
eval $(minishift oc-env)
eval $(minishift docker-env)
oc login $(minishift ip):8443 -u admin -p admin
```

If everything is successful up to this point, you should be able to run the following command:

```
oc get node
NAME        STATUS    AGE       VERSION
localhost   Ready     5m        v1.11.0+d4cacc0
```

Plus, you can view the main web console with the following:

```
minishift dashboard
```

If you have errors along the way, review the current steps of the Istio Tutorial for Java Microservices and potentially file a GitHub issue.

Istio Installation

Istio distributions come bundled with the necessary binary command-line interface (CLI) tool, installation resources, and sample applications. You should download the Istio 1.0.4 release:

```
curl -L https://github.com/istio/istio/releases/download/
                1.0.4/istio-1.0.4/-osx.tar.gz | tar xz
cd istio-1.0.4
```

Now you need to prepare your OpenShift/Kubernetes environment. Istio uses `ValidatingAdmissionWebhook` for validating Istio configuration and `MutatingAdmissionWebhook` for automatically injecting the sidecar proxy into user pods. Update Minishift's default configuration by running the following:

```
minishift openshift config set --target=kube --patch '{
    "admissionConfig": {
        "pluginConfig": {
            "ValidatingAdmissionWebhook": {
                "configuration": {
                    "apiVersion": "v1",
                    "kind": "DefaultAdmissionConfig",
                    "disable": false
                }
            },
            "MutatingAdmissionWebhook": {
                "configuration": {
                    "apiVersion": "v1",
                    "kind": "DefaultAdmissionConfig",
                    "disable": false
                }
            }
        }
    }
}'
```

Now you can install Istio. From the Istio distribution's root folder, run the following:

```
oc apply -f install/kubernetes/helm/istio/templates/crds.yaml
oc apply -f install/kubernetes/istio-demo.yaml
oc project istio-system
```

This will install all the necessary Istio control plane components including Istio Pilot, Mixer (the actual Mixer pods are called telemetry and policy), and Citadel. In addition, it installs some useful companion services: Prometheus, for metrics collection; Jaeger, for distributed tracing support; Grafana for metrics dashboard; and Servicegraph for simple visualization of services. You will be touching these services in Chapter 6.

Finally, because we're on OpenShift, you can expose these services directly through the OpenShift Router. This way you don't have to mess around with node ports:

```
oc expose svc servicegraph
oc expose svc grafana
oc expose svc prometheus
oc expose svc tracing
```

At this point, all the Istio control-plane components and companion services should be up and running. You can verify this by running the following:

```
oc get pods
NAME                               READY STATUS      RESTARTS AGE
grafana-59b787b9b                  1/1   Running     0        3m
istio-citadel-78df8c67d9           1/1   Running     0        3m
istio-cleanup-secrets              0/1   Completed   0        3m
istio-egressgateway-674686c846     1/1   Running     0        3m
istio-galley-58f566cb66            1/1   Running     0        3m
istio-grafana-post-install         0/1   Completed   0        3m
istio-ingressgateway-6bbdd58f8c    1/1   Running     0        3m
istio-pilot-56b487ff45             2/2   Running     0        3m
istio-policy-68797d879             2/2   Running     0        3m
istio-security-post-install        0/1   Completed   0        3m
istio-sidecar-injector-b88dfb954   1/1   Running     0        3m
istio-telemetry-68787476f4         2/2   Running     0        3m
istio-tracing-7596597bd7           1/1   Running     0        3m
prometheus-76db5fddd5              1/1   Running     0        3m
servicegraph-fc55fc579             1/1   Running     2        3m
```

Installing Istio Command-Line Tooling

The last thing that you need to do is make `istioctl` available on the command line. `istioctl` is the Istio command-line tool that you can use to manually inject the istio-proxy sidecar as well as create, update, and delete Istio resources. When you unzip the Istio distribution, you'll have a folder named */bin* that has the `istioctl` binary. You can add that to your path like this:

```
export ISTIO_HOME=~/istio-1.0.4
export PATH=$ISTIO_HOME/bin:$PATH
```

Now, from your command line you should be able to type `istioctl version` and see a valid response:

```
istioctl version
Version: 1.0.4
GitRevision: a44d4c8bcb427db16ca4a439adfbd8d9361b8ed3
User: root@0ead81bba27d
Hub: docker.io/istio
GolangVersion: go1.10.4
BuildStatus: Clean
```

At this point, you can move on to installing the sample services.

Example Java Microservices Installation

To effectively demonstrate the capabilities of Istio, you'll need to use a set of services that interact and communicate with one another. The services we have you work with in this section are a fictitious and simplistic re-creation of a customer portal for a website (think retail, finance, insurance, and so forth). In these scenarios, a customer portal would allow customers to set preferences for certain aspects of the website. Those preferences will have the opportunity to take recommendations from a recommendation engine that offers up suggestions. The flow of communication looks like this:

Customer ⇒ Preference ⇒ Recommendation

From this point forward, it would be best for you to have the source code that accompanies the book. You can checkout the source code from the Istio Tutorial for Java Microservices and switch to the branch book-1.0.4, as demonstrated here:

```
git clone \
https://github.com/redhat-developer-demos/istio-tutorial.git
cd istio-tutorial
git checkout book-1.0.4
```

Navigating the Code Base

If you navigate into the *istio-tutorial* subfolder that you just cloned, you should see a handful of folders. You should see the *customer*, *preference*, and *recommendation* folders. These folders each hold the source code for the respective services we'll use to demonstrate Istio capabilities.

The *customer* and *preference* services have Java Spring Boot implementations as fairly straightforward implementations of REST (representational state transfer) services. For example, here's the endpoint for the *customer* service:

```
@Value("${preferences.api.url:http://preference:8080}")
private String remoteURL;

@RequestMapping("/")
    public ResponseEntity<String> getCustomer(
      @RequestHeader("User-Agent")
        String userAgent,
      @RequestHeader(value = "user-preference",
                     required = false)
        String userPreference) {
```

```
        try {
          /* Carry user-agent as baggage */
          tracer.activeSpan()
            .setBaggageItem("user-agent", userAgent);
          if (userPreference != null &&
              !userPreference.isEmpty()) {
              tracer.activeSpan().setBaggageItem(
                "user-preference", userPreference);
          }
          ResponseEntity<String> responseEntity =
            restTemplate.getForEntity(remoteURL, String.class);
          String response = responseEntity.getBody();
          return ResponseEntity.ok(String.format(
            RESPONSE_STRING_FORMAT, response.trim()));
        } catch (HttpStatusCodeException ex) {...
        } catch (RestClientException ex) {...}
    }
```

We have left out the exception handling for a moment. You can see that this HTTP endpoint simply calls out to the *preference* service, as defined by *remoteURL* and returns the response from *preference* prepended with a fixed string. Note that there are no additional libraries that we use beyond Spring RestTemplate. We do not wrap these calls in circuit breaking, retry, client-side load balancing libraries, and so on. We're not adding any special request-tracking or request-mirroring functionality. This is a crucial point. We want you to write code that allows you to build powerful business logic without having to inject application-networking concerns into your code base and dependency trees.

In the preceding example, we've left out the exception handling for brevity, but the exception handling is also an important part of the code. Most languages provide some mechanism for detecting and handling runtime failures. When you try to call methods in your code that you know *could* fail, you should take care to handle those exceptional behaviors and deal with them appropriately. In the case of the *customer* HTTP endpoint, you are trying to make a call over the network to the *preference* service. This call could fail, and you need to wrap it with some exception handling. You could do interesting things in this exception handler like reach into a cache or call a different service. For instance, we can envision developers doing business-logic type things when they cannot get a preference like returning a list of canned preferences, and so on. This type of alternative path processing is sometimes termed *fallback* in the face of negative path behavior.

You should have also noticed the line that refers to “user-agent”:

```
tracer.activeSpan().setBaggageItem("user-agent", userAgent);
```

Istio allows you to make routing decisions based on HTTP headers, as long as you remember to pass those headers on to invoked services. In this case, we need to make sure “user-agent” is visible in both *preference* and *recommendation*. Chapter 3 focuses on traffic control with “Routing Based on Headers” on page 29.

In our customer example, we have included a special “opentracing-spring-cloud-starter” and “jaeger-traceresolver” dependency in the *pom.xml*. These dependencies enable access to interacting with the OpenTracing API inside of your Java logic.

Throughout the examples and use cases in this book, we use the Jaeger Tracing project from the Cloud Native Computing Foundation (CNCF). Learn more about Jaeger tracing at the Jaeger site, and Istio’s observability capabilities in Chapter 6.

Now that you’ve had a moment to peruse the code base, let’s build these applications and run them in containers on our Kubernetes/Openshift deployment system.

NOTE

This book focuses on the use of *oc* command-line tool instead of *kubectl*. Much like OpenShift is a superset of Kubernetes itself, so is *oc* a superset of *kubectl*. In almost all cases, you can use the two tools interchangeably; however, there are two primary areas where you will notice a distinction between the two tools. The first is related to the concept of projects, which are mapped to Kubernete’s Namespaces. You have the ability to make a project sticky via `oc project myproject"`. The second is related to security and `oc login`. OpenShift distributions, including *minishift*, are secure by default.

Before you deploy your services, make sure that you create the target namespace/project and apply the correct security permissions:

```
oc new-project tutorial
oc adm policy add-scc-to-user privileged -z default -n tutorial
```

The `oc adm` policy adds the privileged Security Context Constraints (SCCs) to the default Service Account for the tutorial namespace.

Building and Deploying the Customer Service

Now, let's build and deploy the *customer* service. Make sure you're logged in to Minishift, which you installed earlier in the section "Istio Installation" on page 11. You can verify your status by using the following command:

```
oc status
oc whoami
```

Also make sure your *docker* tool is pointing at the Minishift Docker daemon:

```
eval $(minishift docker-env)
docker images
```

You should see a long list of docker images for both Istio and OpenShift that are already part of your local docker daemon's repository.

Navigate to the *customer* directory and build the source just as you would any Maven Java project:

```
cd customer/java/springboot
mvn clean package
```

Now you have built your Java project. Next, you will package your application as a Docker image so that you can run it on Kubernetes/OpenShift:

```
docker build -t example/customer .
```

This will build your *customer* service into a docker image. You can see the results of the Docker build command by using the following:

```
docker images | grep example
```

In the *customer/kubernetes* directory, there are two Kubernetes resource files named *Deployment.yml* and *Service.yml*. Deploy the customer application/microservice and also inject the Istio sidecar proxy. Here we are using "manual injection" to add the sidecar. Try running the following command to see what the injected sidecar looks like with your deployment:

```
istioctl kube-inject -f ../../kubernetes/Deployment.yml
```

Examine the following output from the `istioctl kube-inject` command and compare it to the unchanged *Deployment.yml*. You should see the sidecar that has been injected that looks like this:

```
      - args:
        - proxy
```

```
            - sidecar
            - --configPath
            - /etc/istio/proxy
            - --binaryPath
            - /usr/local/bin/envoy
            - --serviceCluster
            - customer
            - --drainDuration
            - 45s
            - --parentShutdownDuration
            - 1m0s
            - --discoveryAddress
            - istio-pilot.istio-system:15007
            - --discoveryRefreshDelay
            - 1s
            - --zipkinAddress
            - zipkin.istio-system:9411
            - --connectTimeout
            - 10s
            - --proxyAdminPort
            - "15000"
            - --controlPlaneAuthPolicy
            - NONE
            ...
            image: docker.io/istio/proxyv2:1.0.4
            imagePullPolicy: IfNotPresent
            name: istio-proxy
```

You will see a second container injected into your deployment with configurations for finding the Istio control plane, and you should see the name of this container is `istio-proxy`. Sections of the listing have been removed for formatting purposes. Now you can create your deployment and its associated Kubernetes Service:

```
oc apply -f <(istioctl kube-inject -f \
../../kubernetes/Deployment.yml) -n tutorial

oc create -f ../../kubernetes/Service.yml -n tutorial
```

Verify that your pod came to life correctly with:

```
oc get pods
NAME                        READY     STATUS    RESTARTS   AGE
customer-6564ff969f-jqkkr   2/2       Running   0          2m
```

Now that you have the istio-proxy sidecar riding along, you will see "2/2" in the READY column. You can also use a few other techniques to get more information about your Istio'ized deployment:

```
oc get deployment customer -o yaml
oc describe pod customer-6564ff969f-jqkkr
oc logs customer-6564ff969f-jqkkr -c customer
```

Take note of the `-c customer` attribute on the `logs` command—you need to specify which container inside the pod that you want logs from.

Because customer is the forward-most microservice (*customer* > *preference* > *recommendation*), you should add an OpenShift route that exposes that endpoint, making it available outside the Kubernetes/OpenShift cluster:

```
oc expose service customer
curl customer-tutorial.$(minishift ip).nip.io
```

Here we're using the nip.io service, which is basically a wildcard DNS system that returns the IP address that you specify in the URL. This is the default behavior for Minishift.

You should see the following error because *preference* and *recommendation* are not yet deployed:

```
customer => I/O error on GET request for
"http://preference:8080": preference: Name or service not known
```

Now you can deploy the rest of the services in this example.

Building and Deploying the Preference Service

Just like you did for the *customer* service, in this section you will build, package, and deploy your *preference* service:

```
cd preference/java/springboot
mvn clean package
docker build -t example/preference:v1 .
```

You can also inject the Istio sidecar proxy into your deployment for the *preference* service as you did previously for the *customer* service:

```
oc apply -f <(istioctl kube-inject -f \
../../kubernetes/Deployment.yml) -n tutorial

oc create -f ../../kubernetes/Service.yml
```

Finally, try to `curl` your *customer* service once more:

```
curl customer-tutorial.$(minishift ip).nip.io
```

The response still contains an error, but a bit differently this time:

```
customer => 503 preference => I/O error on GET request for
"http://recommendation:8080": recommendation: Name or...
```

This time the failure is because the *preference* service cannot reach the *recommendation* service. As such, you will build and deploy the *recommendation* service next.

Building and Deploying the Recommendation Service

The last step to get our services working together is to deploy the *recommendation* service. For the sake of variety, we will use the Vert.x-based implementation of *recommendation*. Just like in the previous services, you will build, package, and deploy onto Kubernetes by using a few steps:

```
cd recommendation/java/vertx

mvn clean package

docker build -t example/recommendation:v1 .

oc apply -f <(istioctl kube-inject -f \
../../kubernetes/Deployment.yml) -n tutorial

oc create -f ../../kubernetes/Service.yml

oc get pods -w
```

Look for "2/2" under the READY column. Ctrl-C to break out of the wait, and now when you do the `curl`, you should receive a better response:

```
curl customer-tutorial.$(minishift ip).nip.io

customer => preference => recommendation v1 from '66b7c9779c': 1
```

Success! The chain of calls between the three services works as expected. If you `curl` the customer endpoint a few more times, you will see that the right-most number is incremented. This item will help you better "see" the JVM's lifecycle.

The string `66b7c9779c'` is the unique identifier for the recommendation pod. It is picked up by the Java code from the *HOSTNAME* environment variable. From your code's perspective, this is the computer that it is running on. You will want to remember that point when we describe more advanced deployment techniques and traffic routing.

Check your pod identifier using the `oc get pods` command:

```
oc get pod -l app=recommendation
NAME                           READY STATUS  RESTARTS AGE
recommendation-v1-66b7c9779c   2/2   Running 0        2m
```

Now that you have your services calling one another, we will move on to discussing some core capabilities of Istio and the power it brings to bear on solving the problems that arise *between* your microservices.

CHAPTER 3

Traffic Control

As we've seen in previous chapters, Istio consists of a control plane and a data plane. The data plane is made up of proxies that live in the application architecture. We've been looking at a proxy-deployment pattern known as the *sidecar*, which means each application *instance* has its own dedicated proxy through which all network traffic travels before it gets to the application. These sidecar proxies can be individually configured to route, filter, and augment network traffic as needed. In this chapter, we look at a handful of traffic control patterns that you can take advantage of via Istio. You might recognize these patterns as some of those practiced by big internet companies like Netflix, Amazon, or Facebook.

Smarter Canaries

The concept of the *canary deployment* has become fairly popular in the last few years. The name comes from the "canary in the coal mine" concept. Miners used to take an actual bird, a canary in a cage, into the mines to detect whether there were dangerous gases present because canaries are more susceptible to these gases than humans. The canary would not only provide nice musical songs to entertain the miners, but if at any point it collapsed off its perch, the miners knew to get out of the mine quickly.

The canary deployment has similar semantics. With a canary deployment, you deploy a new version of your code to production but allow only a subset of traffic to reach it. Perhaps only beta customers, perhaps only internal employees of your organization, perhaps only iOS users, and so on. After the canary is out there, you can monitor it for exceptions, bad behavior, changes in service-level agreement (SLA), and so forth. If the canary deployment/pod exhibits no bad behavior, you can begin to slowly increase end-user traffic to it. If it exhibits bad behavior, you can easily pull it from production. The canary deployment allows you to deploy faster but with minimal disruption should a "bad" code change make it through your automated QA tests in your continous deployment pipeline.

By default, Kubernetes offers out-of-the-box round-robin load balancing of all the pods behind a Kubernetes Service. If you want only 10% of all end-user traffic to hit your newest pod, you must have at least a 10-to-1 ratio of old pods to the new pod. With Istio, you can be much more fine-grained. You can specify that only 2% of traffic, across only three pods be routed to the latest version. Istio will also let you gradually increase overall traffic to the new version until all end users have been migrated over and the older versions of the app logic/code can be removed from the production environment.

Traffic Routing

With Istio, you can specify routing rules that control the traffic to a set of pods. Specifically, Istio uses `DestinationRule` and `VirtualService` resources to describe these rules. The following is an example of a `DestinationRule` that establishes which pods make up a specific subset:

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

And an example `VirtualService` that directs traffic using a subset and a weighting factor:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: recommendation
  namespace: tutorial
spec:
  hosts:
  - recommendation
  http:
  - route:
    - destination:
        host: recommendation
        subset: version-v1
      weight: 100
```

This `VirtualService` definition allows you to configure a percentage of traffic and direct it to a specific version of the *recommendation* service. In this case, *100%* of traffic for the *recommendation* service will always go to pods matching the label `version: v1`. The selection of pods here is very similar to the Kubernetes selector model for matching based on labels. So, any service within the service mesh that tries to communicate with the *recommendation* service will always be routed to v1 of the *recommendation* service.

The routing behavior just described is *not* just for *ingress* traffic; that is, traffic coming into the mesh. This is for all inter-service communication within the mesh. As we've illustrated in the example, these routing rules apply to services potentially deep within a service call graph. If you have a service deployed to Kubernetes that's *not* part of the service mesh, it will *not* see these rules and will adhere to the default Kubernetes load-balancing rules.

Routing to Specific Versions of a Deployment

To illustrate more complex routing, and ultimately what a canary rollout would look like, let's deploy v2 of our *recommendation* service. First, you need to make some changes to the source code for the *recommendation* service. Change the `RESPONSE_STRING_FORMAT` in the `com.redhat.developer.demos.recommendation.RecommendationVerticle` to include "v2":

```
private static final String RESPONSE_STRING_FORMAT =
  "recommendation v2 from '%s': %d\n";
```

Now do a build and package of this code as v2:

```
cd recommendation/java/vertx

mvn clean package

docker build -t example/recommendation:v2 .
```

You can quickly test your code change by executing your fat jar:

```
java -jar target/recommendation.jar
```

And then in another terminal `curl` your endpoint:

```
curl localhost:8080
recommendation v2 from 'unknown': 1
```

Finally, inject the Istio sidecar proxy and deploy this into Kubernetes:

```
oc apply -f <(istioctl kube-inject -f \
../../kubernetes/Deployment-v2.yml) -n tutorial
```

You can run `oc get pods` to see the pods as they all come up; it should look like this when all the pods are running successfully:

```
NAME                            READY   STATUS    RESTARTS   AGE
customer-3600192384-fpljb       2/2     Running   0          17m
preference-243057078-8c5hz      2/2     Running   0          15m
recommendation-v1-60483540      2/2     Running   0          12m
recommendation-v2-99634814      2/2     Running   0          15s
```

At this point, if you `curl` the *customer* endpoint, you should see traffic load balanced across both versions of the *recommendation* service. You should see something like this:

```
#!/bin/bash

while true
do curl customer-tutorial.$(minishift ip).nip.io
sleep .1
done

customer => preference => recommendation v1 from '60483540': 29
customer => preference => recommendation v2 from '99634814': 1
customer => preference => recommendation v1 from '60483540': 30
customer => preference => recommendation v2 from '99634814': 2
customer => preference => recommendation v1 from '60483540': 31
customer => preference => recommendation v2 from '99634814': 3
```

Now you can create your first `DestinationRule` and `VirtualService` to route all traffic to only v1 of the *recommendation* service. You should navigate to the root of the source code you cloned, to the main *istio-tutorial* directory, and run the following command:

```
oc -n tutorial create -f \
istiofiles/destination-rule-recommendation-v1-v2.yml

oc -n tutorial create -f \
istiofiles/virtual-service-recommendation-v1.yml
```

Now if you try to query the *customer* service, you should see all traffic routed to v1 of the service:

```
customer => preference => recommendation v1 from '60483540': 32
customer => preference => recommendation v1 from '60483540': 33
customer => preference => recommendation v1 from '60483540': 34
```

The `VirtualService` has created routes to a subset defined by the `DestinationRule` that includes only the *recommendation* pod with the label "v1".

Canary release of recommendation v2

Now that all traffic is going to v1 of your *recommendation* service, you can initiate a canary release using Istio. The canary release should take 10% of the incoming live traffic. To do this, specify a weighted routing rule in a `VirtualService` that looks like this:

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

As you can see, you're sending 90% of the traffic to v1 and 10% of the traffic to v2 with this `VirtualService`. Try replacing the previous *recommendation* `VirtualService` and see what happens when you put load on the service:

```
oc -n tutorial replace -f \
istiofiles/virtual-service-recommendation-v1_and_v2.yml
```

If you start sending load against the *customer* service like in the previous steps, you should see that only a fraction of traffic actually makes it to v2. This is a canary release. Monitor your logs, metrics, and tracing systems to see whether this new release has introduced any negative or unexpected behaviors into your system.

Continue rollout of recommendation v2

At this point, if no bad behaviors have surfaced, you should have a bit more confidence in the v2 of our *recommendation* service. You might then want to increase the traffic to v2: do this by replacing the `VirtualService` definition with one that looks like the following:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: recommendation
  namespace: tutorial
spec:
  hosts:
  - recommendation
  http:
  - route:
    - destination:
        host: recommendation
        subset: version-v1
      weight: 50
    - destination:
        host: recommendation
        subset: version-v2
      weight: 50
```

With this `VirtualService` we're going to open the traffic up to 50% to v1, and 50% to v2. When you create this rule using `oc`, you should use the `replace` command:

```
oc -n tutorial replace -f \
istiofiles/virtual-service-recommendation-v1_and_v2_50_50.yml
```

Now you should see traffic behavior change in real time, with approximately half the traffic going to v1 of the *recommendation* service and half to v2. It should look something like the following:

```
customer => ... => recommendation v1 from '60483540': 192
customer => ... => recommendation v2 from '99634814': 37
customer => ... => recommendation v2 from '99634814': 38
customer => ... => recommendation v1 from '60483540': 193
customer => ... => recommendation v2 from '99634814': 39
customer => ... => recommendation v2 from '99634814': 40
customer => ... => recommendation v2 from '99634814': 41
customer => ... => recommendation v1 from '60483540': 194
```

Finally, if everything continues to look good with this release, you can switch all the traffic to go to v2 of the *recommendation* service. You need to install the `VirtualService` that routes all traffic to v2:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: recommendation
  namespace: tutorial
spec:
  hosts:
  - recommendation
  http:
  - route:
    - destination:
        host: recommendation
        subset: version-v2
      weight: 100
```

You can replace the rule like this:

```
oc -n tutorial replace -f \
istiofiles/virtual-service-recommendation-v2.yml
```

Now you should see all traffic going to v2 of the *recommendation* service:

```
customer => preference => recommendation v2 from '99634814': 43
customer => preference => recommendation v2 from '99634814': 44
customer => preference => recommendation v2 from '99634814': 45
customer => preference => recommendation v2 from '99634814': 46
customer => preference => recommendation v2 from '99634814': 47
customer => preference => recommendation v2 from '99634814': 48
```

Restore to default behavior

To clean up this section, delete the `VirtualService` for *recommendation* and you will see the default behavior of Kubernetes 50/50 round-robin load balancing:

```
oc delete virtualservice/recommendation -n tutorial
```

Routing Based on Headers

You've seen how you can use Istio to do fine-grained routing based on service metadata. You also can use Istio to do routing based on request-level metadata. For example, you can use matching predicates to set up specific route rules based on requests that match a specified set of criteria. For example, you might want to split traffic to a particular service based on geography, mobile device, or browser. Let's see how to do that with Istio.

