Istio
IN ACTION

Christian Posta
Rinor Maloku

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Istio in Action
Version 9

For more information on this and other Manning titles go to
manning.com

welcome

Thank you for purchasing the MEAP for *Istio in Action*! This book is intended for developers, architects, and operators of cloud-native applications and has something for everyone. Building and sustaining cloud-native applications is a tall order that ends up eliminating a lot of nice assumptions we had in the past and moving a lot of complexity between the applications. Istio is an open-source project that aims to help folks connect and manage their services and applications by solving for some difficult problems like network resilience, security, traffic management, observability and policy enforcement.

Because Istio brings so much to the table, it can be overwhelming at first. Having a firm understanding of what components make up Istio, their respective roles, and how Istio accomplishes what it sets out to do will help you feel more comfortable when adopting this technology. Additionally, Istio can (and should) be adopted incrementally, so understanding how it all works will help you decide which parts will give you the biggest value up front.

In these first few chapters, you'll explore what forces might compel you to use technology like Istio, discover what this "service mesh" thing really is, and start to get hands on with Istio's capabilities. If you have any questions or comments, please post them in the book's forum or in the book's source code repository . We'd especially like feedback on what areas could be made more clear, and what concepts might be missing.

Again, thank you for purchasing this book. We look forward to hearing your thoughts!

— Christian Posta (@christianposta)

— Rinor Maloku (@rinormaloku)

brief contents

1 Introducing Istio Service Mesh

This chapter covers:

- The need to cheaply run experiments and learn faster
- Service communication has many challenges (reliability, control, security, observability)
- How a service mesh can help solve service-communication challenges
- What Istio is, and how it helps solve microservices challenges

Software is the lifeblood of today's companies. As we move to a more digital world, consumers will expect convenience, service, and quality when interacting with these companies and software will be used to deliver these experiences. Customers don't conform nicely to structure, processes, or pre-defined boxes. Customer's demands and needs are fluid, dynamic and unpredictable. For this reason, our companies and software systems will need to have these same characteristics. For some companies (e.g., startups), building software systems that are agile and able to respond to unpredictability will be the difference between survival or not. For others (e.g., existing companies), the inability to use software as a differentiator will mean slower growth, decay, and eventual disruption.

Trends in software architecture recently have been oriented around scaling teams and technology to account for this agility. Cloud has changed the way we consume infrastructure as well as changed some assumptions about how we build applications. As we'll see in the next section, topics like Microservices, DevOps and Cloud are all exciting possibilities and are highly related in that they promise the nirvana of going faster and enabling agility. With any new methodologies or shift in technology that promise great gains, however, you should always expect new challenges to arise.

As we explore how to go faster and take advantage of newer technology like cloud platforms and

containers, we'll find that we encounter an amplification of some past problems, some problems become harder, and we acknowledge that some problems appear impossible to solve. For example, more recently we've acknowledged our applications cannot ignore the network by building resilience into our application. As we start to build larger, more distributed systems, the network must become a central design consideration in our applications. Should applications themselves implement resilience concerns like retries, timeouts, and circuit breakers? Should each application be left to implement these critical pieces of functionality on their own in hopes of correctly implemented resilience?

Additionally, we've known for a while that metric and log collection are vital signals we must collect and use to understand our systems. As our systems become more complex and deployed on elastic, ephemeral, cloud infrastructure, how can we instrument and monitor our system in hopes of understanding it at run time and in real time? Again, should each application be left to figure out how best to instrument and expose its basic signals? Are there basic telemetry indicators that would benefit system operators that can done outside the application?

Basic resilience and metric collection are horizontal concerns and are not application specific. Moreover, these are not differentiating business concerns. Developers are critical resources in a large IT systems and their time is best spent writing capabilities that deliver business value in a differentiating way. Application networking and metrics are a necessary practice, but they aren't differentiating. What we'd like is a way to implement these capabilities in a language and framework agnostic way and expose them as "application infrastructure services" that can be consumed without any special application considerations. Any application with any developers should be able to properly build cloud-native applications without worrying about what nasty bugs they may have introduced because of the network. Application resilience, observability, traffic control, security, and policy all fall into the category of "horizontal concerns" that should be implemented in supporting infrastructure not the application itself.

"Service mesh" is a relatively recent term (though not invented recently) used to describe the decentralized application-networking infrastructure that allows applications to be secure, resilient, observable and controllable. It describes an architecture made up of a data plane which uses application-level proxies to manage the networking traffic and a control plane to manage the proxies. This architecture allows us to build these important capabilities outside of the application with little to no intervention from the application.

Istio is an open-source implementation of a service mesh. Istio was created initially by folks at Lyft, Google, and IBM but now has participants from other companies such as Red Hat, Tigera, and many others in the broader community. Istio allows us to build reliable, secure, cloud-native systems and solve for some of the difficult problems that arise like security, policy management, and observability with minimal (and in some cases, no) application code changes. Istio's data

plane is made up of service proxies based on Envoy proxy, and its control plane implements the APIs to manage and configure the proxies. The service proxies live alongside the applications and are used to effect networking behavior.

Istio is intended for microservices or SOA-style architectures, but is not limited to those. The reality is, most organizations have a lot of investment in existing applications and platforms. They'll most likely build services architectures around their existing applications and this is where Istio really shines. With Istio we can implement these application-networking concerns without forcing changes in existing systems. Since the service proxies live outside of the application, any application for any architecture is a welcomed first class citizen in the service mesh. We'll explore more of this in a hybrid "brownfield" application landscape.

This book will introduce you to Istio, help you understand how all of this is possible, and teach you how to use Istio to build more resilient applications that you can monitor and operate in a cloud environment. Along the way we'll explore Istio's design principles, explain why it's different from past attempts to solve these problems, as well as when Istio is not the solution for your problem.

But we certainly don't want to start using new technology just because it's "new", "hip", or "cool". As technologists, we find ourselves easily getting excited about technology, however, we'd be doing ourselves and our organizations a disservice by not fully understanding when and when not to use a technology. Let's spend a few moments understanding why you would use Istio, what problems it solves, what problems to avoid, and why this technology is exciting going forward.

1.1 Optimize to go faster, safely

Going faster in our enterprise companies is rarely a function of technology alone. IT has historically been seen as an operational burden not a source of business differentiation. In an effort to optimize to control cost, IT organizational boundaries, cultural norms, process and procedure have been defined in such a way that manifested silos, low-trust working environments, increasing levels of coordination, and stifling governance. For example, at ACME company, no work can be done without a funded project. Before development can start, the business folks at ACME get together and decide what the top priorities for the company should be. They convene for weeks if not months to devise swanky-looking power-point slides with proposed cost and ROI numbers to justify why their respective ideas are the best path forward for ACME. Often times these project proposals are packed to the gills with features to justify their high costs and long proposed delivery time frames. Once a project gets approved, the designers and stakeholders devise a long list of requirements they believe can be handed to the next group for implementation. This can take months. At some point from here, we see requirement/user-story prioritization, developer implementation, quality assurance testing and finally operationalization for production. ACME usually ends up struggling to deliver their projects on time and on budget.

