O'REILLY®

Compliments of RED HAT DEVELOPER PROGRAM

Introducing Istio Service Mesh for Microservices

Build and Deploy Resilient, Fault-Tolerant Cloud-Native Applications

Christian Posta & Burr Sutter

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

Printed in the United States of America.

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

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

Editors: Brian Foster and Alicia Young
Production Editor: Colleen Cole
Copyeditor: Octal Publishing, Inc.
Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Rebecca Demarest
Technical Reviewer: Lee Calcote

April 2018: First Edition

Revision History for the First Edition
2018-04-05: First Release

978-1-491-98874-9

[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, WildFly Swarm, and Dropwizard, check out *Microservices for Java Developers* (O'Reilly).

Also, if you are interested in Reactive microservices, an excellent place to start is *Building Reactive Microservices in Java* (O'Reilly) because 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) is an excellent free ebook on that very topic. We will be deploying, interacting, and configuring Istio through the lens of OpenShift, however, the commands we use are portable to vanilla Kubernetes as well.

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

The Challenge of Going Faster

As a software development community, in the era of digital transformation, you have embarked on a relentless pursuit of better serving customers and users.

Today's digital creators, the application programmer, have not only evolved into faster development cycles based on Agile principles, you 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 you make.

Furthermore, the fact that you now have moved from a single deployment every few months to potentially dozens of software deployments happening every week, if not every day, brings with it many new challenges. One simple example is how do you manage to create a more frictionless deployment model that allows code being checked into a source code manager (e.g., Git) to more easily flow through the various stages of your workflow, from dev to code review, to QA, to security audit/review, to a staging environment and finally into production.

Some of the big web companies had to put frameworks and libraries into place to help alleviate some of the challenges of an unreliable network and many code deployments per day. For example, companies like Netflix created projects like Netflix OSS 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 very difficult to bring in new services written in programming languages that didn't have support from these resilience frameworks they created. Whenever these resilience frameworks were updated, the applications also needed to be updated to stay in lock step. Finally, even if they created an implementation of these resiliency frameworks for every possible permutation of language or framework, they'd have massive overhead in trying to maintain this and apply the functionality consistently. Getting these resiliency frameworks right is tough when trying to implement in multiple frameworks and languages. Doing so means redundancy of effort, mistakes, and non-uniform set of behaviors. 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 and a single programming language. 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 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 and Express) and libraries (e.g., jars and npms). However, OpenShift does not get involved with how each of the application components, running in their individual pods, interacts 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 allow 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 complimentary 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 a 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 top-

ology 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 be subjected to not only 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 explanation 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 is.

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*, which is depicted in Figure 1-1.

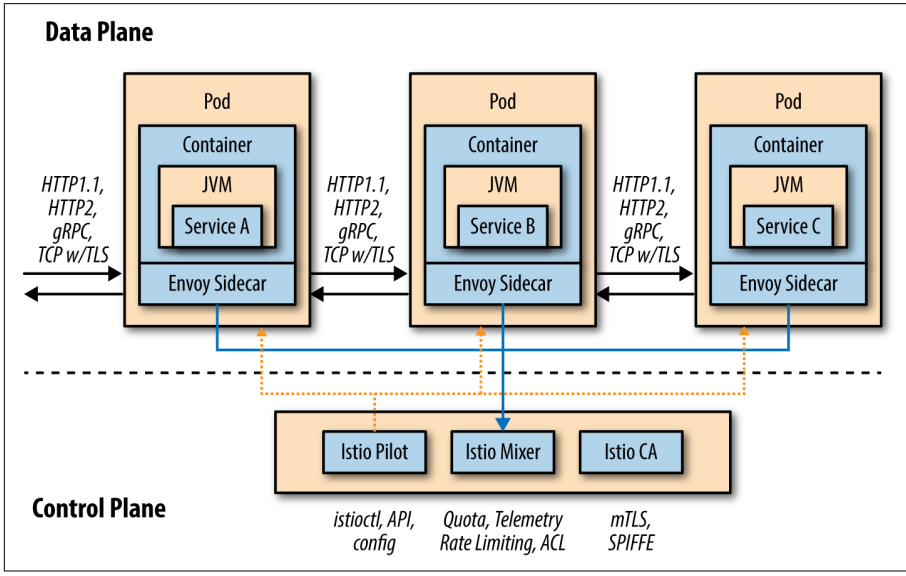

Figure 1-1. Data plane versus control plane

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's RestTemplate and 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.

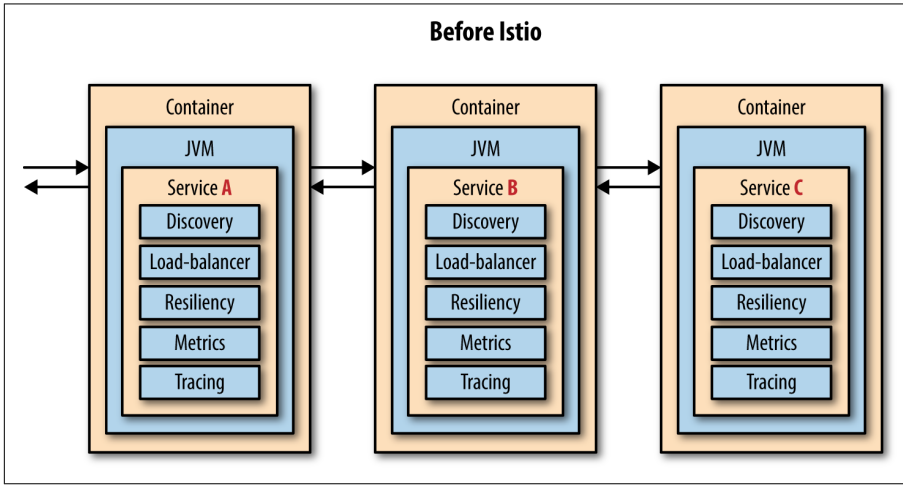

Figure 1-2. Before Istio

The data plane for Istio service mesh is made up of two simple concepts: *service proxy* and *sidecar container*, as shown in Figure 1-3.

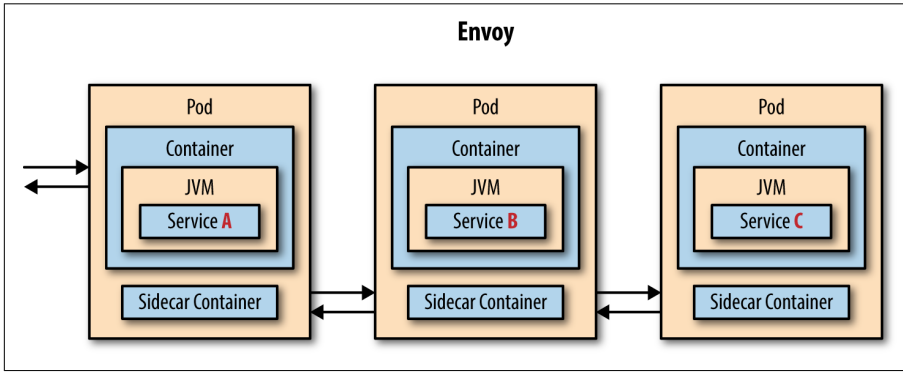

Figure 1-3. With Envoy sidecar (istio-proxy)

Let's explore each concept.

Service proxy

A service proxy is a proxy on which an application service relies for additional capabilities. The service calls through the service proxy any time it needs to communicate with the outside world (i.e., over the network). The proxy acts as an intermediary or interceptor that can add capabilities like automatic retries, timeouts, circuit breaker, service discovery, security, and more. The default service proxy for Istio is based on Envoy Proxy.