With Istio, you can use a match clause in the `VirtualService` to specify a predicate. For example, take a look at the following `VirtualService`:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  creationTimestamp: null
  name: recommendation
  namespace: tutorial
spec:
  hosts:
  - recommendation
  http:
  - match:
    - headers:
        baggage-user-agent:
          regex: .*Safari.*
    route:
    - destination:
        host: recommendation
        subset: version-v2
  - route:
    - destination:
        host: recommendation
        subset: version-v1
```

This rule uses a request header–based matching clause that will match only if the request includes "Safari" as part of the `user-agent` header. If the request matches the predicate, it will be routed to v2 of the *recommendation* service.

Install the rule:

```
oc -n tutorial create -f \
istiofiles/virtual-service-safari-recommendation-v2.yml
```

And let's try it out:

```
curl customer-tutorial.$(minishift ip).nip.io

customer => preference => recommendation v1 from '60483540': 465
```

If you pass in a `user-agent` header of `Safari`, you should be routed to v2:

```
curl -H 'User-Agent: Safari' \
customer-tutorial.$(minishift ip).nip.io

customer => preference => recommendation v2 from '99634814': 318
```

Also try Firefox:

```
curl -A Firefox customer-tutorial.$(minishift ip).nip.io

customer => preference => recommendation v1 from '60483540': 465
```

NOTE

If you test with real browsers, just be aware that Chrome on macOS reports itself as Safari.

Istio's `DestinationRule` and `VirtualService` objects (or Kinds) are declared as CustomResourceDefinitions; therefore you can interact with them like any built-in object type (i.e., Deployment, Pod, Service, ReplicaSet). Try the following commands to explore your Istio objects related to *recommendation*:

```
oc get crd | grep virtualservice

kubectl describe destinationrule recommendation -n tutorial

oc get virtualservice recommendation -o yaml -n tutorial
```

NOTE

oc and *kubectl* can basically be used interchangeably as *oc* is a superset of *kubectl* with some additional features related to `login`, `project`, and `new-app` that provide some shortcuts to typical Kubernetes/OpenShift operations.

Cleaning up rules

After getting to this section, you can clean up all of the Istio objects you've installed:

```
oc delete virtualservice recommendation -n tutorial
oc delete destinationrule recommendation -n tutorial
```

Dark Launch

Dark launch can mean different things to different people. In essence, a dark launch is a deployment to production that is invisible to customers. In this case, Istio allows you to duplicate or mirror traffic to a new version of your application and see how it behaves compared to the live application pod. This way you're able to put production quality requests into your new service without affecting any live traffic.

For example, you could say *recommendation* v1 takes the live traffic and *recommendation* v2 will be your new deployment. You can use Istio to mirror traffic that goes to v1 into the v2 pod. When Istio mirrors traffic, it does so in a fire-and-forget manner. In other words, Istio will do the mirroring asynchronously from the critical path of the live traffic, send the mirrored request to the test pod, and not worry about or care about a response. Let's try this out.

The first thing you should do is make sure that no `DestinationRule` or `VirtualService` is currently being used:

```
oc get destinationrules -n tutorial
No resources found.
oc get virtualservices -n tutorial
No resources found.
```

Let's take a look at a `VirtualService` that configures mirroring:

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

You can see that this directs all traffic to v1 of *recommendation*, and in the `mirror` clause, you specify which host and subset to receive the mirrored traffic.

Next, verify you're in the root directory of the source files you cloned from the Istio Tutorial and run the following commands:

```
oc -n tutorial create -f \
istiofiles/destination-rule-recommendation-v1-v2.yml

oc -n tutorial create -f \
istiofiles/virtual-service-recommendation-v1-mirror-v2.yml
```

In one terminal, tail the logs for the *recommendation* v2 service:

```
oc -n tutorial \
logs -f `oc get pods|grep recommendation-v2|awk
                                            '{ print $1 }'` \
-c recommendation
```

You can also use stern as another way to see logs of both *recommendation* v1 and v2:

```
stern recommendation
```

In another window, you can send in a request:

```
curl customer-tutorial.$(minishift ip).nip.io

customer => preference => recommendation v1 from '60483540': 466
```

You can see from the response that we're hitting v1 of the *recommendation* service as expected. If you observe in your tailing of the v2 logs, you'll also see new entries as it's processing the mirrored traffic.

Mirrored traffic can be used for powerful pre-release testing, but it does come with challenges. For example, a new version of a service might still need to communicate with a database or other collaborator services. For dealing with data in a microservices world, take a look at Edson Yanaga's book *Migrating to Microservice Databases* (O'Reilly). For a more detailed treatment on advanced mirroring techniques, see Christian's blog post, "Advanced Traffic-Shadowing Patterns for Microservices with Istio Service Mesh".

Make sure to clean up your `DestinationRule` and `VirtualService` before moving along in these samples:

```
oc delete virtualservice recommendation -n tutorial
oc delete destinationrule recommendation -n tutorial
```

Egress

By default, Istio directs all traffic originating in a service through the Istio proxy that's deployed alongside the service. This proxy evaluates its routing rules and decides how best to deliver the request. One nice thing about the Istio service mesh is that by default it blocks all outbound (outside of the cluster) traffic unless you specifically and explicitly create rules to allow traffic out. From a security standpoint, this is crucial. You can use Istio in both zero-trust networking architectures as well as traditional perimeter-based security. In both cases, Istio helps protect against a nefarious agent gaining access to a single service and calling back out to a command-and-control system, thus allowing an attacker full access to the network. By blocking any outgoing access by default and allowing routing rules to control not only internal traffic but any and all outgoing traffic, you can make your security posture more resilient to outside attacks irrespective of where they originate.

You can test this concept by shelling into one of your pods and simply running a `curl` command:

```
oc get pods -n tutorial
NAME                            READY  STATUS    RESTARTS  AGE
customer-6564ff969f-jqkkr       2/2    Running   0         19m
preference-v1-5485dc6f49-hrlxm  2/2    Running   0         19m
recommendation-v1-60483540      2/2    Running   0         20m
recommendation-v2-99634814      2/2    Running   0         7m

oc exec -it recommendation-v2-99634814 /bin/bash

[jboss@recommendation-v2-99634814 ~]$ curl -v now.httpbin.org
* About to connect() to now.httpbin.org port 80 (#0)
*   Trying 54.174.228.92...
* Connected to now.httpbin.org (54.174.228.92) port 80 (#0)
> GET / HTTP/1.1
> User-Agent: curl/7.29.0
> Host: now.httpbin.org
> Accept: */*
>
< HTTP/1.1 404 Not Found
< date: Sun, 02 Dec 2018 20:01:43 GMT
< server: envoy
< content-length: 0
<
* Connection #0 to host now.httpbin.org left intact
[jboss@recommendation-v2-99634814 ~]$ exit
```

You will receive a *404 Not Found* back from now.httpbin.org.

To address this issue you need to make egress to httpbin.org accessible with a `ServiceEntry`. Here is the one we are about to apply:

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

But first make sure to have your `DestinationRule` set up and then apply the `ServiceEntry`:

```
oc -n tutorial create -f \
istiofiles/destination-rule-recommendation-v1-v2.yml

oc -n tutorial create -f \
istiofiles/service-entry-egress-httpbin.yml -n tutorial
```

Now when you shell into your pod and run `curl` you get back a correct 200 response:

```
oc exec -it recommendation-v2-99634814 /bin/bash

[jboss@recommendation-v2-99634814 ~]$ curl now.httpbin.org
{"now": {"epoch": 1543782418.7876487...
```

You can list the egress rules like this:

```
oc get serviceentry -n tutorial
SERVICE-ENTRY NAME    HOSTS            PORTS    NAMESPACE  AGE
httpbin-egress-rule   now.httpbin.org  http/80  tutorial   5m
```

For more, visit the constantly updated tutorial that accompanies this book at Istio Tutorial for Java Microservices. There we have a Java-based example where we modify the *recommendation* service so that it attempts a call to *now.httpbin.org*, so you can see the same behavior from a programming perspective.

Finally, clean up your Istio artifacts and return to normal Kubernetes behavior by simply ensuring there are no `DestinationRule`, `VirtualService`, or `ServiceEntry` objects deployed:

```
oc delete serviceentry httpbin-egress-rule -n tutorial
oc delete virtualservice recommendation -n tutorial
oc delete destinationrule recommendation -n tutorial
```

The relatively simple examples provided in this chapter are a solid starting point for your own exploration of Istio's Traffic Management capabilities.

CHAPTER 4
Service Resiliency

Remember that your services and applications will be communicating over unreliable networks. In the past, developers have often tried to use frameworks (EJBs, CORBA, RMI, etc.) to simply make network calls appear like local method invocations. This gave developers a false peace of mind. Without ensuring the application actively guarded against network failures, the entire system was susceptible to cascading failures. Therefore, you should never assume that a remote dependency that your application or microservice is accessing across a network is guaranteed to respond with a valid payload nor within a particular timeframe (or, at all; as Douglas Adams, author of *The Hitchhiker's Guide to the Galaxy*, once said: "A common mistake that people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools"). You do not want the misbehavior of a single service to become a catastrophic failure that hamstrings your business objectives.

Istio comes with many capabilities for implementing resilience within applications, but just as we noted earlier, the actual enforcement of these capabilities happens in the sidecar. This means that the resilience features listed here are not targeted toward any specific programming language/runtime; they're applicable regardless of library or framework you choose to write your service:

Client-side load balancing
: Istio augments Kubernetes out-of-the-box load balancing.

Timeout
: Wait only *N* seconds for a response and then give up.

Retry
: If one pod returns an error (e.g., 503), retry for another pod.

Simple circuit breaker
: Instead of overwhelming the degraded service, open the circuit and reject further requests.

Pool ejection
: This provides auto removal of error-prone pods from the load-balancing pool.

Let's look at each capability with an example. Here, we use the same set of services of *customer*, *preference*, and *recommendation* as in the previous chapters.

Load Balancing

A core capability for increasing throughput and lowering latency is load balancing. A straightforward way to implement this is to have a centralized load balancer with which all clients communicate and that knows how to distribute load to any backend systems. This is a great approach, but it can become both a bottleneck as well as a single point of failure. Load-balancing capabilities can be distributed to clients with client-side load balancers. These client load balancers can use sophisticated, cluster-specific, load-balancing algorithms to increase availability, lower latency, and increase overall throughput. The Istio proxy has the capabilities to provide client-side load balancing through the following configurable algorithms:

ROUND_ROBIN
: This algorithm evenly distributes the load, in order, across the endpoints in the load-balancing pool.

RANDOM
: This evenly distributes the load across the endpoints in the load-balancing pool but without any order.

LEAST_CONN
: This algorithm picks two random hosts from the load-balancing pool and determines which host has fewer outstanding requests

(of the two) and sends to that endpoint. This is an implementation of weighted least request load balancing.

In the previous chapters on routing, you saw the use of `DestinationRules` and `VirtualServices` to control how traffic is routed to specific pods. In this chapter, we show you how to control the behavior of communicating with a particular pod using `DestinationRule` rules. To begin, we discuss how to configure load balancing with Istio `DestinationRule` rules.

First, make sure there are no `DestinationRules` that might interfere with how traffic is load balanced across v1 and v2 of our *recommendation* service. You can delete all `DestinationRules` and `VirtualServices` like this:

```
oc delete virtualservice --all -n tutorial
oc delete destinationrule --all -n tutorial
```

Next, you can scale up the *recommendation* service replicas to `3`:

```
oc scale deployment recommendation-v2 --replicas=3 -n tutorial
```

Wait a moment for all containers to become healthy and ready for traffic. Now, send traffic to your cluster using the same script you used earlier:

```
#!/bin/bash
while true
do curl customer-tutorial.$(minishift ip).nip.io
sleep .1
done
```

You should see a round-robin-style distribution of load based on the outputs:

```
customer => ... => recommendation v1 from '99634814': 1145
customer => ... => recommendation v2 from '6375428941': 1
customer => ... => recommendation v2 from '4876125439': 1
customer => ... => recommendation v2 from '2819441432': 181
customer => ... => recommendation v1 from '99634814': 1146
customer => ... => recommendation v2 from '6375428941': 2
customer => ... => recommendation v2 from '4876125439': 2
customer => ... => recommendation v2 from '2819441432': 182
customer => ... => recommendation v1 from '99634814': 1147
```

Now, change the load-balancing algorithm to `RANDOM`. Here's what the Istio `DestinationRule` would look like for that:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
```

```
metadata:
  name: recommendation
  namespace: tutorial
spec:
  host: recommendation
  trafficPolicy:
    loadBalancer:
      simple: RANDOM
```

This destination policy configures traffic to the *recommendation* service to be sent using a random load-balancing algorithm.

Let's create this `DestinationRule`:

```
oc -n tutorial create -f \
istiofiles/destination-rule-recommendation_lb_policy_app.yml
```

You should now see a more random distribution when you call your service:

```
customer => ... => recommendation v2 from '2819441432': 183
customer => ... => recommendation v2 from '6375428941': 3
customer => ... => recommendation v2 from '2819441432': 184
customer => ... => recommendation v1 from '99634814': 1153
customer => ... => recommendation v1 from '99634814': 1154
customer => ... => recommendation v2 from '2819441432': 185
customer => ... => recommendation v2 from '6375428941': 4
customer => ... => recommendation v2 from '6375428941': 5
customer => ... => recommendation v2 from '2819441432': 186
customer => ... => recommendation v2 from '4876125439': 3
```

Because you'll be creating more `DestinationRules` throughout the remainder of this chapter, now is a good time to clean up:

```
oc -n tutorial delete -f \
istiofiles/destination-rule-recommendation_lb_policy_app.yml

oc scale deployment recommendation-v2 --replicas=1 -n tutorial
```

Timeout

Timeouts are a crucial component for making systems resilient and available. Calls to services over a network can result in lots of unpredictable behavior, but the worst behavior is latency. Did the service fail? Is it just slow? Is it not even available? Unbounded latency means any of those things could have happened. But what does your service do? Just sit around and wait? Waiting is not a good solution if there is a customer on the other end of the request. Waiting also uses resources, causes other systems to potentially wait, and is usually a significant contributor to cascading failures. Your network traffic should always have timeouts in place, and you can use Istio service mesh to do this.

If you look at your *recommendation* service, find the `RecommendationVerticle.java` class and uncomment the line that introduces a delay in the service.

```
public void start() throws Exception {
  Router router = Router.router(vertx);
  router.get("/").handler(this::timeout); // adds 3 secs
  router.get("/").handler(this::logging);
  router.get("/").handler(this::getRecommendations);
  router.get("/misbehave").handler(this::misbehave);
  router.get("/behave").handler(this::behave);

  vertx.createHttpServer().requestHandler(router::accept)
    .listen(LISTEN_ON);
}
```

You should save your changes before continuing and then build the service and deploy it:

```
cd recommendation/java/vertx
mvn clean package
docker build -t example/recommendation:v2 .
oc delete pod -l app=recommendation,version=v2 -n tutorial
```

The last step here is to restart the v2 pod with the latest image of your *recommendation* service. Now, if you call your *customer* service endpoint, you should experience the delay when the call hits the *registration* v2 service:

```
time curl customer-tutorial.$(minishift ip).nip.io

customer => preference => recommendation v2 from
                                          '2819441432': 202


real    0m3.054s
user    0m0.003s
sys     0m0.003s
```

You might need to make the call a few times for it to route to the v2 service. The v1 version of *recommendation* does not have the delay as it is based on the v1 variant of the code.