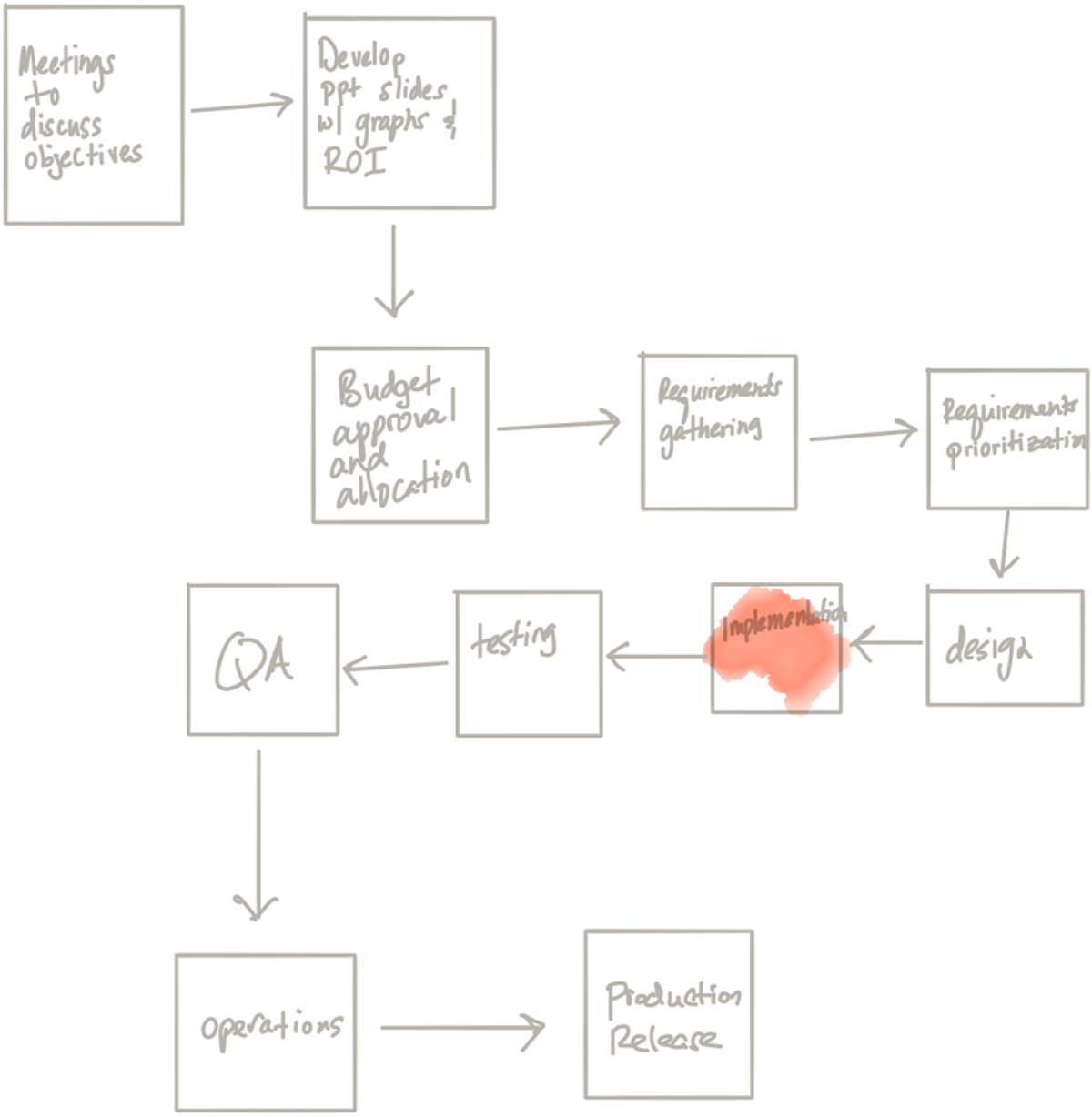


Figure 1.1 ACME project value stream

As you can see there are many steps in the value stream which influence the lead time of delivering changes to customers. Technology is part of the equation, but simply which technology framework you pick doesn't automatically make this process more agile, nor does it reduce your lead time. The system itself is not optimized to go fast. The DevOps and Lean thinking communities aim to tackle the problems around optimizing the system for agility and delivering value faster.

One of the core issues of the previous example is that we spend an incredible amount of money and effort before we see any value. In fact, our entire project is based around a theoretical "proposal" so even the promise of "value" is in doubt. Even if IT were to deliver a project on time and on budget, there's still a decent chance the proposed project doesn't delivery any value. One of the core tenants of Agile development, DevOps, and Lean thinking is we live in a world

of high uncertainty and that our efforts to try and precisely predict the future about what brings value will be in vain. We should take a scientific-method approach by treating our proposed "good ideas" as hypothesis that need to be disproved or proved by running experiments. By running experiments we can reduce uncertainty about what delivers value.

Coming back to IT and our technology systems, we want to create an IT engine that supports teams quickly running cheap experiments and learning what provides value to the business and to its customers. Unfortunately, even customers themselves may not know exactly what they want. Asking customers what they wanted would have produced faster horses and phones with physical keyboards and styluses. What we want to do with our experiments is put things out in front of customers and see how they react. Turns out smart phones with no buttons and cars are what people wanted. We want to more deeply understand what problems customers are trying to solve, and give them ideas about what might be a good way to solve them. Of course, we need the ability to get feedback from these customers. We need a way to observe the impact and consequences of our experiments so that we can learn, create new hypotheses and ultimately build valuable products.

The best way for us to test our hypotheses is with real customers. This means being able to do code releases to production as quickly and safely as possible. Microservices architectures, automated testing, containers, and continuous integration/continuous delivery all aim to help us go faster, but once we get into the concrete details, some new problems crop up. Let’s take a closer look.

1.1.1 Microservices and APIs to build large systems

An aspect to building our system to be agile and quicker to respond to change is to address the complexities of traditional application architectures and how they *devolve* over time. In the not too distant past, it was very common to build a large system out of highly coupled, co-located components that ran in a single application server. Moreover, as the feature requests piled up, management would respond by adding more developers to a team. As the team grew, and features were hacked in, the structure of the code and architecture devolved into a "big ball of mud". This made changes to the application even more precarious. Many teams were trying to work on the same code base, typically by long-lived developer branches. Re-integration of these branches were highly unpredictable and usually done under the stress of expiring deadlines. Testing was manual, needed coordination, and took a long time. Deployments of the big ball of mud were stop-the-world, "big-bang", high-risk scenarios.