Envoy Proxy is a Layer 7 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 (among many others). Written in C++, it is a battle tested, highly performant, and lightweight. It provides features like load balancing for HTTP1.1, HTTP2, gRPC, the ability to collect request-level metrics, tracing spans, active and passive health checking, service discovery, and many 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? A service proxy could be deployed like other popular proxies in which many services' requests get serviced by a single 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 you manage in a Kubernetes/OpenShift world. Why pod? Some think it was some 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 being 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 the real-world sidecar that bolts on to 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 alongside your primary business logic container. 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), 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 error responses.

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 Auth.

Pilot

The Pilot is responsible for managing the overall fleet and all of your microservices running across your Kubernetes/OpenShift cluster. The Istio Pilot is responsible for ensuring that each of the independent and distributed microservices, wrapped as Linux containers and 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 RouteRule and DestinationPolicy. The RouteRule is what gives you that finely grained request distribution. We cover this in more detail in Chapter 3. The DestinationPolicy helps you to address resiliency with timeouts, retries, circuit breaker, and so on. We discuss DestinationPolicy in 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 or pods. With Mixer, you can create ACLs (whitelist and blacklist), you can 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 will be extending Mixer's default capabilities in many new and interesting ways. Many of the capabilities of Mixer fall beyond the scope of this book, but we do address observability in Chapter 6, and security in Chapter 7.

If you would like to explore Mixer further, refer to the upstream project documentation as well as the Istio Tutorial for Java Microservices maintained by the Red Hat team.

Auth

The Istio Auth component, also known as Istio CA, is responsible for certificate signing, certificate issuance and revocation/rotation. Istio issues x509 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.

CHAPTER 2

Installation and Getting Started

Command-Line Tools Installation

In this section, we show you how to get started with Istio on Kubernetes. Istio is not tied to Kubernetes in anyway, and in fact, it's intended to be agnostic of any deployment infrastructure. 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 flavor of an enterprise distribution of Kubernetes named OpenShift.

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

- minishift
- Docker for Mac/Windows
- kubectl
- oc client tools for your OS (note: "minishift oc-env" will output the path to the oc client binary)
- mvn
- stern for easily viewing logs
- siege for load testing
- istioctl (will be installed via the steps that follow momentarily)
- curl, tar part of your bash/cli shell
- Git

Kubernetes/OpenShift Installation

When you bootstrap minishift, you'll need to keep in mind 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 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. (One thing to remember: on minishift, the default pod limit is set to 10 times the number of CPUs.)

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

```
#!/bin/bash

# add the location of minishift executable to PATH
# I also keep other handy tools like kubectl and kubetail.sh
# in that directory

export MINISHIFT_HOME=~/minishift_1.12.0
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 config set openshift-version v3.7.0

minishift start
```

When things have launched correctly, you should be able 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     1d        v1.7.6+a08f5eeb62
```

If you have errors along the way, review the current steps of the *istio-tutorial* and potentially file a GitHub issue.

Istio Installation

Istio distributions come bundled with the necessary binary command-line interface (CLI) tools, installation resources, and sample applications. You should download the Istio 0.5.1 release:

```
curl -L https://github.com/istio/istio/releases/download/0.5.1/istio-0.5.1 \
 -osx.tar.gz | tar xz
cd istio-0.5.1
```

Now you need to prepare your OpenShift/Kubernetes environment. OpenShift has a series of features targeted toward safe, multitenant runtimes and therefore has tight security restrictions. To install Istio, for the moment you can relax those OpenShift security constraints. The Istio community is working hard to make Istio more secure and fit better within the expectations of a modern enterprise's security requirements, striving for "secure by default" with no developer pain.

For now, and for the purposes of understanding Istio and running these samples on OpenShift, let's relax these security constraints. Using the `oc` command-line tool, run the following:

```
oc adm policy add-scc-to-user anyuid -z istio-ingress-service-account \
 -n istio-system
oc adm policy add-scc-to-user anyuid -z default -n istio-system
oc adm policy add-scc-to-user anyuid -z prometheus -n istio-system
```

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

```
oc create -f install/kubernetes/istio.yaml
oc project istio-system
```

This will install all of the necessary Istio control-plane components including Istio Pilot, Mixer, and Auth. You should also install some companion services that are useful for metrics collection, distributed tracing, and overall visualization of our services. Run the following from the root folder of the Istio distribution:

```
oc apply -f install/kubernetes/addons/prometheus.yaml
oc apply -f install/kubernetes/addons/grafana.yaml
oc apply -f install/kubernetes/addons/servicegraph.yaml
oc process -f  https://raw.githubusercontent.com/jaegertracing/jaeger- \
 openshift/master/all-in-one/jaeger-all-in-one-template.yml | oc create -f -
```

This installs Prometheus for metric collection, Grafana for metrics dashboard, Servicegraph for simple visualization of services and Jaeger for distributed-tracing support.

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

```
oc expose svc servicegraph
oc expose svc grafana
oc expose svc prometheus
oc expose svc istio-ingress
```

At this point, all of 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-3617079618-4qs2b          1/1       Running   0          4m
istio-ca-1363003450-tfnjp         1/1       Running   0          4m
istio-ingress-1005666339-vrjln    1/1       Running   0          4m
istio-mixer-465004155-zn78n       3/3       Running   0          5m
istio-pilot-1861292947-25hnm      2/2       Running   0          4m
jaeger-210917857-2w24f            1/1       Running   0          4m
prometheus-168775884-dr5dm        1/1       Running   0          4m
servicegraph-1100735962-tdh78     1/1       Running   0          4m
```

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 files. 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-0.5.1
export PATH=$ISTIO_HOME/bin:$PATH
```

Now, from your command line you should be able to type the following and see a valid response:

```
istioctl version
Version: 0.5.1
GitRevision: c9debceacb63a14a9ae24df433e2ec3ce1f16fc7
User: root@211b132eb7f1
Hub: docker.io/istio
GolangVersion: go1.9
BuildStatus: Clean
```

At this point, you're ready to 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 service would allow customers to set preferences for certain aspects of the website. Those preferences will have the opportunity to take rec-

ommendations 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 *https://github.com/redhat-developer-demos/istio-tutorial* and switch to the branch book, as demonstrated here:

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

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 the respective services we'll use to demonstrate Istio capabilities.

The *customer* and *preference* services are both Java Spring Boot implementations. Have a look at the source code. You should see fairly straightforward implementations of REST services. For example, here's the endpoint for the *customer* service:

```
@RequestMapping("/")
public ResponseEntity<String> getCustomer() {
  try {
    String response = restTemplate.getForObject(remoteURL,
         String.class);
    return ResponseEntity.ok(String.format(RESPONSE_STRING_FORMAT,
         response.trim()));
  } catch (HttpStatusCodeException ex) { ....  }
}
```

We've left out the exception handling for a moment. You can see that this HTTP endpoint simply calls out to the *preference* service and returns the response from *preference* prepended with a fixed string of `customer => %s`. Note that there are no additional libraries that we use beyond Spring's 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 comingle 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

catch 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 *preferences* 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 don't need special libraries to do this for you.

If you peruse the code base for the *customer* service a bit more, you might stumble upon two classes named `HttpHeaderForwarderHandlerInterceptor` and `HttpHeaderForwarderClientHttpRequestInterceptor`. These classes work together to intercept any incoming headers used for tracing and propagates them for further downstream requests. These headers are the OpenTracing headers and are defined in this immutable variable:

```
private static final Set<String> FORWARDED_HEADER_NAMES =
     ImmutableSet.of(
        "x-request-id",
        "x-b3-traceid",
        "x-b3-spanid",
        "x-b3-parentspanid",
        "x-b3-sampled",
        "x-b3-flags",
        "x-ot-span-context",
        "user-agent"
);
```

These headers are used to correlate requests together, submit spans to tracing systems, and can be used for request/response timing analysis, diagnostics, and debugging. Although our little helper interceptor is responsible for shuffling the `x-b3-*` headers along it does *not* communicate with the tracing system directly. In fact, you don't need to include any tracing libraries in your application code or dependency tree. When you propagate these headers, Istio is smart enough to recognize them and submit the proper spans to your tracing backend. Throughout the examples and use cases in this book, we use the Jaeger Tracing project from the Cloud Native Computing Foundation (CNCF). You can learn more about Jaeger Tracing at *http://jaegertracing.io/*. You installed Jaeger as part of the Istio installation in the previous section.

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.

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
```

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. You can verify your status by using the following command:

```
oc status
```

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

```
cd customer
mvn clean package
```

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

```
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/src/main/kubernetes* directory, there are two Kubernetes resource files named *Deployment.yml* and *Service.yml*. Deploy the service and also deploy your application with the Istio sidecar proxy injected into it. Try running the following command to see what the injected sidecar looks like with your deployment:

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

Examine this output 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
          - 2s
          - --parentShutdownDuration
          - 3s
          - --discoveryAddress
          - istio-pilot.istio-system:15003
          - --discoveryRefreshDelay
```

```
- 1s
- --zipkinAddress
- zipkin.istio-system:9411
- --connectTimeout
- 1s
- --statsdUdpAddress
- istio-mixer.istio-system:9125
- --proxyAdminPort
- "15000"
- --controlPlaneAuthPolicy
- NONE
env:
- name: POD_NAME
  valueFrom:
    fieldRef:
      fieldPath: metadata.name
- name: POD_NAMESPACE
  valueFrom:
    fieldRef:
      fieldPath: metadata.namespace
- name: INSTANCE_IP
  valueFrom:
    fieldRef:
      fieldPath: status.podIP
image: docker.io/istio/proxy:0.5.1
imagePullPolicy: IfNotPresent
name: istio-proxy
resources: {}
securityContext:
  privileged: false
  readOnlyRootFilesystem: true
  runAsUser: 1337
volumeMounts:
- mountPath: /etc/istio/proxy
  name: istio-envoy
- mountPath: /etc/certs/
  name: istio-certs
  readOnly: true
```

You will see a second container injected into your deployment with configurations for finding the Istio control plane, volume mounts which mount in any additional secrets, and you should see the name of this container is `istio-proxy`.

Now you can create the Kubernetes service and inject the sidecar into your deployment:

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

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

Because customer is the forwardmost microservice (*customer* > *preference* > *recommendation*), you should add an OpenShift Route that exposes that endpoint:

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

Note 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.

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; nested exception is java.net.UnknownHostException: preference
```

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

mvn clean package

docker build -t example/preference .
```

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 \
src/main/kubernetes/Deployment.yml) -n tutorial

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

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

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

The response still fails, but a little bit differently this time:

```
customer => 503 preference => I/O error on GET request for
"http://recommendation:8080": recommendation; nested exception is
java.net.UnknownHostException: recommendation
```

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 the full cooperation of our services working nicely is to deploy the recommendation service. Just like in the previous services, you will build, package, and deploy onto Kubernetes by using a couple of steps:

```
cd recommendation

mvn clean package

docker build -t example/recommendation:v1 .

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

oc create -f src/main/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 see a better response:

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

customer => preference => recommendation v1 from '99634814-sf4cl': 1
```

Success! The chain of calls between the three services works as expected. Now that you have your services calling one another, we move on to discussing some of the core capabilities of Istio and the power it brings for solving the problems that arise *between* services.

Building and Deploying to Kubernetes

Kubernetes deploys and manages applications that have been built as Docker containers. In the preceding examples, you built and packaged the applications into Docker containers at each step. There are alternatives to the fully manual deployment process of `docker build` and `oc create -f someyaml.yml`. These alternatives include `oc new-app` and a capability known as *source-to-image* (S2I). S2I is an OpenShift-only feature that is not compatible with vanilla Kubernetes. There is also the fabric8-maven-plugin, which is a maven plug-in for Java applications. fabric8-maven-plugin allows you to live comfortably in your existing Java tooling and still build Docker images and interact with Kubernetes without having to know about Dockerfiles or Kubernetes resource files. The maven plug-in automatically builds the Kubernetes resource files and you can also use it to quickly deploy, undeploy, and debug your Java application running in Kubernetes.

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 take a 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 the 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 "canary deployment" comes from the "canary in the coal mine" concept. Miners used to take a canary in a cage into the mines to detect whether there were any dangerous gases present because the canaries are more susceptible to poisonous 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 coal mine rapidly.

The canary deployment has similar semantics. With a canary deployment, you deploy a new version of your code to production, but you 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 it exhibits no bad behavior, you can begin to slowly deploy more instances of the new version of code. If it exhibits bad behav-

ior, you can pull it from production. The canary deployment allows you to deploy faster but with minimal disruption should a "bad" code change be made.

By default, Kubernetes offers out-of-the-box round-robin load balancing of all the pods behind a service. If you want only 10% of all end-user traffic to hit your newest immutable container, 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 the 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

As we touched on previously, Istio allows much more fine-grained canary deployments. With Istio, you can specify routing rules that control the traffic to a deployment. Specifically, Istio uses a `RouteRule` resource to specify these rules. Let's take a look at an example `RouteRule`:

```
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
  name: recommendation-default
spec:
  destination:
    namespace: tutorial
    name: recommendation
  precedence: 1
  route:
  - labels:
      version: v1
    weight: 100
```

This `RouteRule` 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 labels `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 described above is *not* just for *ingress* traffic; that is, traffic coming into the mesh. This is for all interservice 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 (as just mentioned).

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 String` in the `com.redhat.developer.demos.recommendation.RecommendationVerticle` to be something like this:

```
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

mvn clean package

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

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

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

You can run `oc get pods -w` to watch the pods and wait until they all come up. You should see something like this when all of 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-v1-60483540-9snd9    2/2     Running   0          12m
recommendation-v2-2815683430-vpx4p  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 '2819441432-qsp25': 29
customer => preference => recommendation v2 from '99634814-sf4cl': 37
customer => preference => recommendation v1 from '2819441432-qsp25': 30
customer => preference => recommendation v2 from '99634814-sf4cl': 38
customer => preference => recommendation v1 from '2819441432-qsp25': 31
customer => preference => recommendation v2 from '99634814-sf4cl': 39
```

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

```
istioctl create -f istiofiles/route-rule-recommendation-v1.yml \
 -n tutorial
```

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

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

customer => preference => recommendation v1 from '1543936415': 1
customer => preference => recommendation v1 from '1543936415': 2
customer => preference => recommendation v1 from '1543936415': 3
customer => preference => recommendation v1 from '1543936415': 4
customer => preference => recommendation v1 from '1543936415': 5
customer => preference => recommendation v1 from '1543936415': 6
```

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 90% of the incoming live traffic. To do this, you need to specify a weighted routing rule in new `RouteRule` that looks like this:

```
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
  name: recommendation-v1-v2
spec:
  destination:
    namespace: tutorial
    name: recommendation
  precedence: 5
  route:
  - labels:
      version: v1
    weight: 90
  - labels:
      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 `RouteRule`. An important thing to notice about this `RouteRule` is the `precedence` value. In the preceding example, we set this to `5` for this route rule, which means it has *higher* precedence than the earlier rule that routes all traffic to v1. Try creating it and see what happens when you put load on the service:

```
istioctl create -f istiofiles/route-rule-recommendation-v1_and_v2.yml \
-n tutorial
```

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. You should monitor your logs, metrics, and tracing systems to see whether this new release has introduced any negative unintended 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. You can do that with another `RouteRule` that looks like this:

```
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
  name: recommendation-v1-v2
spec:
  destination:
    namespace: tutorial
    name: recommendation
  precedence: 5
  route:
  - labels:
      version: v1
    weight: 50
  - labels:
      version: v2
    weight: 50
```