Let's look at your `VirtualService` that introduces a rule that imposes a timeout when making calls to the *recommendation* service:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
```

```
metadata:
  creationTimestamp: null
  name: recommendation
  namespace: tutorial
spec:
  hosts:
  - recommendation
  http:
  - route:
    - destination:
        host: recommendation
    timeout: 1.000s
```

You create this `VirtualService` with the following command:

```
oc -n tutorial create -f \
istiofiles/virtual-service-recommendation-timeout.yml
```

Now when you send traffic to your *customer* service, you should see either a successful request (if it was routed to v1 of *recommendation*) or a *504 upstream request timeout* error if routed to v2:

```
time curl customer-tutorial.$(minishift ip).nip.io

customer => 503 preference => 504 upstream request timeout

real    0m1.151s
user    0m0.003s
sys     0m0.003s
```

You can clean up by deleting the `VirtualService`:

```
oc delete virtualservice recommendation -n tutorial
```

Retry

Because you know the network is not reliable you might experience transient, intermittent errors. This can be even more pronounced with distributed microservices rapidly deploying several times a week or even a day. The service or pod might have gone down only briefly. With Istio's retry capability, you can make a few more attempts before having to truly deal with the error, potentially falling back to default logic. Here, we show you how to configure Istio to do this.

The first thing you need to do is simulate transient network errors. In the *recommendation* service example, you find a special endpoint that simply sets a flag; this flag indicates that the return value of `getRecommendations` should always be a 503. To change the `misbe` `have` flag to true, exec into the v2 pod and invoke its `localhost:8080/misbehave` endpoint:

```
oc exec -it $(oc get pods|grep recommendation-v2 \
|awk '{ print $1 }'|head -1) -c recommendation /bin/bash

curl localhost:8080/misbehave
```

Now when you send traffic to the *customer* service, you should see some 503 errors:

```
#!/bin/bash
while true
do
curl customer-tutorial.$(minishift ip).nip.io
sleep .1
done

customer => preference => recommendation v1 from '99634814': 200
customer => 503 preference => 503 misbehavior from '2819441432'
```

Let's look at a `VirtualService` that specifies the retry configuration:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: recommendation
  namespace: tutorial
spec:
  hosts:
  - recommendation
  http:
  - route:
    - destination:
        host: recommendation
    retries:
      attempts: 3
      perTryTimeout: 2s
```

This rule sets your retry attempts to `3` and will use a `2s` timeout for each retry. The cumulative timeout is therefore 6 seconds plus the time of the original call.

Let's create your retry rule and try the traffic again:

```
oc -n tutorial create -f \
istiofiles/virtual-service-recommendation-v2_retry.yml
```

Now when you send traffic, you shouldn't see any errors. This means that even though you are experiencing 503s, Istio is automatically retrying the request, as shown here:

```
customer => preference => recommendation v1 from '99634814': 35
customer => preference => recommendation v1 from '99634814': 36
customer => preference => recommendation v1 from '99634814': 37
customer => preference => recommendation v1 from '99634814': 38
```

Now you can clean up all the rules you've installed:

```
oc delete destinationrules --all -n tutorial
oc delete virtualservices --all -n tutorial
```

And bounce/restart the recommendation-v2 pod so that the `misbe have` flag is reset to false:

```
oc delete pod -l app=recommendation,version=v2
```

Circuit Breaker

Much like the electrical safety mechanism in the modern home (we used to have fuse boxes, and "blew a fuse" is still part of our vernacular), the circuit breaker ensures that any specific appliance does not overdraw electrical current through a particular outlet. If you've ever lived with someone who plugged their radio, hair dryer, and perhaps a portable heater into the same circuit, you have likely seen this in action. The overdraw of current creates a dangerous situation because you can overheat the wire, which can result in a fire. The circuit breaker opens and disconnects the electrical current flow.

NOTE

The concepts of the circuit breaker and bulkhead for software systems were first proposed in the book by Michael Nygard titled *Release It!* (Pragmatic Bookshelf), now in its second edition.

The patterns of circuit breaker and bulkhead were popularized with the release of Netflix's Hystrix library in 2012. The Netflix libraries such as Eureka (service discovery), Ribbon (load balancing), and Hystrix (circuit breaker and bulkhead) rapidly became very popular as many folks in the industry also began to focus on microservices and cloud native architecture. Netflix OSS was built before there was a Kubernetes/OpenShift, and it does have some downsides: one, it is Java-only, and two, it requires the application developer to use the embedded library correctly. Figure 4-1 provides a timeline, from when the software industry attempted to break up monolithic application development teams and massive multimonth waterfall work-flows, to the birth of Netflix OSS and the coining of the term *microservices*.

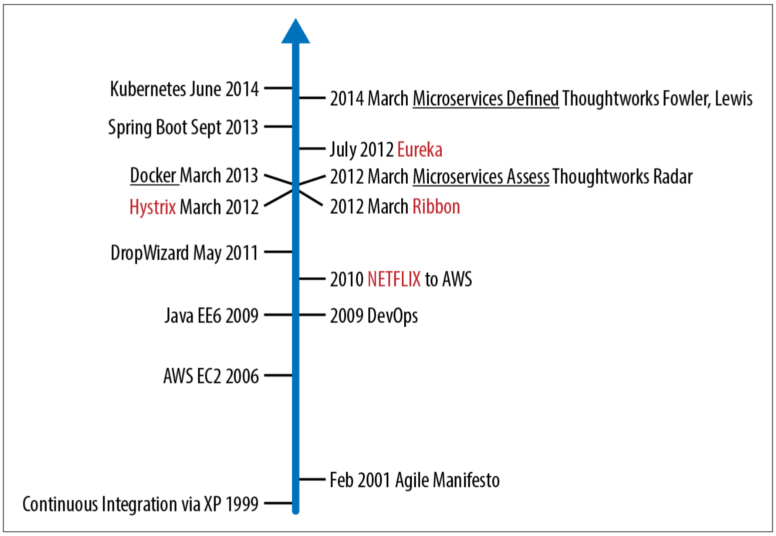


Figure 4-1. Microservices timeline

Istio puts more of the resilience implementation into the infrastructure so that you can focus more of your valuable time and energy on code that differentiates your business from the ever-growing competitive field.

Istio implements circuit breaking at the connection-pool level and at the load-balancing host level. We'll show you examples of both.

To explore the connection-pool circuit breaking, prepare by ensuring the *recommendation* v2 service has the `3s` timeout enabled (from the previous section). The *RecommendationVerticle.java* file should look similar to this:

```
Router router = Router.router(vertx);
router.get("/").handler(this::logging);
router.get("/").handler(this::timeout);  // adds 3 secs
router.get("/").handler(this::getRecommendations);
router.get("/misbehave").handler(this::misbehave);
router.get("/behave").handler(this::behave);
```

You will route traffic to both v1 and v2 of *recommendation* using this Istio `DestinationRule`:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
```

```
metadata:
  creationTimestamp: null
  name: recommendation
  namespace: tutorial
spec:
  host: recommendation
  subsets:
  - labels:
      version: v1
    name: version-v1
  - labels:
      version: v2
    name: version-v2
```

Create this v1 and v2 `DestinationRule` with the following command:

```
oc -n tutorial create -f \
istiofiles/destination-rule-recommendation-v1-v2.yml
```

And create the `VirtualService` with:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  creationTimestamp: null
  name: recommendation
  namespace: tutorial
spec:
  hosts:
  - recommendation
  http:
  - route:
    - destination:
        host: recommendation
        subset: version-v1
      weight: 50
    - destination:
        host: recommendation
        subset: version-v2
      weight: 50
```

Create the 50/50 `VirtualService` with the following command:

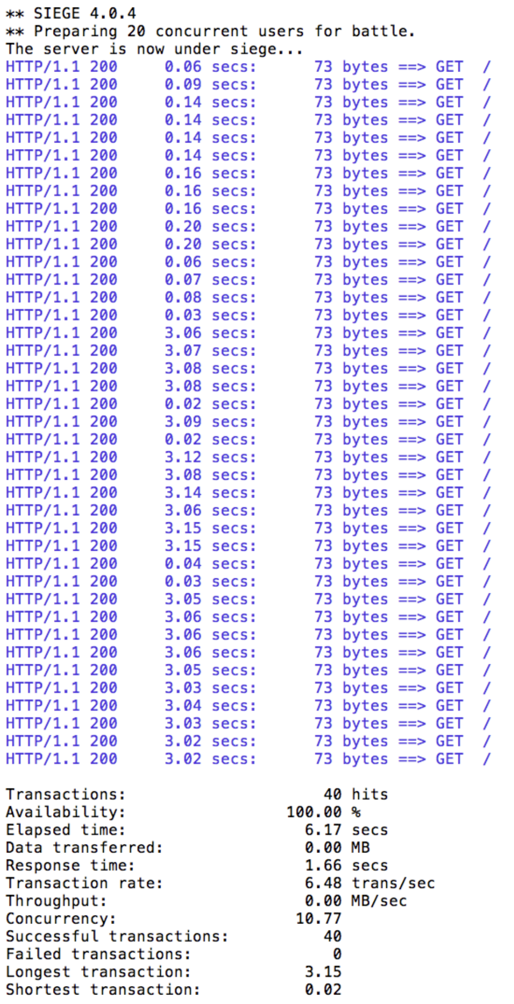
```
oc -n tutorial create -f \
istiofiles/virtual-service-recommendation-v1_and_v2_50_50.yml
```

From the initial installation instructions, we recommend you install the Siege command-line tool. You can use this for load testing with a simple command-line interface (CLI).

We will use 20 clients sending two requests each (concurrently). Use the following command to do so:

```
siege -r 2 -c 20 -v customer-tutorial.$(minishift ip).nip.io
```

You should see output similar to this:



```
** SIEGE 4.0.4
** Preparing 20 concurrent users for battle.
The server is now under siege...
HTTP/1.1 200     0.06 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.09 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.14 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.14 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.14 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.14 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.16 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.16 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.16 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.20 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.20 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.06 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.07 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.08 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.03 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.06 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.07 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.08 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.08 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.02 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.09 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.02 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.12 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.08 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.14 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.06 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.15 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.15 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.04 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.03 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.05 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.06 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.06 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.06 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.05 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.03 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.04 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.03 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.02 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.02 secs:      73 bytes ==> GET  /

Transactions:                     40 hits
Availability:                 100.00 %
Elapsed time:                   6.17 secs
Data transferred:               0.00 MB
Response time:                  1.66 secs
Transaction rate:               6.48 trans/sec
Throughput:                     0.00 MB/sec
Concurrency:                   10.77
Successful transactions:          40
Failed transactions:               0
Longest transaction:            3.15
Shortest transaction:           0.02
```

All the requests to the application were successful, but it took some time to run the test because the v2 pod was a slow performer.

Suppose that in a production system this 3-second delay was caused by too many concurrent requests to the same instance or pod. You don't want multiple requests getting queued or making that instance or pod even slower. So, we'll add a circuit breaker that will open whenever you have more than one request being handled by any instance or pod.

To create circuit breaker functionality for our services, we use an Istio `DestinationRule` that looks like this:

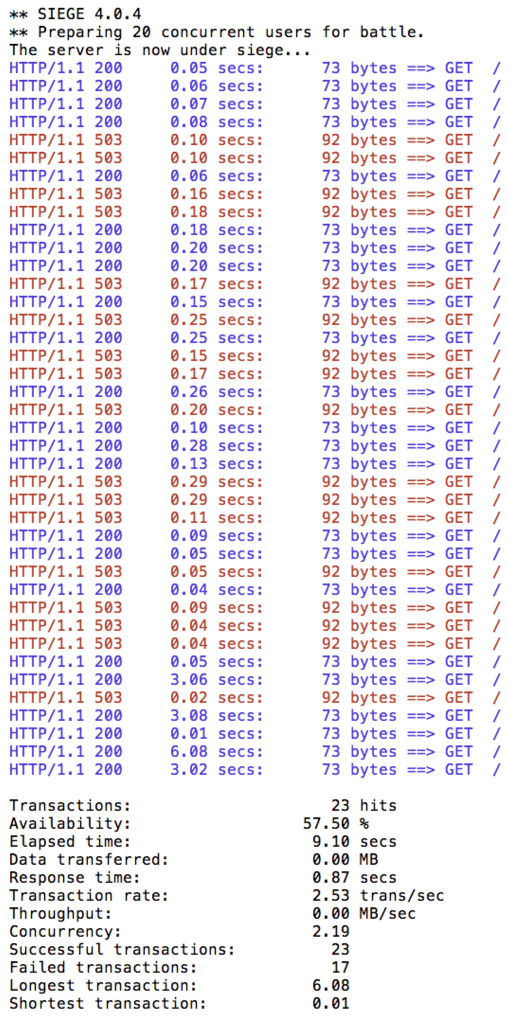
```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  creationTimestamp: null
  name: recommendation
  namespace: tutorial
spec:
  host: recommendation
  subsets:
    - name: version-v1
      labels:
        version: v1
    - name: version-v2
      labels:
        version: v2
      trafficPolicy:
        connectionPool:
          http:
            http1MaxPendingRequests: 1
            maxRequestsPerConnection: 1
          tcp:
            maxConnections: 1
        outlierDetection:
          baseEjectionTime: 120.000s
          consecutiveErrors: 1
          interval: 1.000s
          maxEjectionPercent: 100
```