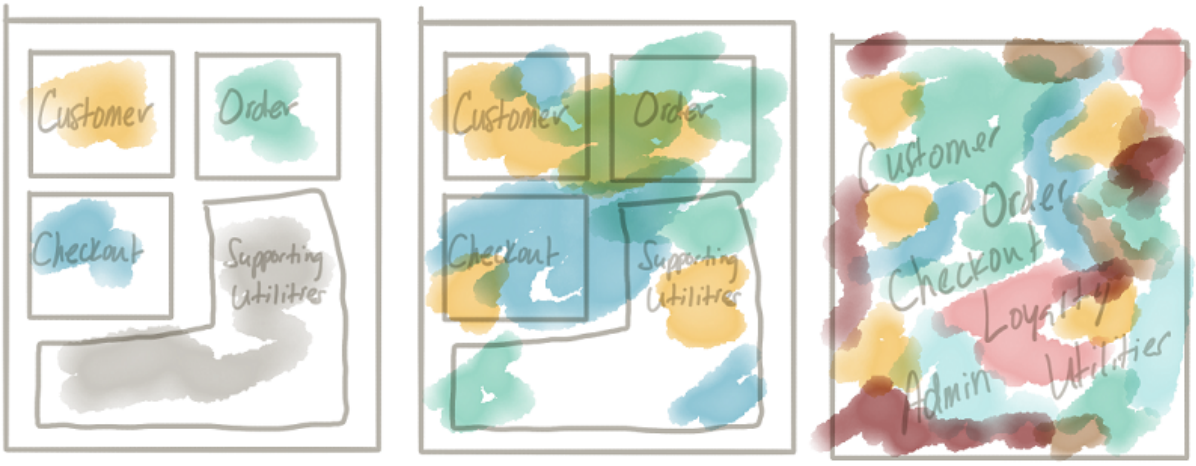


Figure 1.2 Monolith decay to big ball of mud

The service oriented architecture movement from the early 2000s promised a way to save these monolithic architectures by focusing on modularity and **services** being the primary artifact. Services would be built with strong contracts that decoupled implementation details and focus on reusability. These principles were sound, however, the implementation of them resembled the same highly centralized, highly governed, highly siloed properties of the rest of the organization. Conway's law was working it's magic, though not in the positive ways.

NOTE **Conway's law**

Conway's Law is an observation made by Melvin Conway about how organizations structure themselves and how their communication patterns are reflected in the systems they build. He's quoted as saying

"organizations which design systems … are constrained to produce designs which are copies of the communication structures of these organizations."

What we saw with SOA can be explained by Conway's law. The IT organizations needed "integration" capabilities so they created an "integration team". They wanted to control what changed to the system so they created a "governance team". They ended up with more silos that both created bottlenecks as well as gave them the authority to crush ideas for changes.

Although the thinking around SOA was sound, we found in practice it was implemented in a way that didn't take advantage of its strengths. We wanted to get away from long-lived code changes, big-bang releases, etc to a model that allows us to deploy faster and test our hypotheses. Large internet organizations like Google and Amazon implemented a "service-oriented" architecture, but they notably didn't look like that which emerged in the enterprise companies. These "WebOps" shops built a lot of the infrastructure and tooling to support their service oriented architecture on top of unreliable, yet elastic cloud infrastructure. They focused on services with smaller surface areas, solving for distributed-systems concerns up front, and building a lot of

trusted patterns into their respective platforms. Others had great success with this model, including Netflix. At some point in 2012 Martin Fowler and James Lewis help coin a new term for these SOA environments that they observed and that seemed to be working. They called it "microservices" in an effort to differentiate the existing SOA and ESB implementations that took hold in most organizations. Microservices seemed like a good way to deconstruct applications into smaller services so each team could work on its own service without impacting the overall system, and deliver code changes at their own cadence. This would facilitate smaller, more frequent, deployments to gauge customer feedback.

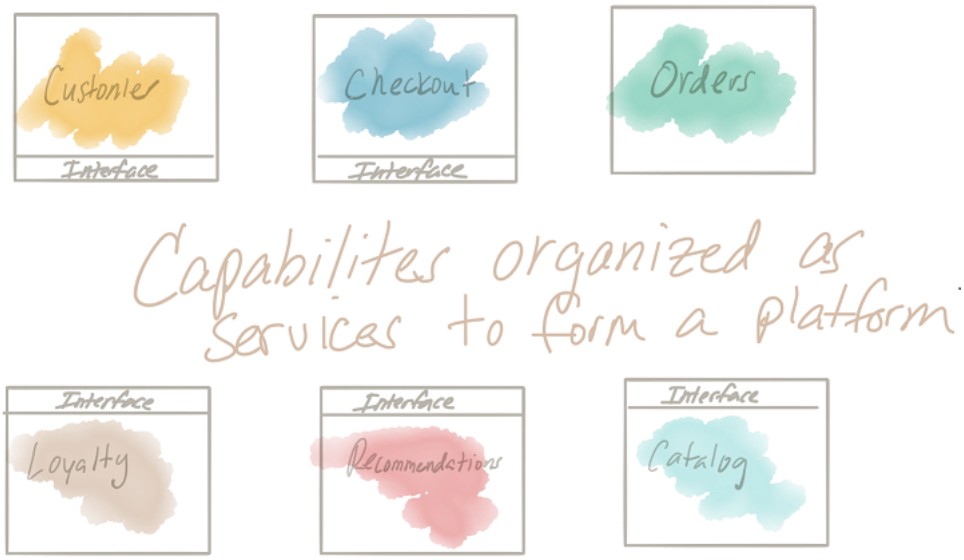


Figure 1.3 Capabilities organized as services

Microservices architecture is a way of working that decouples large systems into constituent smaller-scoped services that work together to implement business functionality. They are limited in size and scope for a very specific reason: to facilitate independent deployment for the purpose of making changes without impacting the rest of the system. Microservices is only possible if you have strong automation and delivery practices, similar to what you'd find in mature self-service platforms as discussed in the next sections. Microservices, cloud platforms and DevOps practices all work hand-in-hand to enable faster software delivery to production and shorter lead times. This approach helps to make it economical to practice the scientific method as a discipline for finding customer value. Imagine being able to deliver software changes to your system hundreds of times per day vs once a quarter. Your ability to experiment and learn at a rapid rate will be the differentiator you have compared to competition. Istio can help with controlled traffic rollouts, inter-service resilience, and metrics collection which are all higly necessary when makings lots of changes and trying to understand what impact they have.

1.1.2 Automated testing

Learning is all about getting feedback and improving our ability to reach a goal. Automated testing allows us to bring important parts of the feedback loop closer to our development efforts. When we make changes to our code we can set up regression suites to catch any bugs or unexpected deviations from intended features. Although unit tests are critical, a strong focus for automated testing should be scenario or feature tests. These types of tests force us to think about the API of the service and also allow us to get a glimpse of how our code will be used as our tests end up being consumers of our API. A strong focus on feature tests will also assert the behavior our customers expect, not just the inner workings of the code. Lastly, as implementation details change because of refactoring, paying down technical debt, maintenance, etc. our feature tests should rarely have to change.