With this `RouteRule` we're going to open the traffic up to 50% to v1, and 50% to v2. Notice the precedence is still the same value (it's `5`, just like the canary release) and that the route rule's name is the same as the canary release (`recommendation-v1-v2`). When you create this route rule using `istioctl`, you should use the `replace` command:

```
istioctl replace -f \
istiofiles/route-rule-recommendation-v1_and_v2_50_50.yml -n tutorial
```

Now you should see traffic behavior change in real time. You should see approximately half the traffic go to v1 of the *recommendation* service and half go to v2. You should see something like the following:

```
customer => preference => recommendation v1 from '1543936415': 192
customer => preference => recommendation v2 from '3116548731': 37
customer => preference => recommendation v2 from '3116548731': 38
customer => preference => recommendation v1 from '1543936415': 193
customer => preference => recommendation v2 from '3116548731': 39
customer => preference => recommendation v2 from '3116548731': 40
customer => preference => recommendation v2 from '3116548731': 41
customer => preference => recommendation v1 from '1543936415': 194
customer => preference => recommendation v2 from '3116548731': 42
```

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

```
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
  name: recommendation-default
spec:
  destination:
    namespace: tutorial
    name: recommendation
  precedence: 1
  route:
  - labels:
      version: v2
    weight: 100
```

You can replace the v1 route rule like this:

```
istioctl replace -n tutorial -f \
istiofiles/route-rule-recommendation-v2.yml
```

Note that the precedence for this route rule is set to 1. This means the traffic-control `RouteRules` you used in the previous steps, which had their precedence values set to `5` would still have higher precedence. That's true. You need to next delete the canary/rollout route rules so that all traffic matches the v2 `routerule`:

```
istioctl delete routerule -n tutorial recommendation-v1-v2
```

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

```
customer => preference => recommendation v2 from '3116548731': 308
customer => preference => recommendation v2 from '3116548731': 309
customer => preference => recommendation v2 from '3116548731': 310
customer => preference => recommendation v2 from '3116548731': 311
customer => preference => recommendation v2 from '3116548731': 312
customer => preference => recommendation v2 from '3116548731': 313
customer => preference => recommendation v2 from '3116548731': 314
customer => preference => recommendation v2 from '3116548731': 315
```

Restore route rules to v1

To clean up this section, replace the route rules to direct traffic back to v1 of the *recommendation* service:

```
istioctl replace -n tutorial -f \
istiofiles/route-rule-recommendation-v1.yml
```

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 `RouteRule` to specify a predicate. For example, take a look at the following `RouteRule`:

```
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
  name: recommendation-safari
spec:
  destination:
    namespace: tutorial
    name: recommendation
  precedence: 2
  match:
    request:
      headers:
        user-agent:
          regex: ".*Safari.*"
  route:
  - labels:
      version: v2
```

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. Note that this also has a precedence that's higher than the default route rule (recall, the default route rule routes every request to v1 and has a precedence of 1).

Install the rule:

```
$  istioctl create -f \
istiofiles/route-rule-safari-recommendation-v2.yml -n tutorial
```

And let's try it out:

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

customer => preference => recommendation v1 from '1543936415': 465
```

And 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 '3116548731': 318
```

Cleaning up route rules

After getting to this section, you can clean up all of the route rules you've installed. First, you should list the route rules you have using `istioctl get`:

```
$  istioctl get routerule -n tutorial

NAME                    KIND                                  NAMESPACE
recommendation-default  RouteRule.v1alpha2.config.istio.io  tutorial
recommendation-safari   RouteRule.v1alpha2.config.istio.io  tutorial
```

Now you can delete them:

```
istioctl delete routerule recommendation-safari -n tutorial
istioctl delete routerule recommendation-default -n tutorial
```

Dark Launch

Dark launch can mean different things to different people. In essence, a dark launch is a deployment to production that goes unnoticed to customer traffic. You might choose to release to a subset of customers (like internal or nonpaying customers) but the broader user base does not see the release. Another option is to duplicate or mirror production traffic into a cluster that has a new deployment and see how it behaves compared to the live traffic. 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 cluster. 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 cluster, and not worry about or care about a response. Let's try this out.

The first thing you should do is make sure that there are no route rules currently being used:

```
istioctl get routerules -n tutorial
```

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

```
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
  name: recommendation-mirror
spec:
  destination:
    namespace: tutorial
    name: recommendation
  precedence: 2
  route:
  - labels:
```

```
      version: v1
    weight: 100
  - labels:
      version: v2
    weight: 0
  mirror:
    namespace: tutorial
    name: recommendation
    labels:
      version: v2
```

You can see that this directs all traffic to v1 of *recommendation* and no traffic to v2. In the `mirror` clause, you specify which cluster to receive the mirrored traffic.

Next, verify you're in the root directory of the source files you gleaned from the *istio-tutorial* and run the following command:

```
istioctl create -f istiofiles/route-rule-recommendation-v1-mirror-v2.yml\
 -n tutorial
```

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

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

In another window, you can send in a request:

```
$  curl customer-tutorial.$(minishift ip).nip.io
customer => preference => recommendation v1 from '1543936415': 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.

You can use mirrored traffic to do powerful prerelease testing, but it does not come without its own 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 Microservices Databases*. For a more detailed treatment on advanced mirroring techniques, you can take a look at Christian's blog post "Advanced Traffic-shadowing Patterns for Microservices With Istio Service Mesh".

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 routing 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.

To demonstrate, we will have you create a service that makes a call out to an external website, namely, *httpbin.org*, and see how it behaves in the service mesh.

From the root of the companion source code you've cloned earlier, go to the *egress/egresshttpbin* folder. This is another Spring Boot Java application that does the following salient functionality:

```
@RequestMapping
public String headers() {
    RestTemplate restTemplate = new RestTemplate();
    String url = "http://httpbin.org/headers";

    HttpHeaders httpHeaders = new HttpHeaders();
    HttpEntity<String> httpEntity =
        new HttpEntity<>("", httpHeaders);

    String responseBody;
    try {
        ResponseEntity<String> response
        = restTemplate.exchange(url, HttpMethod.GET,
            httpEntity,
            String.class);
        responseBody = response.getBody();
    } catch (Exception e) {
        responseBody = e.getMessage();
    }
        return responseBody + "\n";
}
```

This HTTP endpoint, when called, will make a call out to *httpbin.org/headers*, which is a service residing on the public internet that returns a list of headers that were sent to the HTTP GET /headers endpoint.

Now you can build, package, and deploy and expose this service:

```
$  cd egress/egresshttpbin

$  mvn clean package

$  docker build -t example/egresshttpbin:v1 .