Here, you're configuring the circuit breaker for any client calling into v2 of the *recommendation* service. Remember in the previous `VirtualService` that you are splitting (50%) traffic between both v1 and v2, so this `DestinationRule` should be in effect for half the traffic. You are limiting the number of connections and number of pending requests to one. (We discuss the other settings in "Pool Ejection" on page 50, in which we look at outlier detection.) Let's create this circuit breaker policy:

```
oc -n tutorial replace -f \
istiofiles/destination-rule-recommendation_cb_policy_
                                           version_v2.yml
```

Now try the `siege` load generator one more time:

```
siege -r 2 -c 20 -v customer-tutorial.$(minishift ip).nip.io
```



```
** SIEGE 4.0.4
** Preparing 20 concurrent users for battle.
The server is now under siege...
HTTP/1.1 200     0.05 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.06 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.07 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.08 secs:      73 bytes ==> GET  /
HTTP/1.1 503     0.10 secs:      92 bytes ==> GET  /
HTTP/1.1 503     0.10 secs:      92 bytes ==> GET  /
HTTP/1.1 200     0.06 secs:      73 bytes ==> GET  /
HTTP/1.1 503     0.16 secs:      92 bytes ==> GET  /
HTTP/1.1 503     0.18 secs:      92 bytes ==> GET  /
HTTP/1.1 200     0.18 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.20 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.20 secs:      73 bytes ==> GET  /
HTTP/1.1 503     0.17 secs:      92 bytes ==> GET  /
HTTP/1.1 200     0.15 secs:      73 bytes ==> GET  /
HTTP/1.1 503     0.25 secs:      92 bytes ==> GET  /
HTTP/1.1 200     0.25 secs:      73 bytes ==> GET  /
HTTP/1.1 503     0.15 secs:      92 bytes ==> GET  /
HTTP/1.1 503     0.17 secs:      92 bytes ==> GET  /
HTTP/1.1 200     0.26 secs:      73 bytes ==> GET  /
HTTP/1.1 503     0.20 secs:      92 bytes ==> GET  /
HTTP/1.1 200     0.10 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.28 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.13 secs:      73 bytes ==> GET  /
HTTP/1.1 503     0.29 secs:      92 bytes ==> GET  /
HTTP/1.1 503     0.29 secs:      92 bytes ==> GET  /
HTTP/1.1 503     0.11 secs:      92 bytes ==> GET  /
HTTP/1.1 200     0.09 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.05 secs:      73 bytes ==> GET  /
HTTP/1.1 503     0.05 secs:      92 bytes ==> GET  /
HTTP/1.1 200     0.04 secs:      73 bytes ==> GET  /
HTTP/1.1 503     0.09 secs:      92 bytes ==> GET  /
HTTP/1.1 503     0.04 secs:      92 bytes ==> GET  /
HTTP/1.1 503     0.04 secs:      92 bytes ==> GET  /
HTTP/1.1 200     0.05 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.06 secs:      73 bytes ==> GET  /
HTTP/1.1 503     0.02 secs:      92 bytes ==> GET  /
HTTP/1.1 200     3.08 secs:      73 bytes ==> GET  /
HTTP/1.1 200     0.01 secs:      73 bytes ==> GET  /
HTTP/1.1 200     6.08 secs:      73 bytes ==> GET  /
HTTP/1.1 200     3.02 secs:      73 bytes ==> GET  /

Transactions:                     23 hits
Availability:                  57.50 %
Elapsed time:                   9.10 secs
Data transferred:               0.00 MB
Response time:                  0.87 secs
Transaction rate:               2.53 trans/sec
Throughput:                     0.00 MB/sec
Concurrency:                    2.19
Successful transactions:          23
Failed transactions:              17
Longest transaction:            6.08
Shortest transaction:           0.01
```

You can now see that almost all calls completed in less than a second with either a success or a failure. You can try this a few times to see that this behavior is consistent. The circuit breaker will short circuit any pending requests or connections that exceed the specified threshold (in this case, an artificially low number, 1, to demonstrate these capabilities). The goal of the circuit breaker is to fail fast.

You can clean up these Istio rules with a simple “oc delete”:

```
oc delete virtualservice recommendation -n tutorial
oc delete destinationrule recommendation -n tutorial
```

Pool Ejection

The last of the resilience capabilities that we discuss has to do with identifying badly behaving cluster hosts and not sending any more traffic to them for a cool-off period (essentially kicking the bad-behaving pod out of the load-balancing pool). Because the Istio proxy is based on Envoy and Envoy calls this implementation *outlier detection*, we’ll use the same terminology for discussing Istio.

In a scenario where your software development teams are deploying their components into production, perhaps multiple times per week, during the middle of the workday, being able to kick out misbehaving pods adds to overall resiliency. Pool ejection or outlier detection is a resilience strategy that is valuable whenever you have a group of pods (multiple replicas) to serve a client request? If the request is forwarded to a certain instance and it fails (e.g., returns a 50x error code), Istio will eject this instance from the pool for a certain sleep window. In our example, the sleep window is configured to be `15s`. This increases the overall availability by making sure that only healthy pods participate in the pool of instances.

First, you need to ensure that you have a `DestinationRule` and `VirtualService` in place. Let’s use a 50/50 split of traffic:

```
oc -n tutorial create -f \
istiofiles/destination-rule-recommendation-v1-v2.yml

oc -n tutorial create -f \
istiofiles/virtual-service-recommendation-v1_and_v2_50_50.yml
```

Next, you can scale the number of pods for the v2 deployment of *recommendation* so that you have multiple instances in the load-balancing pool with which to work:

```
oc scale deployment recommendation-v2 --replicas=2 -n tutorial
```

Wait a moment for all of the pods to get to the ready state then generate some traffic against the *customer* service:

```
#!/bin/bash
while true
do curl customer-tutorial.$(minishift ip).nip.io
```

```
sleep .1
done
```

You will see the load balancing 50/50 between the two different versions of the *recommendation* service. And within version v2, you will also see that some requests are handled by one pod and some requests are handled by the other pod:

```
customer => ... => recommendation v1 from '99634814': 448
customer => ... => recommendation v2 from '3416541697': 27
customer => ... => recommendation v1 from '99634814': 449
customer => ... => recommendation v1 from '99634814': 450
customer => ... => recommendation v2 from '2819441432': 215
customer => ... => recommendation v1 from '99634814': 451
customer => ... => recommendation v2 from '3416541697': 28
customer => ... => recommendation v2 from '3416541697': 29
customer => ... => recommendation v2 from '2819441432': 216
```

To test outlier detection, you'll want one of the pods to misbehave. Find one of them and exec to it and instruct it to misbehave:

```
oc get pods -l app=recommendation,version=v2
```

You should see something like this:

```
recommendation-v2-2819441432       2/2      Running   0       1h
recommendation-v2-3416541697       2/2      Running   0       7m
```

Now you can get into one of the pods and add some erratic behavior to it. Get one of the pod names from your system and run the following command:

```
oc -n tutorial exec -it \
recommendation-v2-3416541697 -c recommendation /bin/bash
```

You will be inside the application container of your pod `recommendation-v2-3416541697`. Now execute a simple curl command and then exit:

```
curl localhost:8080/misbehave
exit
```

This is a special endpoint that will make our application return only 503s:

```
#!/bin/bash
while true
do curl customer-tutorial.$(minishift ip).nip.io
sleep .1
done
```

You'll see that whenever the pod `recommendation-v2-3416541697` receives a request, you get a 503 error:

```
customer => ... => recommendation v1 from '2039379827': 495
customer => ... => recommendation v2 from '2036617847': 248
customer => ... => recommendation v1 from '2039379827': 496
customer => ... => recommendation v1 from '2039379827': 497
customer => 503 preference => 503 misbehavior from '3416541697'
customer => ... => recommendation v2 from '2036617847': 249
customer => ... => recommendation v1 from '2039379827': 498
customer => 503 preference => 503 misbehavior from '3416541697'
```

Now let's see what happens when you configure Istio to eject misbehaving hosts. Look at the `DestinationRule` in the following:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  creationTimestamp: null
  name: recommendation
  namespace: tutorial
spec:
  host: recommendation
  subsets:
  - labels:
      version: v1
    name: version-v1
    trafficPolicy:
      connectionPool:
        http: {}
        tcp: {}
      loadBalancer:
        simple: RANDOM
      outlierDetection:
        baseEjectionTime: 15.000s
        consecutiveErrors: 1
        interval: 5.000s
        maxEjectionPercent: 100
  - labels:
      version: v2
    name: version-v2
    trafficPolicy:
      connectionPool:
        http: {}
        tcp: {}
      loadBalancer:
        simple: RANDOM
      outlierDetection:
        baseEjectionTime: 15.000s
        consecutiveErrors: 1
```

```
          interval: 5.000s
          maxEjectionPercent: 100
```

In this DestinationRule, you're configuring Istio to check every 5 seconds for misbehaving pods and to remove those pods from the load-balancing pool after one consecutive error (artificially low for this example) and keep it out for 15 seconds:

```
oc -n tutorial replace -f
istiofiles/destination-rule-recommendation_cb_policy_pool
                                              _ejection.yml
```

Let's put some load on the service now and see its behavior:

```
#!/bin/bash
while true
do curl customer-tutorial.$(minishift ip).nip.io
sleep .1
Done
```

You will see that whenever you get a failing request with 503 from the pod recommendation-v2-3416541697, it is ejected from the pool and doesn't receive more requests until the sleep window expires, which takes at least 15 seconds:

```
customer => ... => recommendation v1 from '2039379827': 509
customer => 503 preference => 503 misbehavior from '3416541697'
customer => ... => recommendation v1 from '2039379827': 510
customer => ... => recommendation v1 from '2039379827': 511
customer => ... => recommendation v1 from '2039379827': 512
customer => ... => recommendation v2 from '2036617847': 256
customer => ... => recommendation v2 from '2036617847': 257
customer => ... => recommendation v1 from '2039379827': 513
customer => ... => recommendation v2 from '2036617847': 258
customer => ... => recommendation v2 from '2036617847': 259
customer => ... => recommendation v2 from '2036617847': 260
customer => ... => recommendation v1 from '2039379827': 514
customer => ... => recommendation v1 from '2039379827': 515
customer => 503 preference => 503 misbehavior from '3416541697'
customer => ... => recommendation v1 from '2039379827': 516
customer => ... => recommendation v2 from '2036617847': 261
```

Combination: Circuit Breaker + Pool Ejection + Retry

Even with pool ejection your application still does not look that resilient, probably because you are still letting some errors be propagated to your clients. But you can improve this. If you have enough instances or versions of a specific service running in your system, you can combine multiple Istio capabilities to achieve the ultimate backend resilience:

- Circuit breaker, to avoid multiple concurrent requests to an instance
- Pool ejection, to remove failing instances from the pool of responding instances
- Retries, to forward the request to another instance just in case you get an open circuit breaker or pool ejection

By simply adding a retry configuration to our current `VirtualService`, we are able to completely get rid of our 503 responses. This means that whenever you receive a failed request from an ejected instance, Istio will forward the request to another healthy instance:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  creationTimestamp: null
  name: recommendation
  namespace: tutorial
spec:
  hosts:
  - recommendation
  http:
  - retries:
      attempts: 3
      perTryTimeout: 4.000s
    route:
    - destination:
        host: recommendation
        subset: version-v1
      weight: 50
    - destination:
        host: recommendation
        subset: version-v2
      weight: 50
```

And replace the `VirtualService`:

```
oc -n tutorial replace -f \
istiofiles/virtual-service-recommendation-v1_and_v2_retry.yml
```

Throw some requests at the *customer* endpoint:

```
#!/bin/bash
while true
do curl customer-tutorial.$(minishift ip).nip.io
```

```
sleep .1
done
```

You will no longer receive 503s:

```
customer => ... => recommendation v1 from '2039379827': 538
customer => ... => recommendation v1 from '2039379827': 539
customer => ... => recommendation v1 from '2039379827': 540
customer => ... => recommendation v2 from '2036617847': 281
customer => ... => recommendation v1 from '2039379827': 541
customer => ... => recommendation v2 from '2036617847': 282
customer => ... => recommendation v1 from '2039379827': 542
customer => ... => recommendation v1 from '2039379827': 543
customer => ... => recommendation v2 from '2036617847': 283
customer => ... => recommendation v2 from '2036617847': 284
```

Your misbehaving pod `recommendation-v2-3416541697` never shows up, thanks to pool ejection and retry.

Clean up by removing the timeout in `RecommendationVerticle.java`, rebuilding the docker image, deleting the misbehaving pod, and then removing the `DestinationRule` and `VirtualService` objects:

```
cd recommendation/java/vertx
mvn clean package
docker build -t example/recommendation:v2 .

oc scale deployment recommendation-v2 --replicas=1 -n tutorial
oc delete pod -l app=recommendation,version=v2 -n tutorial

oc delete virtualservice recommendation -n tutorial
oc delete destinationrule recommendation -n tutorial
```

Now that you have seen how to make your service to service calls more resilent and robust, it is time to purposely break things by introducing some chaos in Chapter 5.