Testing is also not relegated to pre-production testing with pre-defined tests. We should strive to have safe, low-risk ways to bring code changes to production and to explore the nooks and crannies of our systems that pre-defined tests would miss. Testing in production is a way to understand whether your code changes will run as expected. Exploratory testing allows you to inject chaos into the system and observe its behavior in a controlled way. It’s quite difficult to reproduce an exact copy of a production environment (with its configuration, infrastructure, networking, etc) in staging or QA environments. Tests passing in those lower environments may give a false sense of confidence when bringing changes to production. The only way to know that your changes are ready for production is to run them in production and observe its behavior. There are ways to do this with Istio while containing your deployment and any negative consequences and we’ll be exploring those in the next chapters.

1.1.3 Containers

Linux containers have contributed a massive shift in how we build, package, and run our applications. Container technology like Docker makes it easy for developers to take their applications, configuration, and dependencies and neatly package it up into an "image" that can then be run in "containers". Containers run your application as simple processes on a host that’s isolated by built-in linux primitives.

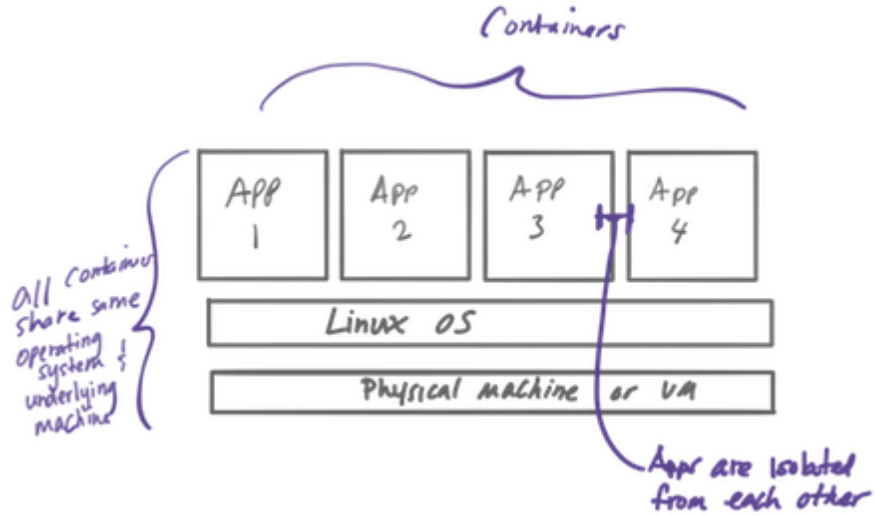


Figure 1.4 Containers share a operating-system kernel and are isolated by operating-system controls

In the past we may have done this with VM images and VMs, however containers allow us to package just the necessary bits and share the underlying linux Kernel. They're much lighter weight and also bring hardware density benefits. Once in containers, we can safely move applications between environments and eliminate configuration and environment drift. This gives us much more confidence when we deploy our services.

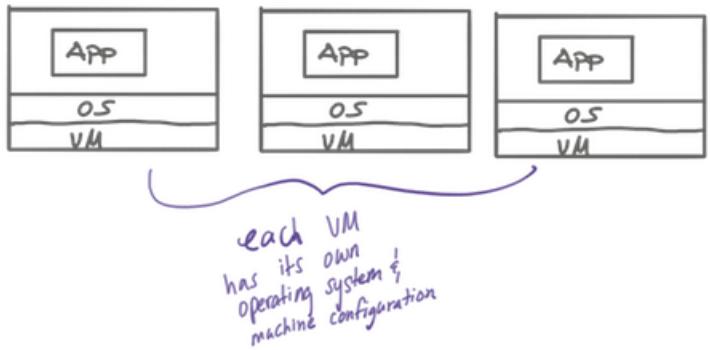


Figure 1.5 Each VM has its own operating system and virtualized hardware

Since containers provide a simple, uniform API for starting, stopping, inspecting, applications, we can build generic tools to run these applications on any kind of infrastructure that's capable of running Linux. Your service operator and deployment tools no longer have to be hand crafted for specific languages and their idiosyncracies. For example, Kubernetes is a leading container deployment platform that wraps container-deployment with a higher-level application API allowing the ability to deploy and manage containers across a cluster of machines with sophisticated orchestration, placement, and security considerations. Kubernetes has constructs like "Deployment" that ensure a minimum number of instances are deployed for a particular service and also actively performs health checking to make sure each instance is running correctly. Kubernetes also has constructs for "Services" which build simple load balancing and

service discovery into the platform which is usable by any application regardless of what implementation. We will explore this concept further when we dig into Istio.

1.1.4 Continuous integration and Continuous Delivery

In the previous section, we focused on getting feedback during the development cycle with automated testing. To be successful with a cloud-native or microservices architecture, we need to automate as much as possible the mechanisms for building and deploying code changes into production. Continuous Integration is a practice that gives developers feedback as quickly as possible by forcing code changes to be integrated as quickly as possible, ideally at least once a day. This means code changes from all team members are committed to source control, built, and tested to verify the application is still stable and in a position to be released if needed. Continuous delivery builds on continuous integration (CI and CD) by giving teams an automated pipeline to deploy their application into a new environment. Continuous delivery orchestrates the steps needed to successfully take an application from code commit to production. We should reduce the complexity inherent in any attempts to automate deployments, which is why containers and container platforms are excellent choices on which to build continuous delivery pipelines. Containers package our applications, along with their dependencies, to reduce configuration drift and unexpected environment settings so we can be more confident when we deliver our applications.

Although deployment is a key consideration to a CI/CD platform, as we'll see, traffic control and routing are also important. What happens when we bring a new deployment to an environment, especially production? We cannot just take down the old version and bring up the new version that would mean an outage. We also cannot replace them in place (unless we're magicians). What we want is a way to control the rollout of a new release by selectively bringing traffic to the deployment. To do this, we need the ability to control the traffic, which is what Istio gives us. With Istio we can finely control traffic to new deployments and reduce the risk of doing deployments. As we aspire to do deployments quickly we should also lower the risks of doing those deployments. Istio helps with that.

1.2 Challenges of going faster

The technology teams at ACME company did buy into microservices, automated testing, containers, and CI/CD. They decided to split out module A and B from ACMEmono, their core revenue generation system, into their own standalone services. They also needed some new capabilities that they decided to build as service C. They packaged their new services in containers and used a Kubernetes-based platform into which to deploy. As they began to implement these approaches, they found some inexorable challenges.