$  oc apply -f <(istioctl kube-inject -f \
src/main/kubernetes/Deployment.yml)
```

```
$  oc create -f src/main/kubernetes/Service.yml
$  oc expose service egresshttpbin
```

You should not be able to query the service like this:

```
$  curl http://egresshttpbin-tutorial.$(minishift ip).nip.io
```

You should see a response like this:

```
404 Not Found
```

Dang! This service cannot communicate with services in the public internet that live outside of our cluster!

Let's go back to the root of your source code and create an egress rule that looks like this:

```
apiVersion: config.istio.io/v1alpha2
kind: EgressRule
metadata:
  name: httpbin-egress-rule
spec:
  destination:
    service: httpbin.org
  ports:
    - port: 80
      protocol: http
```

This `EgressRule` allows your traffic to reach the outside internet but only for the *httpbin.org* website. Here, you can create the rule and try querying your service again:

```
istioctl create -f istiofiles/egress_httpbin.yml -n tutorial
```

You can list the egress rules like this:

```
$  istioctl get egressrule
NAME                    KIND                                    NAMESPACE
httpbin-egress-rule     EgressRule.v1alpha2.config.istio.io  tutorial
```

Now you can try to `curl` the service again:

```
curl http://egresshttpbin-tutorial.$(minishift ip).nip.io
```

Yay! It should have worked this time! Istio `EgressRules` allowed your service to reach the outside internet for this specific service. If you had a failure at this step, you can file a GitHub issue for the *istio-tutorial*.

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 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 autoremoval of error-prone pods from the load-balancing pool.

Let's take a look at each capability with an example. Here, we use the same set of services from the previous examples.

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 communicates and 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 `RouteRule`s to control how traffic is routed to specific clusters. In this chapter, we show you how to control the behavior of communicating with a particular cluster using `DestinationPolicy` rules. To begin, we discuss how to configure load balancing with Istio `DestinationPolicy` rules.

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

```
istioctl delete routerule --all
```

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 => preference => recommendation v1 from '99634814': 1145
customer => preference => recommendation v2 from '2819441432': 1
customer => preference => recommendation v2 from '2819441432': 2
customer => preference => recommendation v2 from '2819441432': 181
customer => preference => recommendation v1 from '99634814': 1146
customer => preference => recommendation v2 from '2819441432': 3
customer => preference => recommendation v2 from '2819441432': 4
customer => preference => recommendation v2 from '2819441432': 182
```

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

```
apiVersion: config.istio.io/v1alpha2
kind: DestinationPolicy
metadata:
  name: recommendation-loadbalancer
  namespace: tutorial
spec:
  source:
    name: preference
  destination:
    name: recommendation
  loadBalancing:
    name: RANDOM
```

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

Let's create this destination policy:

```
istioctl create -f istiofiles/recommendation_lb_policy_app.yml -n tutorial
```

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

```
customer => preference => recommendation v2 from '2819441432': 10
customer => preference => recommendation v2 from '2819441432': 3
customer => preference => recommendation v2 from '2819441432': 11
customer => preference => recommendation v1 from '99634814': 1153
customer => preference => recommendation v1 from '99634814': 1154
customer => preference => recommendation v1 from '99634814': 1155
customer => preference => recommendation v2 from '2819441432': 12
customer => preference => recommendation v2 from '2819441432': 4
customer => preference => recommendation v2 from '2819441432': 5
customer => preference => recommendation v2 from '2819441432': 13
customer => preference => recommendation v2 from '2819441432': 14
```

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

```
istioctl delete -f istiofiles/recommendation_lb_policy_app.yml \
-n tutorial
```

Timeout

Timeouts are a crucial component to 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 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 take a look at your *recommendation* service, find the `RecommendationVer ticle.java` class and uncomment the line that introduces a delay in the service. You should save your changes before continuing:

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

  HealthCheckHandler hc = HealthCheckHandler.create(vertx);
  hc.register("dummy-health-check", future ->
         future.complete(Status.OK()));
  router.get("/health").handler(hc);
```

```
  vertx.createHttpServer().requestHandler(router::accept).listen(8080);
}
```

You can now build the service and deploy it:

```
cd recommendation
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 Docker 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 '751265691-qdznv': 2

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

Note that 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.

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

```
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
  name: recommendation-timeout
spec:
  destination:
    namespace: tutorial
    name: recommendation
  precedence: 1
  route:
  - labels:
      app: recommendation
  httpReqTimeout:
    simpleTimeout:
      timeout: 1s
```

You can now create this route rule:

```
istioctl create -f istiofiles/route-rule-recommendation-timeout.yml \
-n tutorial
```

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 this route rule:

```
istioctl delete routerule recommendation-timeout -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. You could do this in your Java code, but you're going to use Istio, instead. You're going to inject transient HTTP 503 errors into your call to *recommendation* service. We cover fault injection in more detail in Chapter 5, but for the moment, trust that installing the following route rule will introduce HTTP 503 errors:

```
istioctl create -f istiofiles/route-rule-recommendation-v2_503.yml \
-n tutorial
```

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

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

customer => preference => recommendation v2 from '2036617847': 190
customer => preference => recommendation v2 from '2036617847': 191
customer => preference => recommendation v2 from '2036617847': 192
customer => 503 preference => 503 fault filter abort
customer => preference => recommendation v2 from '2036617847': 193
customer => 503 preference => 503 fault filter abort
customer => preference => recommendation v2 from '2036617847': 194
customer => 503 preference => 503 fault filter abort
customer => preference => recommendation v2 from '2036617847': 195
customer => 503 preference => 503 fault filter abort
```

Let's take a look at a `RouteRule` that specifies your retry configuration:

```
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
  name: recommendation-v2-retry
spec:
  destination:
    namespace: tutorial
    name: recommendation
  precedence: 3
  route:
  - labels:
      version: v2
  httpReqRetries:
    simpleRetry:
      perTryTimeout: 2s
      attempts: 3
```

This rule sets your retry attempts to `3` and will use a `2s` timeout for each retry. The cumulative timeout is therefore six seconds plus the timeout of the original call. (To specify an overall timeout, see the previous section on timeouts.)

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

```
istioctl create -f istiofiles/route-rule-recommendation-v2_retry.yml \
-n tutorial
```

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

```
customer => preference => recommendation v2 from '751265691-n65j9': 35
customer => preference => recommendation v2 from '751265691-n65j9': 36
customer => preference => recommendation v2 from '751265691-n65j9': 37
customer => preference => recommendation v2 from '751265691-n65j9': 38
customer => preference => recommendation v2 from '751265691-n65j9': 39
customer => preference => recommendation v2 from '751265691-n65j9': 40
customer => preference => recommendation v2 from '751265691-n65j9': 41
customer => preference => recommendation v2 from '751265691-n65j9': 42
customer => preference => recommendation v2 from '751265691-n65j9': 43
```

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

```
oc delete routerule --all
```

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 insures that any specific appliance does not overdraw electrical current through a particular outlet. If you ever lived with someone who plugged in 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*. The book was first published in 2007, long before the term microservices was even coined. A second edition of the book was just released in 2018.

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 embed 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 workflows, to the birth of Netflix OSS and the coining of the term "microservices."

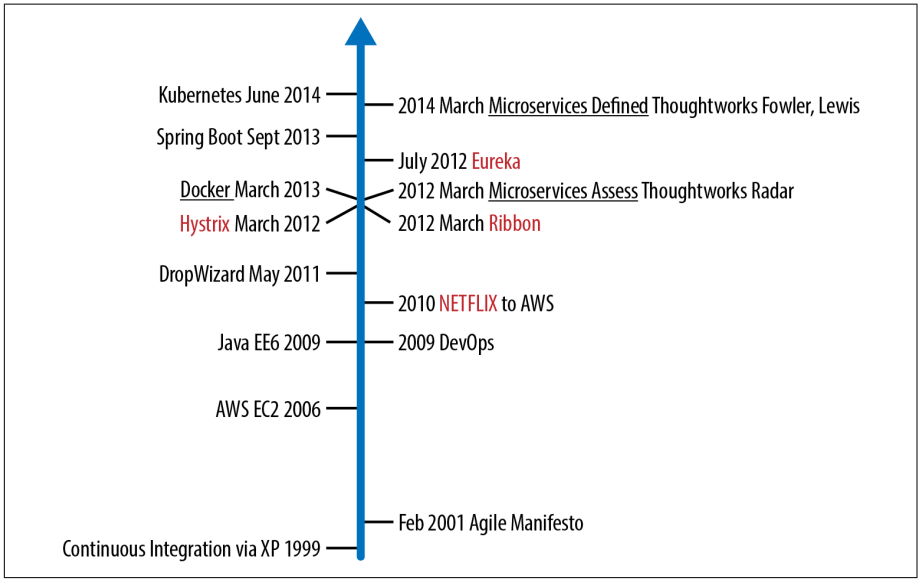

Figure 4-1. Microservices timeline

Istio puts more of the resilience implementation into the infrastructure so that you can focus more of their valuable time and energy on code that differentiates their 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 *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);
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 `RouteRule`:

```
istioctl create -f \
istiofiles/route-rule-recommendation-v1_and_v2_50_50.yml -n tutorial
```

From the initial installation instructions, we recommended you install the `seige` 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 of the requests to your system were successful, but it took some time to run the test because the v2 instance or pod was a slow performer. Note that for each call to v2, it took three seconds or more to complete (this is from the delay functionality you enabled).