CHAPTER 5

Chaos Testing

The unveiling of a relatively famous OSS project by the team at Netflix called “Chaos Monkey” had a disruptive effect on the IT world. The concept that Netflix had built code that randomly kills various services in their production environment blew people’s minds. When many teams struggle just maintaining their uptime requirements, promoting self-sabotage and attacking oneself seemed absolutely crazy. Yet from the moment Chaos Monkey was born, a new movement arose: *chaos engineering*.

According to the Principles of Chaos Engineering website, “Chaos Engineering is the discipline of experimenting on a distributed system in order to build confidence in the system’s capability to withstand turbulent conditions in production.”

In complex systems (software systems or ecological systems), things can and will fail, but the ultimate goal is to stop catastrophic failure of the overall system. So how do you verify that your overall system —your network of microservices—is in fact resilient? You inject a little chaos. With Istio, this is a relatively simple matter because the istio-proxy is intercepting all network traffic; therefore, it can alter the responses including the time it takes to respond. Two interesting faults that Istio makes easy to inject are *HTTP error codes* and *network delays*.

HTTP Errors

Based on exercises earlier in this book, make sure that *recommendation* v1 and v2 are both deployed with no code-driven misbehavior or long waits/latency. Now, you will be injecting errors via Istio instead of using Java code:

```
oc get pods -l app=recommendation -n tutorial
NAME                              READY     STATUS    RESTARTS   AGE
recommendation-v1-3719512284      2/2       Running   6          18m
recommendation-v2-2815683430      2/2       Running   0          13m
```

Also, double-check that you do not have any `DestinationRule`s or `VirtualService`s:

```
oc delete virtualservices --all -n tutorial
oc delete destinationrules --all -n tutorial
```

We use the combination of Istio's `DestinationRule` and `VirtualService` to inject a percentage of faults—in this case, returning the HTTP 503 error code 50% of the time.

The `DestinationRule`:

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

The `VirtualService`:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: recommendation
  namespace: tutorial
spec:
  hosts:
  - recommendation
  http:
  - fault:
      abort:
        httpStatus: 503
        percent: 50
```

```
    route:
    - destination:
        host: recommendation
        subset: app-recommendation
```

And you apply the `DestinationRule` and `VirtualService`:

```
oc -n tutorial create -f \
istiofiles/destination-rule-recommendation.yml

oc -n tutorial create -f \
istiofiles/virtual-service-recommendation-503.yml
```

Testing the change is as simple as issuing a few curl commands at the customer endpoint. Make sure to test it a few times, looking for the resulting 503 approximately 50% of the time:

```
curl customer-tutorial.$(minishift ip).nip.io
customer => preference => recommendation v1 from
                                        '3719512284': 88

curl customer-tutorial.$(minishift ip).nip.io
customer => 503 preference => 503 fault filter abort
```

You can now see if the *preference* service is properly handling the exceptions being returned by the *recommendation* service.

Clean up by removing the `VirtualService` but leave the `DestinationRule` in place:

```
oc delete virtualservices --all -n tutorial
```

Delays

The most insidious of possible distributed computing faults is not a "dead" service but a service that is responding slowly, potentially causing a cascading failure in your network of services. More importantly, if your service has a specific service-level agreement (SLA) it must meet, how do you verify that slowness in your dependencies doesn't cause you to fail in delivering to your awaiting customer?

Much like the HTTP error injection, network delays use the `VirtualService` kind, as well. The following manifest injects 7 seconds of delay into 50% of the responses from the *recommendation* service:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  creationTimestamp: null
  name: recommendation
  namespace: tutorial
spec:
  hosts:
  - recommendation
  http:
  - fault:
      delay:
        fixedDelay: 7.000s
        percent: 50
    route:
    - destination:
        host: recommendation
        subset: app-recommendation
```

Apply the new `VirtualService`:

```
oc -n tutorial create -f \
istiofiles/virtual-service-recommendation-delay.yml
```

Then, send a few requests at the customer endpoint and notice the "time" command at the front. This command will output the elapsed time for each response to the `curl` command, allowing you to see that 7-second delay:

```
#!/bin/bash
while true
do
time curl customer-tutorial.$(minishift ip).nip.io
sleep .1
done
```

Many requests to the customer end point now have a delay. If you are monitoring the logs for *recommendation* v1 and v2, you will also see the delay happens *before* the *recommendation* service is actually called. The delay is in the Istio proxy (Envoy), not in the actual endpoint:

```
oc logs recommendation-v2-2815683430 -f -c recommendation
```

In Chapter 4 you saw how to deal with errors bubbling up from your code and now in this chapter, you played the role of self-saboteur by injecting errors/delays via Istio's `VirtualService`. At this time there should be a key question in your mind: "How do I know that these errors are happening within my application?" The answer is in Chapter 6.

Clean up:

```
oc delete virtualservice recommendation -n tutorial
oc delete destinationrule recommendation -n tutorial
```

CHAPTER 6

Observability

One of the greatest challenges with the management of a microservices architecture is simply trying to understand the relationships between individual components of the overall system. A single end-user transaction might flow through several, perhaps a dozen or more, independently deployed microservices running in their independent pods, and discovering where performance bottlenecks or errors have occurred provides valuable information.

In this chapter, we will touch on tracing via Jaeger, metrics via Grafana and Prometheus, plus service graphing via Kiali.

Istio's Mixer capability is implemented as two different pods and services within the *istio-system* namespace. The following queries make them easier to spot as they have a common label:

```
oc get pods -l istio=mixer -n istio-system
oc get services -l istio=mixer -n istio-system
```

istio-policy and *istio-telemetry* are the services that make up Istio's Mixer functionality.

Tracing

Often the first thing to understand about your microservices architecture is specifically which microservices are involved in an end-user transaction. If many teams are deploying their dozens of microservices, all independently of one another, it is often difficult to understand the dependencies across that "mesh" of services. Istio's Mixer comes "out of the box" with the ability to pull tracing spans from your distributed microservices. This means that tracing is programming-language agnostic so that you can use this capability in a polyglot world where different teams, each with its own microservice, can be using different programming languages and frameworks.

Although Istio supports both Zipkin and Jaeger, for our purposes we focus on Jaeger, which implements OpenTracing, a vendor-neutral tracing API. Jaeger was originally open sourced by the Uber Technologies team and is a distributed tracing system specifically focused on microservices architecture.

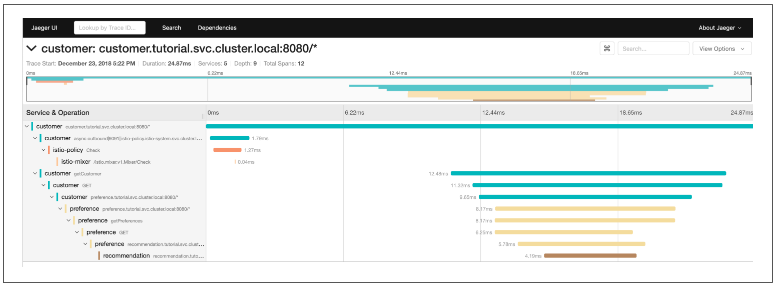
An important term to understand here is *span*, which Jaeger defines as "a logical unit of work in the system that has an operation name, the start time of the operation, and the duration. Spans can be nested and ordered to model causal relationships. An RPC call is an example of a span."

Another important term is *trace*, which Jaeger defines as "a data/execution path through the system, and can be thought of as a directed acyclic graph of spans."

Open the Jaeger console by using the following command:

```
minishift openshift service tracing --in-browser
```

You can then select Customer from the drop-down list box and explore the traces found, as illustrated in Figure 6-1.



customer: customer.tutorial.svc.cluster.local:8080/*

Trace Start: **December 23, 2018 5:22 PM** | Duration: **24.87ms** | Services: **5** | Depth: **9** | Total Spans: **12**

Service & Operation	Duration
customer customer.tutorial.svc.cluster.local:8080/*	
customer async outbound|9091||istio-policy.istio-system.svc.cluster.l…	1.79ms
istio-policy Check	1.27ms
istio-mixer /istio.mixer.v1.Mixer/Check	0.04ms
customer getCustomer	12.48ms
customer GET	11.32ms
customer preference.tutorial.svc.cluster.local:8080/*	9.65ms
preference preference.tutorial.svc.cluster.local:8080/*	8.17ms
preference getPreferences	8.17ms
preference GET	6.25ms
preference recommendation.tutorial.svc.clust…	5.78ms
recommendation recommendation.tuto…	4.19ms

Figure 6-1. Jaeger's view of the customer-preference-recommendation trace

It's important to remember that your programming logic must forward the OpenTracing headers from your inbound call to every outbound call:

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

However, your chosen framework may have support for automatically carrying those headers. In the case of the *customer* and *preference* services, for the Spring Boot implementations, there is opentracing_spring_cloud.

Our *customer* and *preference* services are using the *TracerResolver* library, so that the concrete tracer can be loaded automatically without our code having a hard dependency on Jaeger. Given that the Jaeger tracer can be configured via environment variables, we don't need to do anything in order to get a properly configured Jaeger tracer ready and registered with OpenTracing. That said, there are cases where it's appropriate to manually configure a tracer. Refer to the Jaeger documentation for more information on how to do that.

By default, Istio captures or samples 100% of the requests flowing through the mesh. This is valuable for a development scenario where you are attempting to debug aspects of the application but it might be too voluminous in a different setting such as performance benchmark or production environment. The sampling rate is defined by the "PILOT_TRACE_SAMPLING" environment variable on the Istio Pilot Deployment. This can can be viewed/edited via the following command:

```
oc edit deployment istio-pilot -n istio-system
```

Metrics

By default, Istio will gather telemetry data across the service mesh by leveraging Prometheus and Grafana to get started with this important capability. You can get the URL to the Grafana console using the minishift `service` command:

```
minishift openshift service grafana --url
```

Make sure to select Istio Workload Dashboard in the upper left of the Grafana dashboard, as demonstrated in Figure 6-2.

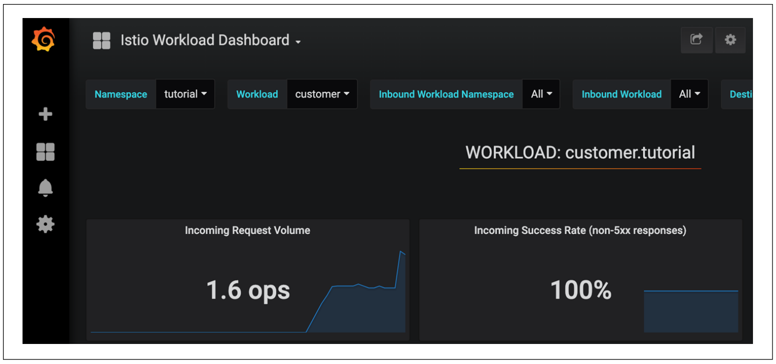


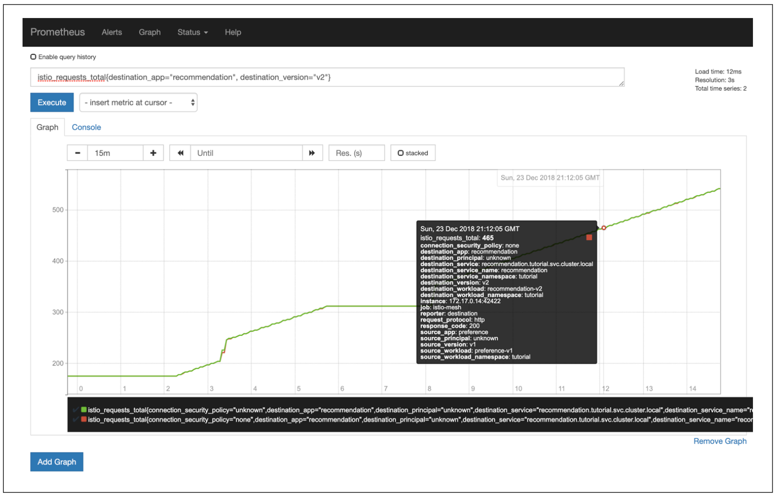
Figure 6-2. The Grafana dashboard—selecting Istio Workload Dashboard

You can also visit the Prometheus dashboard directly with the following command:

```
minishift openshift service prometheus --in-browser
```

The Prometheus dashboard allows you to query for specific metrics and graph them. For instance, you can review the total request count for the *recommendation* service, specifically the "v2" version as seen in Figure 6-3:

```
istio_requests_total{destination_app="recommendation",\
 destination_version="v2"}
```



```
istio_requests_total{destination_app="recommendation", destination_version="v2"}
```

```
Sun, 23 Dec 2018 21:12:05 GMT
istio_requests_total: 465
connection_security_policy: none
destination_app: recommendation
destination_principal: unknown
destination_service: recommendation.tutorial.svc.cluster.local
destination_service_name: recommendation
destination_service_namespace: tutorial
destination_version: v2
destination_workload: recommendation-v2
destination_workload_namespace: tutorial
instance: 172.17.0.14:42422
job: istio-mesh
reporter: destination
request_protocol: http
response_code: 200
source_app: preference
source_principal: unknown
source_version: v1
source_workload: preference-v1
source_workload_namespace: tutorial
```

```
🟩 istio_requests_total{connection_security_policy="unknown",destination_app="recommendation",destination_principal="unknown",destination_service="recommendation.tutorial.svc.cluster.local",destination_service_name="r
🟥 istio_requests_total{connection_security_policy="none",destination_app="recommendation",destination_principal="unknown",destination_service="recommendation.tutorial.svc.cluster.local",destination_service_name="recor
```

Figure 6-3. The Prometheus dashboard

You can also review other interesting datapoints such as the pod memory usage with the following query string:

```
container_memory_rss{container_name="customer"}
```

Prometheus is a very powerful tool for gathering and extracting metric data from your Kubernetes/OpenShift cluster. Prometheus is currently a top-level or graduated project within the Cloud Native Computing Foundation alongside Kubernetes itself. For more information on query syntax and alerting, please review the documentation at the Prometheus website.