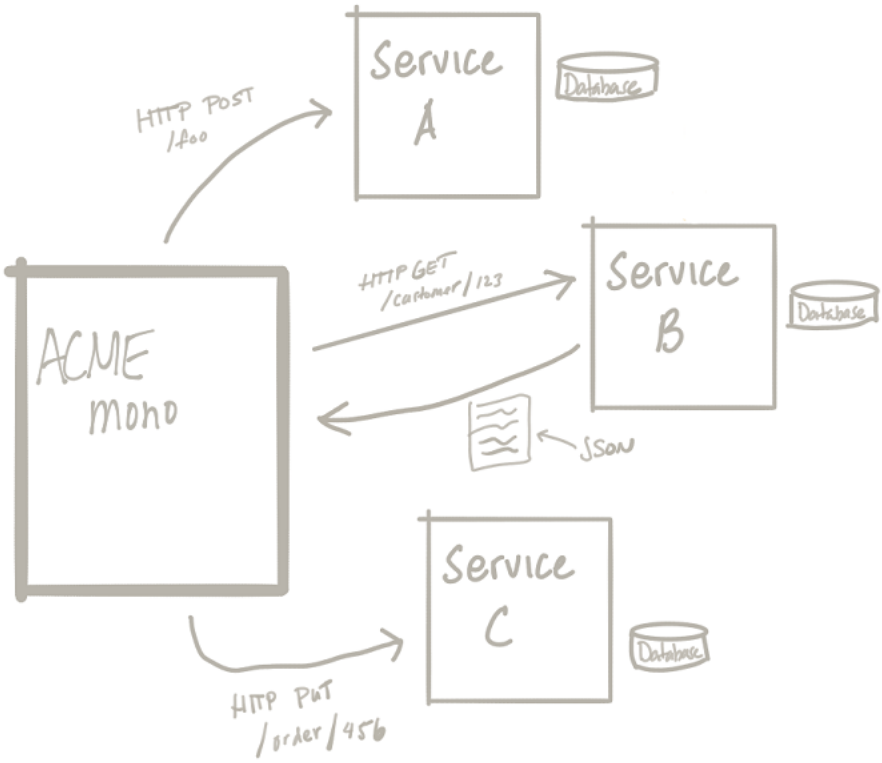


Figure 1.6 ACMEMono modernization with complementary services

The first thing ACME company noticed is that sometimes services in the architecture were very inconsistent in how long it took to process requests. At some points during the peak of customer usage, they noticed some services experienced intermittent issues and were unable to service any traffic. Furthermore they identified that if service B was experiencing some trouble processing requests that for some reason service A also did, but only for certain requests.

The second thing they noticed is that, as they practiced automated deployments, at times they introduced bugs into the system that weren't caught by automated testing. They practiced a deployment approach called "blue-green" deployment which means they brought up the new deployment (the "green" deployment) in its own cluster, and then at some point cut over the traffic from the old cluster (the "blue" deployment) to the new cluster. They had hoped blue-green deployments would lower their risk of doing deployments, but experienced more of a "big-bang" release which is what they wanted to avoid.

Lastly, ACME company found that the service teams implementing service A and B were handling security completely differently. Team A favored secure connections with certificates and private keys, while service B created their own custom framework built on servlet interceptors. The team operating service C decided they didn't need any additional security since these were "internal" services behind the company firewall.

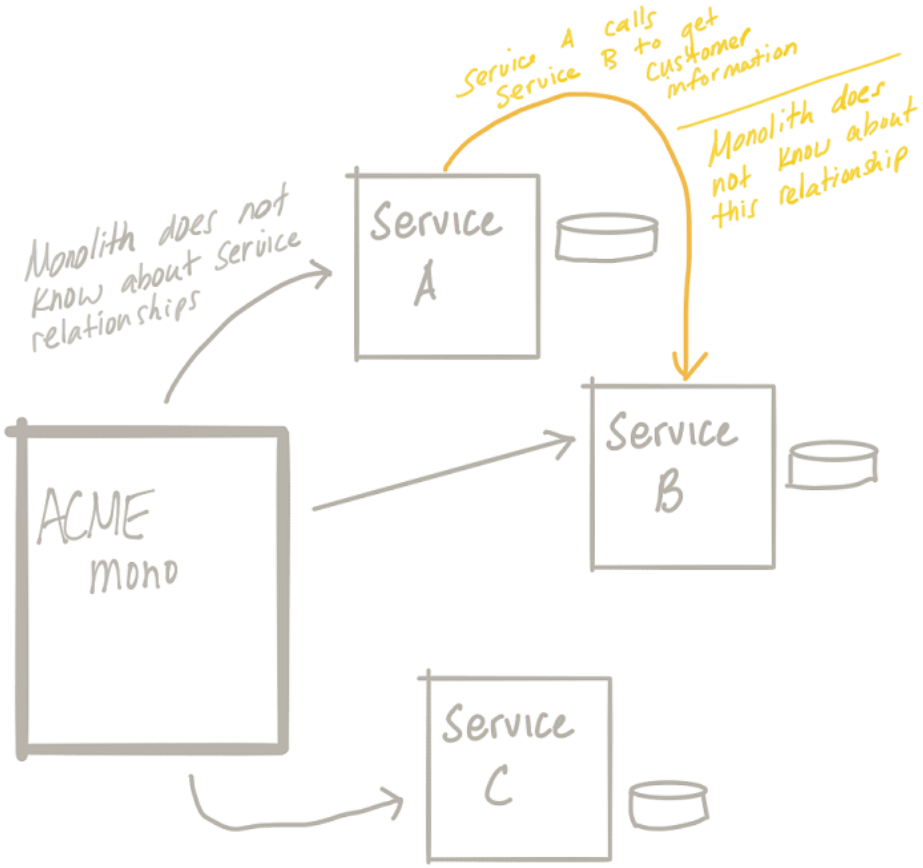


Figure 1.7 ACMEMono modernization service A calling service B

ACME company experienced problems when moving to an architecture and methodology that enables them to go faster. These challenges are not unique to ACME company, nor are the extent of the challenges limited to what they encountered. Things they have to overcome when moving to a services architecture:

- Keeping faults from jumping isolation boundaries
- Building applications/services capable of responding to changes in their environment
- Building systems capable of running in partially failed conditions
- Understanding what's happening to the overall system as it constantly changes and evolves
- Inability to control the runtime behaviors of the system
- Implementing strong security as the attack surface grows
- How to lower the risk of making changes to the system
- How to enforce policies about who/what/when can use the components in the system

As we dig into Istio, we'll explore these in more detail and explain how to deal with them. These are core challenges to building services-based architectures on any cloud infrastructure. In the past, non-cloud architectures did have to contend with some of these problems but, in today's cloud environments, these problems are highly amplified and have the possibility of taking down your entire system if not taken into account correctly. Let's look a little bit closer at the problems encountered with unreliable infrastructure.

1.2.1 Our cloud infrastructure is not reliable

Even though as consumers of cloud infrastructure we don't see the actual hardware, clouds are made up of millions of pieces of hardware and software components. These components comprise the compute, storage, and networking virtualized infrastructure that we can provision via self-service APIs. Any of these components can, and do, fail. In the past we did everything we could to make infrastructure highly available and build our applications on top of that with assumptions of availability and reliability. In the cloud we cannot do that.

Let's take a very simple example. Let's say a Preference service is in charge of managing customer preferences and ends up making calls to a Customer service.