But suppose that in a production system this three-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 `DestinationPolicy` that looks like this:

```
apiVersion: config.istio.io/v1alpha2
kind: DestinationPolicy
metadata:
  name: recommendation-circuitbreaker
spec:
  destination:
    namespace: tutorial
    name: recommendation
    labels:
      version: v2
  circuitBreaker:
    simpleCb:
      maxConnections: 1
      httpMaxPendingRequests: 1
      sleepWindow: 2m
      httpDetectionInterval: 1s
      httpMaxEjectionPercent: 100
      httpConsecutiveErrors: 1
      httpMaxRequestsPerConnection: 1
```

Here, you're configuring the circuit breaker for any client calling into v2 of the *recommendation* service. Remember in the previous `RouteRule` that you are splitting (50%) traffic between both v1 and v2, so this `DestinationPolicy` 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 the next section, in which we look at outlier detection.) Let's create this circuit breaker policy:

```
istioctl create -f istiofiles/recommendation_cb_policy_version_v2.yml \
-n tutorial
```

Now try the `seige` 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).

You can clean up these destination policies and route rules like this:

```
istioctl delete routerule recommendation-v1-v2 -n tutorial
istioctl delete -f istiofiles/recommendation_cb_policy_version_v2.yml
```

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. Because the Istio proxy is based on Envoy and Envoy calls this implementation *outlier detection*, we'll use the same terminology for discussing Istio.

Pool ejection or outlier detection is a resilience strategy that takes place whenever you have a pool of instances or pods 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 `RouteRule` in place. Let's use a 50/50 split of traffic:

```
oc create -f istiofiles/route-rule-recommendation-v1_and_v2_50_50.yml \
-n tutorial
```

Next, you can scale the number of pods for the v2 deployment of *recommendation* so that you have some hosts 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. You can watch their progress with the following:

```
oc get pods -w
```

Now, let's generate some simple load 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 => preference => recommendation v1 from '2039379827': 447
customer => preference => recommendation v2 from '2036617847': 26
customer => preference => recommendation v1 from '2039379827': 448
customer => preference => recommendation v2 from '2036617847': 27
```

```
customer => preference => recommendation v1 from '2039379827': 449
customer => preference => recommendation v1 from '2039379827': 450
customer => preference => recommendation v2 from '2036617847': 28
customer => preference => recommendation v1 from '2039379827': 451
customer => preference => recommendation v1 from '2039379827': 452
customer => preference => recommendation v2 from '2036617847': 29
customer => preference => recommendation v2 from '2036617847': 30
customer => preference => recommendation v2 from '2036617847': 216
```

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

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

You should see something like this:

```
recommendation-v2-2036617847         2/2       Running   0          1h
recommendation-v2-2036617847-spdrb   2/2       Running   0          7m
```

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

```
oc exec -it recommendation-v2-2036617847-spdrb -c recommendation /bin/bash
```

You will be inside the application container of your pod `recommendation-v2-2036617847-spdrb`. Now execute:

```
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-v2-2036617847-spdrb` receives a request, you get a 503 error:

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

Now let's see what happens when you configure Istio to eject misbehaving hosts. Take a look at the `DestinationPolicy` in the following:

```
istiofiles/recommendation_cb_policy_pool_ejection.yml

apiVersion: config.istio.io/v1alpha2
kind: DestinationPolicy
metadata:
  name: recommendation-poolejector-v2
  namespace: tutorial
spec:
  destination:
    namespace: tutorial
    name: recommendation
    labels:
      version: v2
  loadBalancing:
    name: RANDOM
  circuitBreaker:
    simpleCb:
      httpConsecutiveErrors: 1
      sleepWindow: 15s
      httpDetectionInterval: 5s
      httpMaxEjectionPercent: 100
```

In this `DestinationPolicy`, you're configuring Istio to check every five seconds for misbehaving hosts and to remove hosts from the load balancing pool after one consecutive error (artificially low for this example). You are willing to eject up to 100% of the hosts (effectively temporarily suspending any traffic to the cluster).

```
istioctl create -f istiofiles/recommendation_cb_policy_pool_ejection.yml \
-n tutorial
```

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-v2-2036617847-spdrb`, it is ejected from the pool and it doesn't receive any more requests until the sleep window expires—which takes at least 15 seconds.

```
customer => preference => recommendation v1 from '2039379827': 509
customer => 503 preference => 503 recommendation misbehavior from
'2036617847'
customer => preference => recommendation v1 from '2039379827': 510
customer => preference => recommendation v1 from '2039379827': 511
customer => preference => recommendation v1 from '2039379827': 512
```

```
customer => preference => recommendation v1 from '2039379827': 513
customer => preference => recommendation v1 from '2039379827': 514
customer => preference => recommendation v2 from '2036617847': 256
customer => preference => recommendation v2 from '2036617847': 257
customer => preference => recommendation v1 from '2039379827': 515
customer => preference => recommendation v2 from '2036617847': 258
customer => preference => recommendation v2 from '2036617847': 259
customer => preference => recommendation v2 from '2036617847': 260
customer => preference => recommendation v1 from '2039379827': 516
customer => preference => recommendation v1 from '2039379827': 517
customer => preference => recommendation v1 from '2039379827': 518
customer => 503 preference => 503 recommendation misbehavior from
'2036617847'
customer => preference => recommendation v1 from '2039379827': 519
customer => preference => recommendation v1 from '2039379827': 520
customer => preference => recommendation v1 from '2039379827': 521
customer => preference => recommendation v2 from '2036617847': 261
customer => preference => recommendation v2 from '2036617847': 262
customer => preference => recommendation v2 from '2036617847': 263
customer => preference => recommendation v1 from '2039379827': 522
customer => preference => recommendation v1 from '2039379827': 523
customer => preference => recommendation v2 from '2036617847': 264
customer => preference => recommendation v1 from '2039379827': 524
customer => preference => recommendation v1 from '2039379827': 525
customer => preference => recommendation v1 from '2039379827': 526
customer => preference => recommendation v1 from '2039379827': 527
customer => preference => recommendation v2 from '2036617847': 265
customer => preference => recommendation v2 from '2036617847': 266
customer => preference => recommendation v1 from '2039379827': 528
customer => preference => recommendation v2 from '2036617847': 267
customer => preference => recommendation v2 from '2036617847': 268
customer => preference => recommendation v2 from '2036617847': 269
customer => 503 preference => 503 recommendation misbehavior
from '2036617847'
customer => preference => recommendation v1 from '2039379827': 529
customer => preference => recommendation v2 from '2036617847': 270
```

Combination: Circuit-Breaker + Pool Ejection + Retry

Even with pool ejection your application doesn't look that resilient. That's probably because you're still letting some errors to be propagated to your clients. But you can improve this. If you have enough instances or versions of a specific service running into 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 `RouteRule`, we are able to completely get rid of our 503s requests. This means that whenever you receive a failed request from an ejected instance, Istio will forward the request to another supposedly healthy instance:

```
istioctl replace -f istiofiles/route-rule-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, but the requests from recommendation v2 are still taking more time to get a response:

```
customer => preference => recommendation v1 from '2039379827': 538
customer => preference => recommendation v1 from '2039379827': 539
customer => preference => recommendation v1 from '2039379827': 540
customer => preference => recommendation v2 from '2036617847': 281
customer => preference => recommendation v1 from '2039379827': 541
customer => preference => recommendation v2 from '2036617847': 282
customer => preference => recommendation v1 from '2039379827': 542
customer => preference => recommendation v1 from '2039379827': 543
customer => preference => recommendation v1 from '2039379827': 544
customer => preference => recommendation v2 from '2036617847': 283
customer => preference => recommendation v2 from '2036617847': 284
customer => preference => recommendation v1 from '2039379827': 545
customer => preference => recommendation v1 from '2039379827': 546
customer => preference => recommendation v1 from '2039379827': 547
customer => preference => recommendation v2 from '2036617847': 285
customer => preference => recommendation v2 from '2036617847': 286
customer => preference => recommendation v1 from '2039379827': 548
customer => preference => recommendation v2 from '2036617847': 287
customer => preference => recommendation v2 from '2036617847': 288
customer => preference => recommendation v1 from '2039379827': 549
customer => preference => recommendation v2 from '2036617847': 289
customer => preference => recommendation v2 from '2036617847': 290
customer => preference => recommendation v2 from '2036617847': 291
customer => preference => recommendation v2 from '2036617847': 292
customer => preference => recommendation v1 from '2039379827': 550
customer => preference => recommendation v1 from '2039379827': 551
customer => preference => recommendation v1 from '2039379827': 552
customer => preference => recommendation v1 from '2039379827': 553
customer => preference => recommendation v2 from '2036617847': 293
customer => preference => recommendation v2 from '2036617847': 294
customer => preference => recommendation v1 from '2039379827': 554
```

Your misbehaving pod `recommendation-v2-2036617847-spdrb` never shows up in the console, thanks to pool ejection and retry.

Clean up (note, we'll leave the route rules in place as those will be used in the next chapter):

```
oc scale deployment recommendation-v2 --replicas=1 -n tutorial
oc delete pod -l app=recommendation,version=v2
oc delete routerule recommendation-v1-v2 -n tutorial
istioctl delete -f istiofiles/recommendation_cb_policy_pool_ejection.yml
-n tutorial
```

CHAPTER 5

Chaos Testing

A relatively famous OSS project called *Chaos Monkey* came from the developer team at Netflix, and its unveiling to the IT world was quite disruptive. The concept that Netflix had built code that random kills various services in their production environment blew people's minds. When many teams struggle 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." (You can read more at *http://principlesofchaos.org/*).

In complex systems (software systems or ecological systems), things do and will fail, but the ultimate goal is 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

This simple concept allows you to explore your overall system's behavior when random faults pop up within the system. Throwing in some HTTP errors is actually very simple when using Istio's `RouteRule` construct. Based on previous exercises earlier in this book, recommendation v1 and v2 are both deployed and being randomly load balanced because that is the default behavior in Kubernetes/OpenShift. Make sure to comment out the "timeout" line if that was used in a

previous exercise. Now, you will be injecting errors and timeouts via Istio instead of using Java code:

```
oc get pods -l app=recommendation -n tutorial
NAME                                READY   STATUS   RESTARTS   AGE
recommendation-v1-3719512284-7mlzw  2/2     Running  6          18h
recommendation-v2-2815683430-vn77w  2/2     Running  0          3h
```

We use the Istio `RouteRule` to inject a percentage of faults, in this case, returning 50% HTTP 503's:

```
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
  name: recommendation-503
spec:
  destination:
    namespace: tutorial
    name: recommendation
  precedence: 2
  route:
  - labels:
      app: recommendation
  httpFault:
    abort:
      percent: 50
      httpStatus: 503
```

And you apply the `RouteRule` with the `istioctl` command-line tool:

```
istioctl create -f istiofiles/route-rule-recommendation-503.yml -n tutorial
```

Testing the change is as simple as issuing a few `curl` commands at the customer end point. 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 '99634814-sf4cl': 88

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

Clean up:

```
istioctl delete -f istiofiles/route-rule-recommendation-503.yml -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 important, if your service has a specific Service-Level Agreement (SLA) it must meet, how do you verify that slowness in your dependencies do not cause you to fail in delivery to your awaiting cus-

tomer? Injecting network delays allows you to see how the system behaves when a critical service or three simply adds notable extra time to a percentage of responses.

Much like the HTTP Fault injection, network delays use the `RouteRule` kind, as well. The following YAML injects seven seconds of delay into 50% of the responses from recommendation service:

```
apiVersion: config.istio.io/v1alpha2
kind: RouteRule
metadata:
  name: recommendation-delay
spec:
  destination:
    namespace: tutorial
    name: recommendation
  precedence: 2
  route:
  - labels:
      app: recommendation
  httpFault:
    delay:
      percent: 50
      fixedDelay: 7s
```

Use the `istioctl` create command to apply the new `RouteRule`:

```
istioctl create -f istiofiles/route-rule-recommendation-delay.yml \
-n tutorial
```

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 seven-second delay.

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

Notice that 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.

```
stern recommendation -n tutorial
```

Clean up:

```
istioctl delete -f istiofiles/route-rule-recommendation-delay.yml \
-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 or pods, and discovering where performance bottlenecks have occurred provides valuable information.

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 challenging 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 original open sourced by the Uber Technologies team and is a distributed tracing system specifically focused on microservices architecture.

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

You open the Jaeger console by using the following command:

```
minishift openshift service jaeger-query --in-browser
```

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

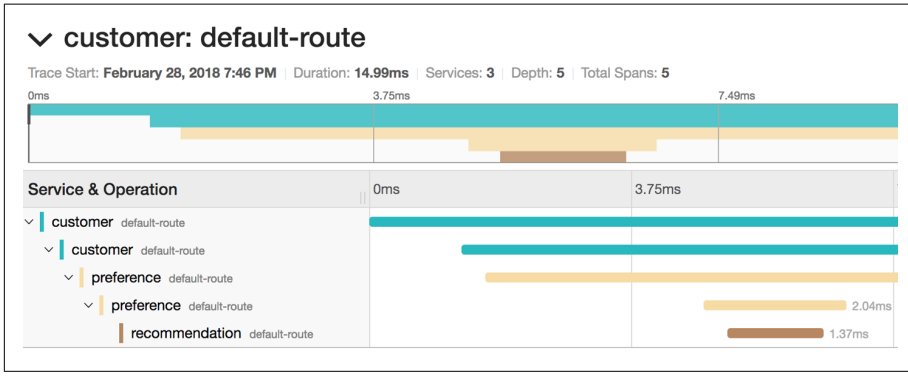

Figure 6-1. Jaeger's view of the customer-preference-recommendation trace

One aspect that is *important* to remember is that your programming logic must forward the OpenTracing headers with 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
```

You can see an example of this concept in the customer class called `HttpHeaderForwarderHandlerInterceptor` in the accompanying sample code.