Service Graph

Istio has provided the out-of-the-box basic Servicegraph visualization since its earliest days. Now, a new, more comprehensive service graph tool and overall health monitoring solution called Kiali has been created by the Red Hat team, as depicted in Figure 6-4. The Kiali project provides answers to interesting questions like: What microservices are part of my Istio service mesh and how are they connected?

At the time of this writing, Kiali must be installed separately and those installation steps are somewhat complicated. Kiali wants to know the URLs for both Jaeger and Grafana and that requires some interesting environment variable substitution. The *envsubst* tool comes from a package called *gettext* and is available for Fedora via:

```
dnf install gettext
```

Or macOS:

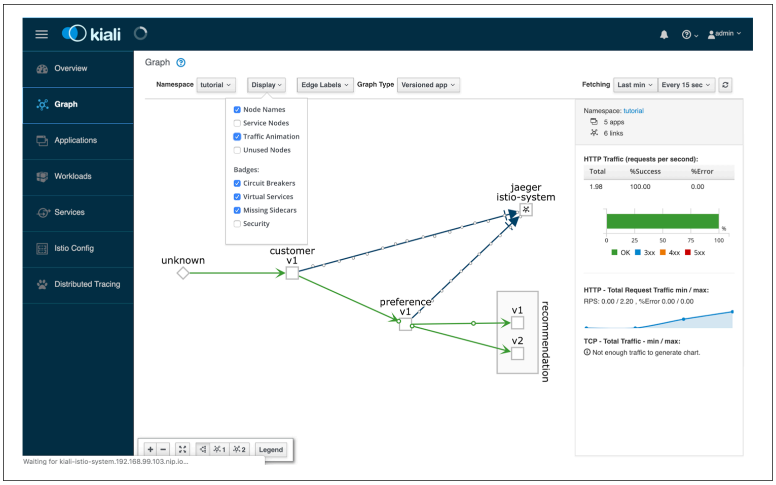
```
brew install gettext
```

And the Kiali installation steps:

```
# URLS for Jaeger and Grafana
export JAEGER_URL="https://tracing-istio-system.$(minishift
                                                   ip).nip.io"
export GRAFANA_URL="https://grafana-istio-system.$(minishift
                                                    ip).nip.io"
export IMAGE_VERSION="v0.10.0"

curl -L http://git.io/getLatestKiali | bash
```

If you run into trouble with the installation process, make sure to visit the Kiali user forum. Like Istio itself Kiali is a fast-moving project that continues to change rapidly.



Namespace tutorial | Graph Type Versioned app | Fetching Last min | Every 15 sec

☑ Node Names
☐ Service Nodes
☑ Traffic Animation
☐ Unused Nodes

Badges:
☑ Circuit Breakers
☑ Virtual Services
☑ Missing Sidecars
☐ Security

Namespace: tutorial
5 apps
6 links

HTTP Traffic (requests per second):

Total	%Success	%Error
1.98	100.00	0.00

🟩 OK 🟦 3xx 🟧 4xx 🟥 5xx

HTTP - Total Request Traffic min / max:
RPS: 0.00 / 2.20 , %Error 0.00 / 0.00

TCP - Total Traffic - min / max:
Not enough traffic to generate chart.

Waiting for kiali-istio-system.192.168.99.103.nip.io...

Figure 6-4. The Kiali dashboard

As you can see with the various out-of-the-box as well as third-party additions, Istio makes your overall application's components—its mesh of microservices—much more visible and more observable. In previous chapters you introduced errors as well as network delays, and now with these additional tools, you can better track where the potential problems are.

CHAPTER 7

Security

More modern cloud native application architecture may have a number of independent development teams, executing at independent sprint intervals, deploying new capabilities at a weekly or daily pace, and responsible for their own “App Ops”—their production readiness. Istio’s mission is to enable cross-cutting concerns across a series of microservices that make up the overall application, ensuring some level of consistency across all these independent services. One key capability of Istio is its ability to apply security constraints across the application with zero impact to the actual programming logic of each microservice. With the sidecar istio-proxy in place, you are applying these constraints at the network level between the services that comprise the application.

Even the super-simple application explored in this book, where the *customer* service/microservice calls *preference* which calls *recommendation*, exposes a number of possible areas where service mesh level security constraints can be applied.

In this chapter, we will explore Istio’s mTLS, Mixer Policy, and RBAC capabilities.

mutual Transport Layer Security (mTLS)

mTLS provides encryption between sidecar-injected, istio-enabled services. By default, traffic among our three services of *customer*, *preference*, and *recommendation* is in “clear text” as they just use HTTP. This means that another team, with access to your cluster, could deploy their own service and attempt to sniff the traffic flowing through the system. To make that point, open up two command shells where one is using `tcpdump` to sniff traffic while the other is performing a `curl` command.

Shell 1:

```
PREFPOD=$(oc get pod -n tutorial -l app=preference -o \
'jsonpath={.items[0].metadata.name}')

oc exec -it $PREFPOD -n tutorial -c istio-proxy /bin/bash

sudo tcpdump -A -s 0 \
'tcp port 8080 and (((ip[2:2]-((ip[0]&0xf)<<2))-
                                  ((tcp[12]&0xf0)>>2))!= 0)'
```

Shell 2:

```
PREFPOD=$(oc get pod -n tutorial -l app=preference -o \
'jsonpath={.items[0].metadata.name}')

oc exec -it $PREFPOD -n tutorial -c preference /bin/bash

curl recommendation:8080
```

The results for Shell 1 should look similar to the following:

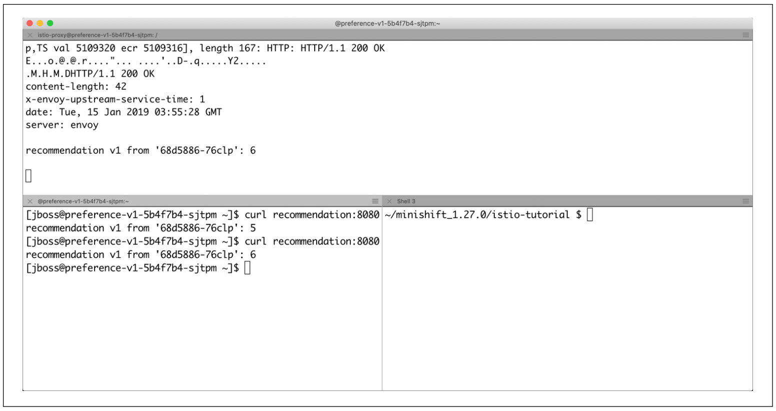
```
..:...:.HTTP/1.1 200 OK
content-length: 47
x-envoy-upstream-service-time: 0
date: Mon, 24 Dec 2018 17:16:13 GMT
server: envoy

recommendation v1 from '66b7c9779c-75fpl': 345
```

And the results for Shell 2 will be:

```
recommendation v1 from '66b7c9779c-75fpl': 345
```

The `curl` command works and the `tcpdump` command outputs the results in clear text as seen in Figure 7-1.


@preference-v1-5b4f7b4-sjtpm:~
istio-proxy@preference-v1-5b4f7b4-sjtpm: /
@preference-v1-5b4f7b4-sjtpm:~
Shell 3

```
p,TS val 5109320 ecr 5109316], length 167: HTTP: HTTP/1.1 200 OK
E...o.@.@.r...."... ....'..D-.q.....Y2.....
.M.H.M.DHTTP/1.1 200 OK
content-length: 42
x-envoy-upstream-service-time: 1
date: Tue, 15 Jan 2019 03:55:28 GMT
server: envoy

recommendation v1 from '68d5886-76clp': 6
```

```
[jboss@preference-v1-5b4f7b4-sjtpm ~]$ curl recommendation:8080
recommendation v1 from '68d5886-76clp': 5
[jboss@preference-v1-5b4f7b4-sjtpm ~]$ curl recommendation:8080
recommendation v1 from '68d5886-76clp': 6
[jboss@preference-v1-5b4f7b4-sjtpm ~]$
```

```
~/minishift_1.27.0/istio-tutorial $
```

Figure 7-1. Three shells before mTLS policy

Enabling mTLS in Istio uses the combination of `Policy` and `DestinationRule` objects. The Policy declaration is as follows:

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

This applies to all services in the tutorial namespace. You can also optionally set a "mode" of PERMISSIVE versus STRICT. which allows for both mTLS and non-mTLS traffic, useful for scenarios where the sidecar, istio-proxy, has not yet been applied to all of your services. See the documentation on mTLS Migration for an example of how to have both legacy no-sidecar services as well as services that have the sidecar applied.

For now, our example services of *customer*, *preference*, and *recommendation* all have the sidecar injected, so we can apply mTLS across the whole of the tutorial namespace.

In a third shell, apply the policy manifest that was provided in the Istio Tutorial when you git cloned the repository.

Shell 3:

```
oc apply -n tutorial -f istiofiles/authentication-enable-tls.yml
```

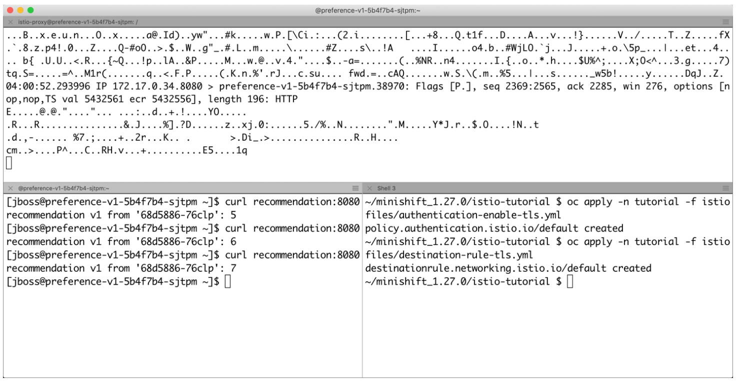
Now, apply the `DestinationRule` that enables mTLS amongst the services in the tutorial namespace

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

Shell 3:

```
oc apply -n tutorial -f istiofiles/destination-rule-tls.yml
```

And in Shell 2 run your `curl` command again, with successful results as seen in Figure 7-2:



@preference-v1-5b4f7b4-sjtpm:~
istio-proxy@preference-v1-5b4f7b4-sjtpm: /
@preference-v1-5b4f7b4-sjtpm:~
Shell 3

```
...B..x.e.u.n...0..x.....a@.Id)..yw"...#k.....w.P.[\Ci.:...(2.i........[...+8...Q.t1f...D....A...v...!}......V../.....T..Z.....fX
.`.8.z.p4!.0...Z....Q-#oO..>.$..W..g"_.#.L..m.....\......#Z....s\..!A    ....I......o4.b..#WjLO.`j...J.....+.o.\5p_...|...et...4..
.. b{ .U.U..<.R...{~Q...!p..lA..&P.....M...w.@..v.4."....$..-a=.......(..%NR..n4.......I.{..o..*.h....$U%^;....X;0<^...3.g.....7)
tq.S=.....=^..M1r(.......q..<.F.P.....(.K.n.%'.rJ...c.su....  fwd.=..cAQ.......w.S.\(.m..%5...|...s......_w5b!.....y......DqJ..Z.
04:00:52.293996 IP 172.17.0.34.8080 > preference-v1-5b4f7b4-sjtpm.38970: Flags [P.], seq 2369:2565, ack 2285, win 276, options [n
op,nop,TS val 5432561 ecr 5432556], length 196: HTTP
E.....@.@."....".. ...:..d..+.!....YO.....
.R...R...............&.J....%].?D......z..xj.0:......5./%..N........".M.....Y*J.r..$.O....!N..t
.d.,-...... %7.;....+..2r...K..  .       >.Di_.>..............R..H....
cm..>....P^...C..RH.v...+..........E5....1q
```

```
[jboss@preference-v1-5b4f7b4-sjtpm ~]$ curl recommendation:8080
recommendation v1 from '68d5886-76clp': 5
[jboss@preference-v1-5b4f7b4-sjtpm ~]$ curl recommendation:8080
recommendation v1 from '68d5886-76clp': 6
[jboss@preference-v1-5b4f7b4-sjtpm ~]$ curl recommendation:8080
recommendation v1 from '68d5886-76clp': 7
[jboss@preference-v1-5b4f7b4-sjtpm ~]$
```

```
~/minishift_1.27.0/istio-tutorial $ oc apply -n tutorial -f istio
files/authentication-enable-tls.yml
policy.authentication.istio.io/default created
~/minishift_1.27.0/istio-tutorial $ oc apply -n tutorial -f istio
files/destination-rule-tls.yml
destinationrule.networking.istio.io/default created
~/minishift_1.27.0/istio-tutorial $
```

Figure 7-2. Three shells after mTLS DestinationRule

You should notice that the `tcpdump` shell is no longer providing clear text and your `curl` command executes successfully. You can also use the *istioctl* tool to verify if mTLS is enabled:

```
istioctl authn tls-check | grep tutorial
```

Now it is time to test from the external world's perspective. In Shell 2, exit from the *preference* container back to your host OS. Then `curl` the customer endpoint, which results in "Empty reply from server":

```
curl customer-tutorial.$(minishift ip).nip.io
curl: (52) Empty reply from server
```

This particular external URL was generated via an OpenShift Route and minishift leverages a special service called nip.io for DNS resolution. Now that you have enabled mTLS, you need to leverage a gateway to achieve end-to-end encryption. Istio has its own ingress gateway, aptly named Istio Gateway, a solution that exposes a URL external to the cluster and supports Istio features such as monitoring, traffic management, and policy.