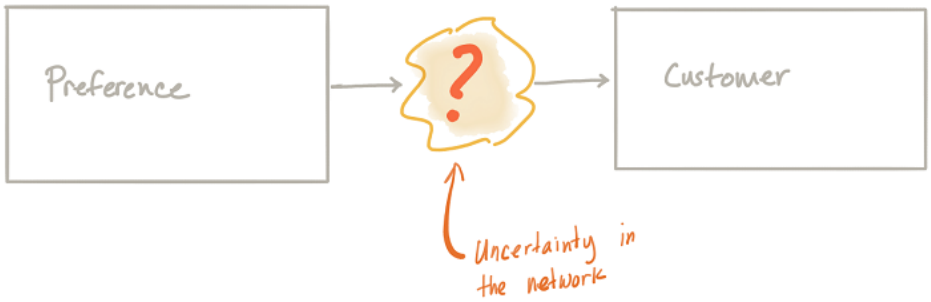


Figure 1.8 Simple service communication over an unreliable network

If Preference service calls Customer service to update some Customer data and experiences severe slow-downs when it sends a message, what does it do? A slow downstream dependency can wreak havoc on the Preference service, including causing it to fail (thus initiating a cascading failure). This scenario can happen for any number of reasons:

- The Customer service is overloaded and is running slowly
- The Customer service has a bug
- The network has firewalls and is purposely slowing traffic
- The network is congested and is slowing traffic
- The network experienced some failed hardware and is re-routing traffic
- The network card on the Customer service hardware is experiencing failures

And many other reasons. The problem is, the Preference service cannot really distinguish whether this is a failure of the Customer service or not. Again, in a cloud environment with millions of pieces of hardware and software components, these type of scenarios happen all the time.

1.2.2 Making service interaction resilient

The Preference service can try a few things. It can retry the request, although in a scenario where things are overloaded that might just add to the downstream issues. If it does retry the request, it cannot be sure that previous attempts didn't succeed. It could timeout the request after some threshold and throw an error. It can retry to a different instance of the Customer service, maybe in a different availability zone. If the Customer service experiences these or similar issues for an extended period of time, the Preference service may opt to stop calling the Customer service altogether for a "cool-off period" (a form of "circuit breaking", which we'll cover in more depth in later chapters).

Some patterns have evolved to help mitigate these types of scenarios and help make applications more resilient to unplanned, unexpected failures:

- Client side load balancing - give the client the list of possible endpoints and let it decide which to call
- Service discovery - a mechanism for finding the periodically updated list of healthy endpoints for a particular logical service
- Circuit breaking - shedding load for a period of time to a service that appears to be misbehaving
- Bulk heading - limiting client resource usage with explicit thresholds (connections, threads, sessions, etc) when making calls to a service
- Timeouts - Enforcing time limitations on requests, sockets, liveness, etc when making calls to a service
- Retries - Retrying a failed request
- Retry budgets - Applying constraints to retries; ie, limiting the number of retries in a given period (e.g., can only retry 50% of the calls in a 10s window)
- Deadlines - giving requests context about how long a response may still be useful; if outside of the deadline, disregard processing the request

Collectively these types of patterns can be though of "application networking". They have a lot of overlap with similar constructs at lower levels of the networking stack except they operate at the level of "messages" instead of "packets".

1.2.3 Understanding what's happening in real time

A very important aspect to "going faster" is making sure we're "going in the right direction". We try to get deployments out quickly so we can test how customers react to them, but they will not have an opportunity to react (or will avoid your service all together) if it's slow or not available. As we make changes to our services, do we have an understanding of what impact (positive or negative) it will have? Do we know how things are running before we make changes?

Knowing things like what services are talking with which, what typical service load looks like, how many failures we expect to see, what happens when services fail, service health, etc are all critical things to know about our services architecture. Each time we make a change by

deploying new code or configuration, we introduce the possibility of negatively impacting our key metrics. When network and infrastructure unreliability rear their ugly heads, or if we deploy new code with bugs in it, can we be confident we have enough of a pulse on what's really happening to trust that the system isn't on verge of collapse? Observing the system with metrics, logs, and traces is a crucial part of running a services architecture.

1.3 Solving these challenges with application libraries

The first organizations to figure out how to run their applications and services in a cloud environment were the large internet companies, many of whom pioneered cloud infrastructure as we know it today. These companies invested massive amounts of time and resources into building libraries and frameworks for a select set of languages that everyone had to use which helped solve the challenges of running services in a cloud-native architecture. Google built frameworks like Stubby, Twitter built Finagle, and in 2012 Netflix open sourced their microservices libraries to the open-source community. For example, with NetflixOSS, libraries targeted for Java developers handled cloud-native concerns like:

- Hystrix - circuit breaking / bulkheading
- Ribbon - client-side load balancing
- Eureka - service registration and discovery
- Zuul - dynamic edge proxy

Since these libraries were targeted for Java runtimes, they could only be used in Java projects. To use them, we'd have to create an application dependency on them, pull them into our classpath, and then use them in our application code. For example of using NetflixOSS Hystrix:

Pulling a dependency on Hystrix in your dependency control system

Listing 1.1 Hystrix dependency for Maven (Java)

```
<dependency>
    <groupId>com.netflix.hystrix</groupId>
    <artifactId>hystrix-core</artifactId>
    <version>x.y.z</version>
</dependency>
```

To use Hystrix we wrap our commands with a base Hystrix class `HystrixCommand`

Listing 1.2 Using Hystrix to implement circuit breaking

```
public class CommandHelloWorld extends HystrixCommand<String> {

    private final String name;

    public CommandHelloWorld(String name) {
        super(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"));
        this.name = name;
    }

    @Override
    protected String run() {
        // a real example would do work like a network call here
        return "Hello " + name + "!";
    }
}
```

If each application is responsible for building resilience into its code, then we can distribute the handling of these concerns and eliminate central bottlenecks. In large scale deployments on unreliable cloud infrastructure, this is a desirable trait of our system.

1.3.1 Drawbacks to application-specific libraries

Although we've mitigated a concern about large-scale services architectures when we decentralize and distribute the implementation of application resiliency into the applications themselves, we've introduced some new challenges. The first challenge is around the expected assumptions of any application. If we wish to introduce a new service into our architecture, it will be constrained to implementation decisions made by other people and other teams. For example, to use NetflixOSS Hystrix, you must use Java or some JVM based technology. Typically circuit breaking and load balancing go together, so you'd need to use both of those resilience libraries. To use Netflix Ribbon for load balancing, you need some kind of registry to discover service endpoints which may mean using Eureka. Going down this path of using libraries introduces implicit constraints around a very undefined protocol for interacting with the rest of the system.

The second issue is around introducing a new language or framework to implement a service. You may find that NodeJS is a better fit for implementing user-facing APIs but the rest of your architecture uses Java and leverages NetflixOSS. You may opt to find a different set of libraries to implement resilience patterns for you. You may try to find analogous packages like Resilient (www.npmjs.com/package/resilient) or `hystrixjs` (www.npmjs.com/package/hystrixjs). And for each language you wish to introduce (microservices enables a polyglot development environment — though standardizing on a handful of languages is usually best), you'll need to search, certify, and introduce to your development stack. Each of these libraries will have a different implementation making different assumptions. In some cases you may not be able to find analogous replacements for each framework/language combination. What you end up with is a partial implementation for some languages and overall inconsistency in the implementation that

makes it very difficult to reason about in failure scenarios and possibly contributes to obscuring/propagating failures.