Metrics

By default, Istio's default configuration will gather telemetry data across the service mesh. Simply installing Prometheus and Grafana is enough to get started with this important service, however do keep in mind many other backend metrics/telemetry-collection services are supported. In Chapter Chapter 2, you saw the following four commands to install and expose the metrics system:

```
oc apply -f install/kubernetes/addons/prometheus.yaml
oc apply -f install/kubernetes/addons/grafana.yaml
```

```
oc expose svc grafana
oc expose svc prometheus
```

You can then launch the Grafana console using the minishift service command:

```
open "$(minishift openshift service grafana -u)/dashboard/db/istio-
dashboard?var-source=All"
```

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

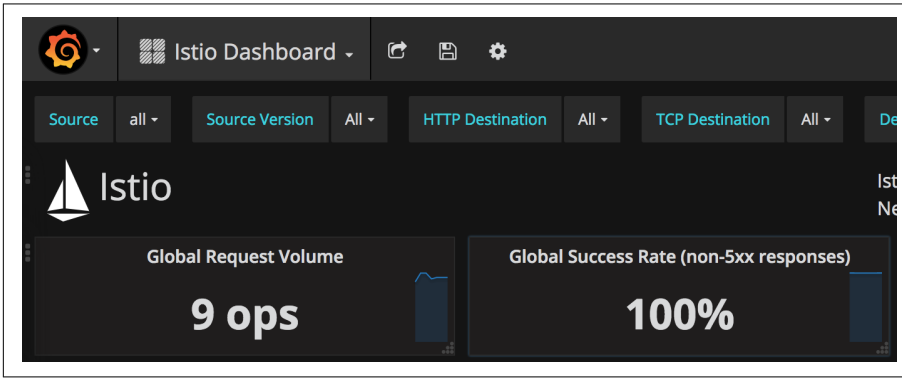

Figure 6-2. The Grafana dashboard—selecting Istio dashboard

As of this writing, you do need to append `?var-source=All` to the Grafana dashboard URL. This is likely to change in the future, watch the *istio-tutorial* for changes.

Here's an example URL:

http://grafana-istio-system.192.168.99.101.nip.io/dashboard/db/istio-dashboard?var-source=All

Figure 6-3 shows the dashboard. You can also visit the Prometheus dashboard directly at the following (note, this will open the URL in a browser for you; you could use `--url` instead of `--in-browser` to get just the URL):

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

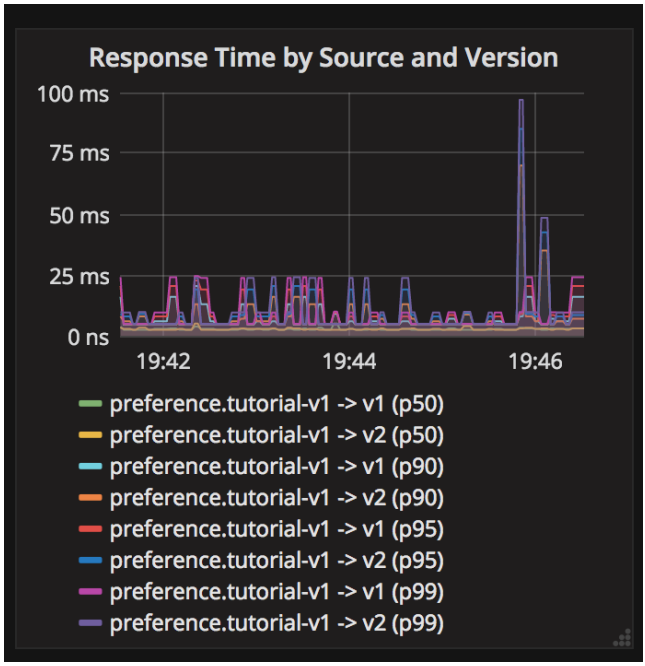

Figure 6-3. Grafana graph

CHAPTER 7
Security

Istio's security capabilities are evolving quickly, and as of this writing, the Access Control List (ACL) is one of the primary tools to inject security constructs into the application with zero impact to the actual programming logic. In this chapter, we explore the concepts of *Blacklist* and *Whitelist*.

Blacklist

Let's begin with the concept of the blacklist, conditionally denying requests using Mixer selectors. The blacklist is explicit denials of particular invocation paths. In the example that follows, we want to explicitly close the route from *customer* to *preference*. In this case, any requests from the *customer* to *preference* would return the HTTP error 403 Forbidden. Establishing this requires the use of three different kinds of Istio-mixer configurations: `denier`, `checknothing`, and `rule`:

```
apiVersion: "config.istio.io/v1alpha2"
kind: denier
metadata:
  name: denycustomerhandler
spec:
  status:
    code: 7
    message: Not allowed
---
apiVersion: "config.istio.io/v1alpha2"
kind: checknothing
metadata:
  name: denycustomerrequests
spec:
---
apiVersion: "config.istio.io/v1alpha2"
kind: rule
metadata:
```

```
  name: denycustomer
spec:
  match: destination.labels["app"] == "preference" &&
   source.labels["app"]=="customer"
  actions:
  - handler: denycustomerhandler.denier
    instances: [ denycustomerrequests.checknothing ]
```

You use `istioctl` to establish the denier-checknothing-rule:

```
istioctl create -f istiofiles/acl-blacklist.yml -n tutorial
```

Next, attempt to `curl` the customer endpoint:

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

The result will be a 403 from the preference microservice customer => 403 PERMISSION_DENIED:denycustomerhandler.denier.tutorial:Not allowed

Clean up:

```
istioctl delete -f istiofiles/acl-blacklist.yml -n tutorial
```

Whitelist

The whitelist is a deny everything rule, except for approved invocation paths. In this example, we are approving only the route of *recommendations* > *preferences* which means the customer who normally talks to *preference* can no longer even see it. The whitelist configuration uses the Mixer kinds of: `listchecker`, `listentry`, `rule`.

```
apiVersion: "config.istio.io/v1alpha2"
kind: listchecker
metadata:
  name: preferencewhitelist
spec:
  overrides: ["recommendation"]
  blacklist: false
---
apiVersion: "config.istio.io/v1alpha2"
kind: listentry
metadata:
  name: preferencesource
spec:
  value: source.labels["app"]
---
apiVersion: "config.istio.io/v1alpha2"
kind: rule
metadata:
  name: checkfromcustomer
spec:
  match: destination.labels["app"] == "preference"
  actions:
  - handler: preferencewhitelist.listchecker
```

```
    instances:
    - preferencesource.listentry
```

Using the `istioctl` tool, create the blacklist components:

```
istioctl create -f istiofiles/acl-whitelist.yml -n tutorial
```

Then, hit the customer end point using the `curl` command:

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

Which results in the following:

```
customer => 404 NOT_FOUND:preferencewhitelist.listchecker.tutorial:
customer is not whitelisted
```

Clean up:

```
istioctl delete -f istiofiles/acl-whitelist.yml -n tutorial
```

The Red Hat team will be exploring more interesting and advanced security use cases at the *istio-tutorial* as the Istio open source project matures.

Conclusion

You have now taken a 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, whether developing microservices architectures or monoliths or anything in between. You have seen how Istio concepts like observability, resiliency and chaos injection can be immediately beneficial to your current application. Although we focused on services running on Kubernetes/OpenShift and deployed in containers, Istio is not tied to any of these environments and can be used on bare metal, VM, and other deployment platforms.

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

- End-user authentication
- Policy enforcement
- Mesh expansion
- Hybrid deployments
- Phasing in Istio to an existing environment
- Gateway/Advanced ingress
- Latest evolution of `RouteRules`/resources

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 as well as Red Hat's evolving *istio-tutorial*.

About the Authors

Christian Posta (@christianposta) is a chief architect of cloud applications at Red Hat and well known in the community for being an author (*Microservices for Java Developers*, O'Reilly, 2016), frequent blogger, speaker, open source enthusiast, and committer on various open source projects, including Istio, Apache ActiveMQ, Fabric8, and others. Christian has spent time at web-scale companies and now helps organizations create and deploy large-scale, resilient, distributed architectures—many of what we now call microservices. He enjoys mentoring, training, and leading teams to be successful with distributed systems concepts, microservices, DevOps, and cloud-native application design.

Burr Sutter 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 bush of Kenya. You can find Burr on Twitter @burrsutter and via the web at *burrsutter.com*.