To set up the Istio Gateway for the *customer* service, create the `Gateway` and its supporting `VirtualService` objects

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: customer-gateway
  namespace: tutorial
spec:
  selector:
    istio: ingressgateway # use istio default controller
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "*"
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: customer
  namespace: tutorial
spec:
  hosts:
  - "*"
  gateways:
  - customer-gateway
  http:
  - match:
    - uri:
        exact: /
    route:
    - destination:
        host: customer
        port:
          number: 8080
```

And you can apply these manifests:

```
oc apply -f istiofiles/gateway-customer.yml
```

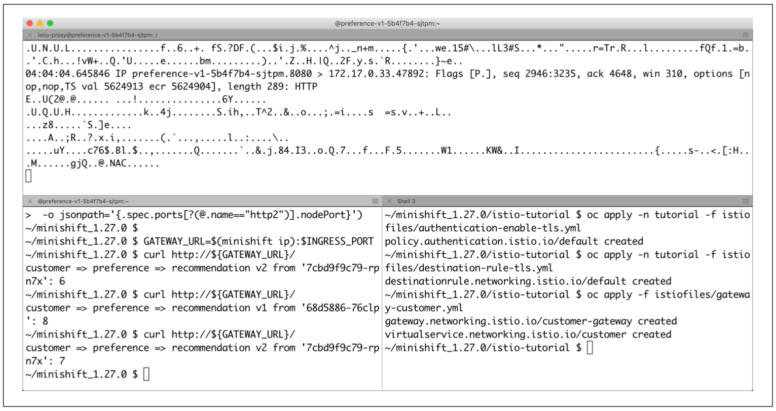
On minishift or minikube, the Istio Gateway service exposes a NodePort to make it visible outside the cluster. This port is available via minishift or minikube's IP address:

```
INGRESS_PORT=$(oc -n istio-system get service
                                      istio-ingressgateway \
 -o jsonpath='{.spec.ports[?(@.name=="http2")].nodePort}')

GATEWAY_URL=$(minishift ip):$INGRESS_PORT

curl http://${GATEWAY_URL}/
```

Issuing `curl` commands via the gateway works as expected as seen in Figure 7-3.



```
.U.N.U.L................f..6..+. fS.?DF.(...$i.j.%....^j.._n+m.....{.'...we.15#\...lL3#S...*..."....r=Tr.R...l.........fQf.1.=b.
.'.C.h...!vW+..Q.'U.....e......bm.........)..'.Z..H.|Q..2F.y.s.`R........}~e..
04:04:04.645846 IP preference-v1-5b4f7b4-sjtpm.8080 > 172.17.0.33.47892: Flags [P.], seq 2946:3235, ack 4648, win 310, options [n
op,nop,TS val 5624913 ecr 5624904], length 289: HTTP
E..U(2@.@...... ...!..............6Y......
.U.Q.U.H.............k..4j........S.ih,..T^2..&..o...;.=i....s  =s.v..+..L..
...z8.....`S.]e....
....A..;R..?.x.i,.......(.`...,.....l..:....\..
.....uY....c76$.Bl.$..,.......Q.......`..&.j.84.I3..o.Q.7...f...F.5.......W1......KW&..I.......................{.....s-..<.[:H..
.M......gjQ..@.NAC......
```

```
>  -o jsonpath='{.spec.ports[?(@.name=="http2")].nodePort}')
~/minishift_1.27.0 $
~/minishift_1.27.0 $ GATEWAY_URL=$(minishift ip):$INGRESS_PORT
~/minishift_1.27.0 $ curl http://${GATEWAY_URL}/
customer => preference => recommendation v2 from '7cbd9f9c79-rp
n7x': 6
~/minishift_1.27.0 $ curl http://${GATEWAY_URL}/
customer => preference => recommendation v1 from '68d5886-76clp
': 8
~/minishift_1.27.0 $ curl http://${GATEWAY_URL}/
customer => preference => recommendation v2 from '7cbd9f9c79-rp
n7x': 7
~/minishift_1.27.0 $
```

```
~/minishift_1.27.0/istio-tutorial $ oc apply -n tutorial -f istio
files/authentication-enable-tls.yml
policy.authentication.istio.io/default created
~/minishift_1.27.0/istio-tutorial $ oc apply -n tutorial -f istio
files/destination-rule-tls.yml
destinationrule.networking.istio.io/default created
~/minishift_1.27.0/istio-tutorial $ oc apply -f istiofiles/gatewa
y-customer.yml
gateway.networking.istio.io/customer-gateway created
virtualservice.networking.istio.io/customer created
~/minishift_1.27.0/istio-tutorial $
```

Figure 7-3. Three shells with Istio customer gateway

Clean up:

```
oc delete -n tutorial -f istiofiles/gateway-customer.yml
oc delete -n tutorial -f istiofiles/destination-rule-tls.yml
oc delete -n tutorial -f istiofiles/authentication-
                                    enable-tls.yml
```

And return to the original invocation mechanism with the OpenShift Route:

```
oc expose service customer

curl customer-tutorial.$(minishift ip).nip.io
```

Access Control with Mixer Policy

Istio's Mixer Policy service allows you to construct a series of rules that ensure the various microservices that make up your application follow an approved invocation path. In the case of the example services, it is expected that *customer* calls *preference* and then *preference* calls *recommendation*, in that specific order. Therefore there are some alternative paths that are specifically denied:

- *customer* is not allowed to call *recommendation*
- *preference* is not allowed to call *customer*
- *recommendation* is not allowed to call *customer*
- *recommendation* is not allowed to call *preference*

Istio has some additional objects or Kinds involved in this sort of access control: `denier`, `checknothing`, and `rule`.

Before you apply the `denier` and `rules`, first explore the potential hazard associated with *customer*, *preference*, and *recommendation* as it may not be apparent at first glance.

First, grab the recommendation pod's name/id and exec into its business logic container:

```
RECPOD=$(oc get pod -n tutorial -l app=recommendation -o \
'jsonpath={.items[0].metadata.name}')

oc exec -it $RECPOD -n tutorial -c recommendation /bin/bash
```

Next, `curl` the *customer* service and see that it succeeds, because all the services are visible to one another by default (as expected in a Kubernetes/OpenShift cluster):

```
curl customer:8080
customer => preference => recommendation v2 from
                                          '7cbd9f9c79': 23
```

Also `curl` the *preference* service:

```
curl preference:8080
preference => recommendation v1 from '66b7c9779c': 152
```

And for the sake of argument, the business/organization has determined that only the customer → preference → recommendation invocation path is the correct one. You can close down the others with the `denier` and `rules`. The listing of these rules is a bit long but make sure to check the manifest before applying it.

The syntax for the `rules` are straightforward, based on source and destination labels. You can check your pod labels:

```
oc get pods -n tutorial --show-labels
```

This has an output similar to the following (output truncated here for formatting reasons):

```
NAME                          READY STATUS   LABELS
customer-6564ff969f           2/2   Running  app=customer,
                                                 version=v1
preference-v1-5485dc6f49      2/2   Running  app=preference,
                                                 version=v1
recommendation-v1-66b7c9779c  2/2   Running  app=recommendation,
                                                 version=v1
recommendation-v2-7cbd9f9c79  2/2   Running  app=recommendation,
                                                 version=v2
```

And you can apply these rules with the following command:

```
oc -n tutorial apply -f \
istiofiles/acl-deny-except-
           customer2preference2recommendation.yml
```

Now when you exec into *recommendation* and attempt a `curl` of *preference*:

```
curl preference:8080
PERMISSION_DENIED:do-not-pass-go.denier.tutorial:
Customer -> Preference -> Recommendation ONLY
```

Make sure to double-check that your normal invocation path continues to execute as expected:

```
curl customer-tutorial.$(minishift ip).nip.io
customer => preference => recommendation v2 from
                                         '7cbd9f9c79': 238
```

Use the `describe` verb for the `kubectl` or `oc` tool to see the rules you have in place:

```
oc get rules
NAME                               AGE
no-customer-to-recommendation      3m
no-preference-to-customer          3m
no-recommendation-to-customer      3m
no-recommendation-to-preference    3m

oc describe rule no-preference-to-customer
...
Spec:
  Actions:
    Handler:  do-not-pass-go.denier
    Instances:
      just-stop.checknothing
  Match:  source.labels["app"]=="preference" &&
   destination.labels["app"] == "customer"
Events:   <none>
```

You can remove these rules to return to original state:

```
oc delete rules --all -n tutorial
```

Istio's Mixer also supports a *whitelist* and *blacklist* mechanism involving the `listchecker` and `listentry` objects. If you are interested in that capability check out the Istio Tutorial and/or the Istio Documentation.

Role-Based Access Control (RBAC)

Istio includes a Role-Based Access Control (RBAC) authorization feature that can be used to further constrain which services (e.g., *customer*, *preference*, *recommendation*) are accessible by particular users.

Make sure there are no pre-existing `DestinationRule`, `VirtualSer vice`, `Gateway`, or `Policy` objects:

```
oc get destinationrule -n tutorial
oc get virtualservice -n tutorial
oc get gateway -n tutorial
oc get policy -n tutorial
```

Setting up Istio's RBAC support is simple enough using the RbacConfig object:

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

Where *mode* can be:

- OFF: Istio authorization is disabled.
- ON: Istio authorization is enabled for all services in the mesh.
- ON_WITH_INCLUSION: Enabled only for services and namespaces specified in the inclusion field.
- ON_WITH_EXCLUSION: Enabled for all services in the mesh except the services and namespaces specified in the exclusion field.

That is the line required to apply/create the RbacConfig object:

```
oc create -f istiofiles/authorization-enable-rbac.yml
                        -n tutorial
```

Now if you `curl` your customer endpoint, you will receive “RBAC: access denied”:

```
curl customer-tutorial.$(minishift ip).nip.io
RBAC: access denied
```

Istio’s RBAC uses a deny-by-default strategy, meaning that nothing is permitted until you explicitly define an access-control policy to grant access to any service. To reopen the customer endpoint to end-user traffic, create a `ServiceRole` and a `ServiceRoleBinding`:

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

Apply it:

```
oc -n tutorial apply -f \
istiofiles/namespace-rbac-policy.yml
```

Now try your `curl` command again:

```
curl customer-tutorial.$(minishift ip).nip.io
customer => preference => recommendation v1 from
                                            '66b7c9779c': 36
```

Istio’s `ServiceRole` object allows you to specify which services are protected either by naming them directly or using the `destina` `tion.labels[app]` constraint demonstrated in the example. You can also specify which methods are allowed such as GET versus POST. The `ServiceRoleBinding` object allows you to specify which users are permitted. In the current case, user: "*" with no additional properties means that any user is allowed to access these services.

The concept of users and user management has always been unique per organization, often unique per application. In the case of a Kubernetes cluster, your cluster administrator will likely have a preferred strategy for user authentication and authorization.

Istio has support for user authentication and authorization via JWT (JSON Web Token). From the "Introduction to JSON Web Tokens" page: "JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object." To leverage JWT, you will need a JWT issuer like auth0.com, or perhaps a local service based on the open source software project called Keycloak, which supports OpenID Connect, OAuth 2.0, and SAML 2.0.

Setup and configuration of a JWT Issuer is beyond the scope of this book, but more information can be found at the Istio Tutorial.

In addition, Istio Security has more information about Istio's security capabilities.

Conclusion

You have now taken a relatively quick tour through some of the capabilities of Istio service mesh. You saw how this service mesh can solve distributed systems problems in cloud native environments, and how Istio concepts like observability, resiliency, and chaos injection can be immediately beneficial to your current application.

Moreover, Istio has capabilities beyond those we discussed in this book. If you're interested, we suggest that you explore the following topics more deeply:

- Policy enforcement
- Mesh expansion
- Hybrid deployments
- Phasing in Istio into an existing environment

- Gateway/Advanced ingress

Istio is also evolving at a rapid rate. To keep up with the latest developments, we suggest that you keep an eye on the upstream community project page as well as Red Hat's evolving Istio Tutorial.

About the Authors

Burr Sutter (@burrsutter) is a lifelong developer advocate, community organizer, technology evangelist, and featured speaker at technology events around the globe—from Bangalore to Brussels and Berlin to Beijing (and most parts in between). He is currently Red Hat's Director of Developer Experience. A Java Champion since 2005 and former president of the Atlanta Java User Group, Burr founded the DevNexus conference, now the second-largest Java event in the United States. When spending time away from the computer, he enjoys going off-grid in the jungles of Mexico and the bush of Kenya. You can find Burr online at *burrsutter.com*.

Christian Posta (@christianposta) is Field CTO at solo.io and well known in the community for being an author (*Istio in Action*, Manning; *Microservices for Java Developers*, O'Reilly), frequent blogger, speaker, open-source enthusiast and committer on various open-source projects including Istio and Kubernetes. Christian has spent time at web-scale companies and now helps companies create and deploy large-scale, resilient, distributed architectures—many of what we now call Serverless and Microservices. He enjoys mentoring, training, and leading teams to be successful with distributed systems concepts, microservices, devops, and cloud native application design.