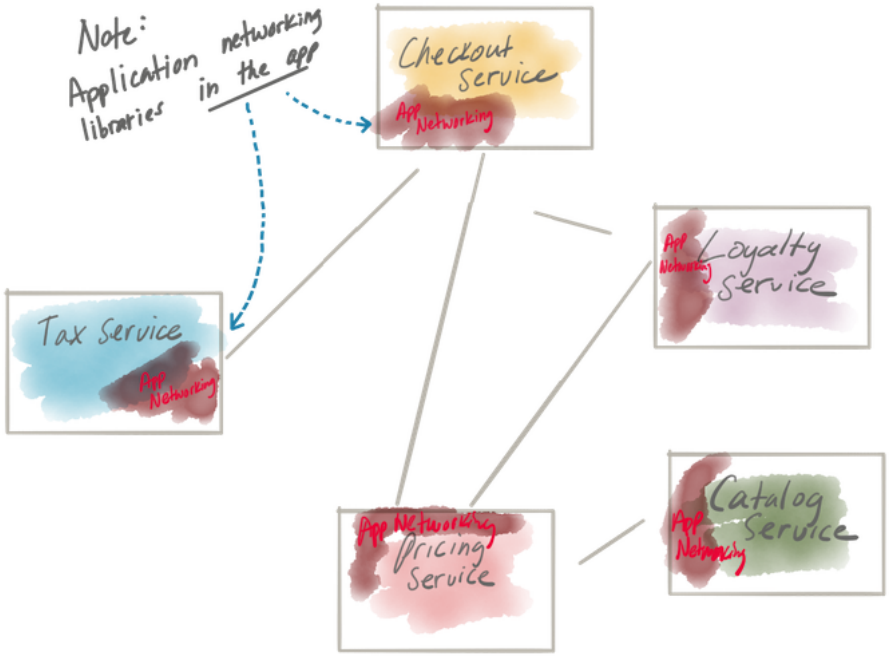


Figure 1.9 Application networking libraries commingled with application

Lastly, maintaining a handful of libraries across a bunch of programming languages and frameworks requires a lot of discipline and is very hard to get right. The key here is ensuring all of the implementations are consistent and correct. One deviation and you've introduced more unpredictability into your system. Pushing out updates and changes across a fleet of services all at the same time can be a daunting task as well.

Although the decentralization of application networking is better for cloud architectures, the operational burden and constraints it puts on a system in exchange will be difficult for most organizations to swallow. Even if they do decide to take on that challenge, getting it right is even harder. What if there was a way to get the benefits of decentralization without paying the price of massive overhead in maintaining and operating these applications with embedded libraries?

1.4 Pushing these concerns to the infrastructure

These basic application-networking concerns are not specific to any particular application, language, or framework. Retries, timeouts, client-side load balancing, circuit breaking, etc. are also not differentiating application features. They are critical concerns to have as part of your service but investing massive time and resources into language-specific implementations for each language you intend to use (including the other drawbacks from the previous section) has a smell. What we really want is a technology-agnostic way to implement these concerns and relieve applications from having to implement this themselves.

1.4.1 Don't we already have this in our container platforms?

Earlier, we mentioned Linux containers simplifying the application surface insofar starting, stopping, and overall managing an application can rely on generic, non-language specific tooling. We also mentioned Kubernetes takes this to the next level by implementing application constructs (deployments, services, service-affinity/anti-affinity, health checking, scaling, etc) that also apply to services regardless of language or technology. In fact, Kubernetes has a simple load-balancing and service discovery mechanism built into it as well. In Kubernetes we can use a single virtual IP to communicate with backend pods. Kubernetes will automatically send the traffic to the pods in a round-robin or random manner. Kubernetes handles the registration/eviction of instances in the group based on their health status and whether they match a grouping predicate (labels and selectors). Then services can use DNS for service discovery and load balancing regardless of their implementation. No need for special language-specific libraries or registration clients. In this example, Kubernetes has allowed us to move simple networking concerns out of the applications and into the infrastructure.

Kubernetes is a great container deployment and management platform, and has set an example for creating generic distributed-system management primitives and exposing them as infrastructure services. However, Kubernetes is a container deployment platform. It will not evolve into the many facets of application networking. Kubernetes is built to be extended in a very natural way through API federation and the expectation for any higher-order application services to be built as addons.

1.4.2 The application-aware service proxy

A way to move these horizontal concerns into the infrastructure is to use a proxy. A proxy is an intermediate infrastructure component that can handle connections and redirect them to appropriate backends. We use proxies all the time (whether we know or not) to handle network traffic, enforce security, and load balance work to backend servers. For example, HA proxy is a simple but powerful reverse proxy for distributing connections across many backend servers. `mod_proxy` is a module for the Apache HTTP server that also acts as a reverse proxy. In our corporate IT systems, typically all outgoing internet traffic is routed through forwarding proxies. These proxies monitor the traffic and block certain types of activities.

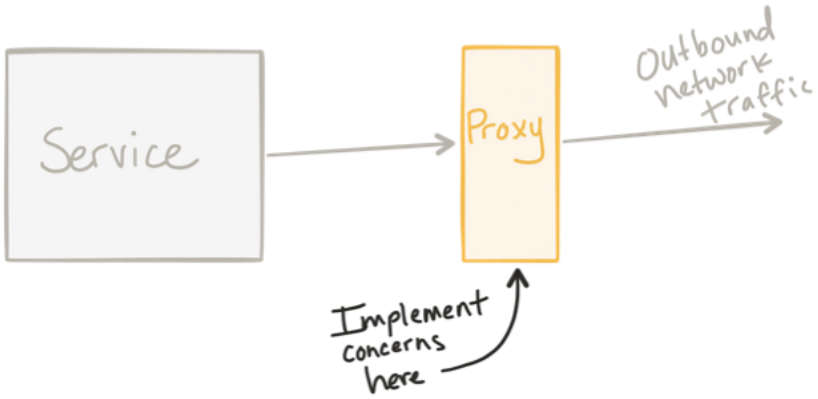


Figure 1.10 Use a proxy to push these horizontal concerns such as resilience, traffic control, security, etc. out of the application implementation

What we want for this problem, however, is a proxy that's "application aware" and able to perform application networking on behalf of our services. To do this, this "service proxy" will need to understand application constructs like messages and requests instead of more traditional infrastructure proxies which understand connections and packets. In other words, we need an L7 proxy.

1.4.3 Meet Envoy proxy

A service proxy that has emerged in the open-source community to be a versatile, performant, and capable application-level proxy is Envoy Proxy (envoyproxy.io). Envoy was developed at Lyft as part of their Service Oriented Architecture infrastructure and is capable of implementing application resilience and other networking concerns outside of the application. Envoy gives us networking capabilities like retries, timeouts, circuit breaking, client-side load balancing, service discovery, security, and metrics-collection without any explicit language or framework dependencies.

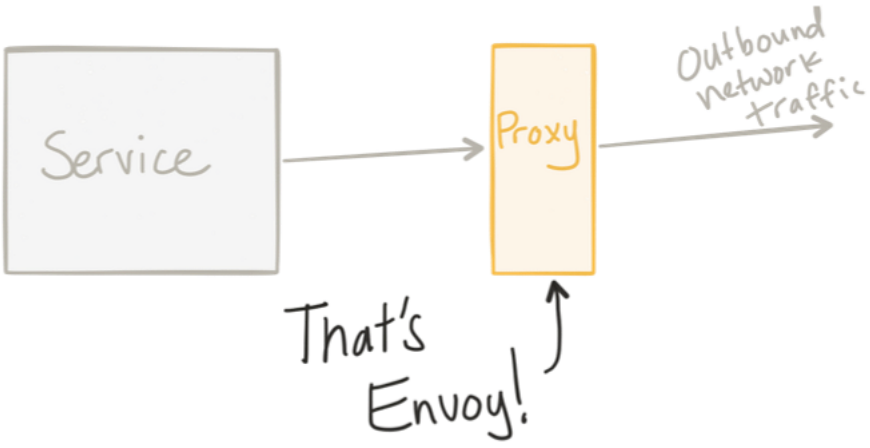


Figure 1.11 Envoy proxy as out-of-process participant in application networking

The power of Envoy is not limited to these application-level resilience aspects. Envoy also captures many application-networking metrics like requests per second, number of failures,

circuit-breaking events, and more. By using Envoy, we can automatically get visibility into what's happening between our services which is where we start to see a lot of unanticipated complexity. Envoy proxy forms the foundation for solving cross-cutting, horizontal reliability and observability concerns for a services architecture and allows us to push these concerns outside of the applications and into the infrastructure. We'll cover more of Envoy in ensuing sections and chapters.

We can deploy these service proxies alongside our applications so we can get these features (resilience and observability) out of process from the application, but at a fidelity that is very application specific. In this model, applications that wish to communicate with the rest of the system do so by passing their requests to Envoy first, which then handles the communication upstream:

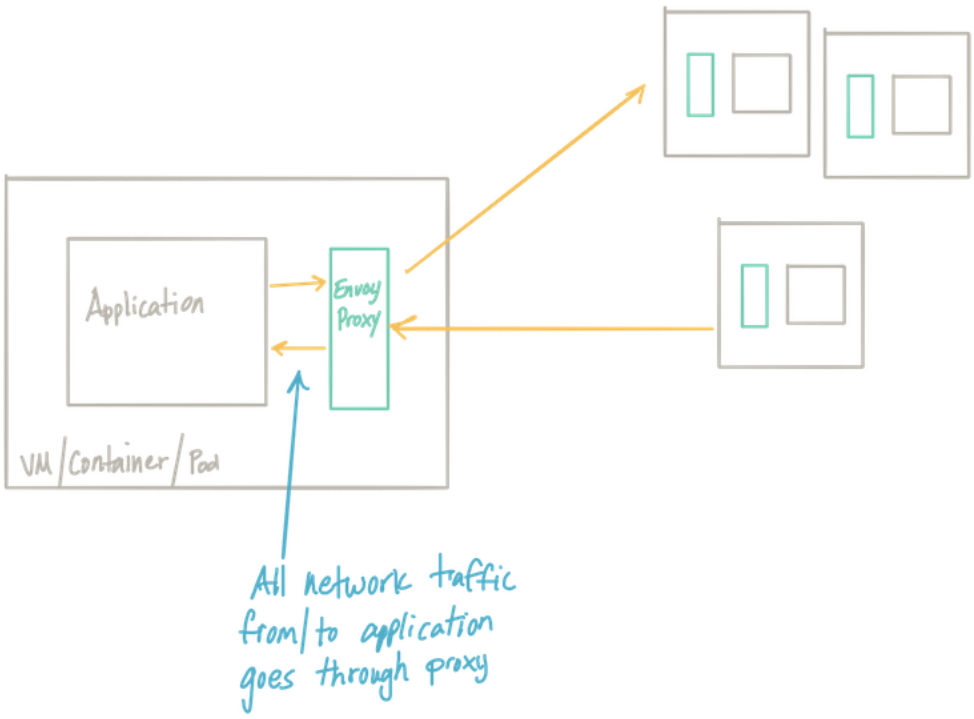


Figure 1.12 Envoy Proxy out of process to application

Service proxies can also do things like collect distributed tracing spans so we can stitch together all of the steps a particular request took. We can see how long each step took and look for potential bottlenecks or bugs in our system. If all applications talk through their own proxy to the outside world, and all incoming traffic to an application goes through our proxy, we've gained some important capabilities for our application without changing any application code. This proxy+application combination forms the foundation of a communication bus known as a service mesh.

We can deploy a service proxy like Envoy along each instance of our application as a single atomic unit. For example, in Kubernetes we can co-deploy a service proxy with our application

in a single Pod. This kind of deployment pattern is known as a **sidecar** deployment in which the service proxy gets deployed to complement the main application instance.

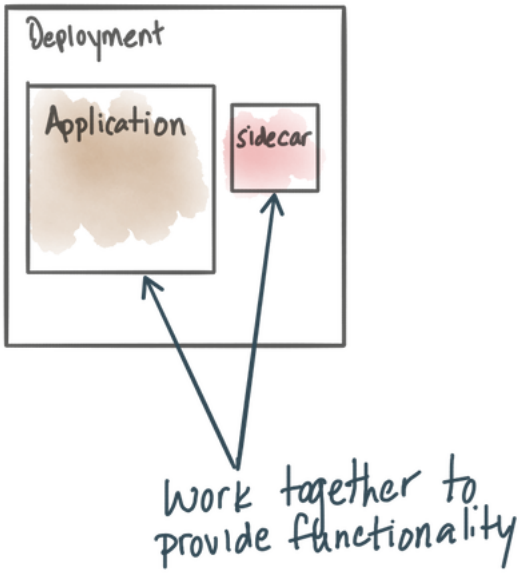


Figure 1.13 A sidecar deployment is an additional process that works cooperatively with the main application process to deliver a piece of functionality

1.5 What's a service mesh?

Service proxies like Envoy help add important capabilities to our services architecture running in a cloud environment. Each application can have its own requirements or configurations for how a proxy should behave given its workload goals. With an increasing number of applications and services, it can be quite difficult to configure and manage a large fleet of proxies. Moreover, having these proxies in place at each application instance opens opportunities for building interesting higher-order capabilities that we would otherwise have to do in the applications themselves.

A *service mesh* is a distributed application infrastructure that is responsible for handling network traffic on behalf of the application in a transparent, out of process manner.

The service proxies form the "data plane" through which all traffic is handled and observed. The data plane is responsible for establishing, securing, and controlling the traffic through the mesh. The management components that instruct the data plane how to behave is known as the "control plane". The control plane is the brains of the mesh and exposes an API for operators to manipulate the network behaviors. Together, the data plane and the control plane provide important capabilities necessary in any cloud-native architecture such as:

- Service resilience
- Observability signals
- Traffic control capabilities
- Security

- Policy enforcement

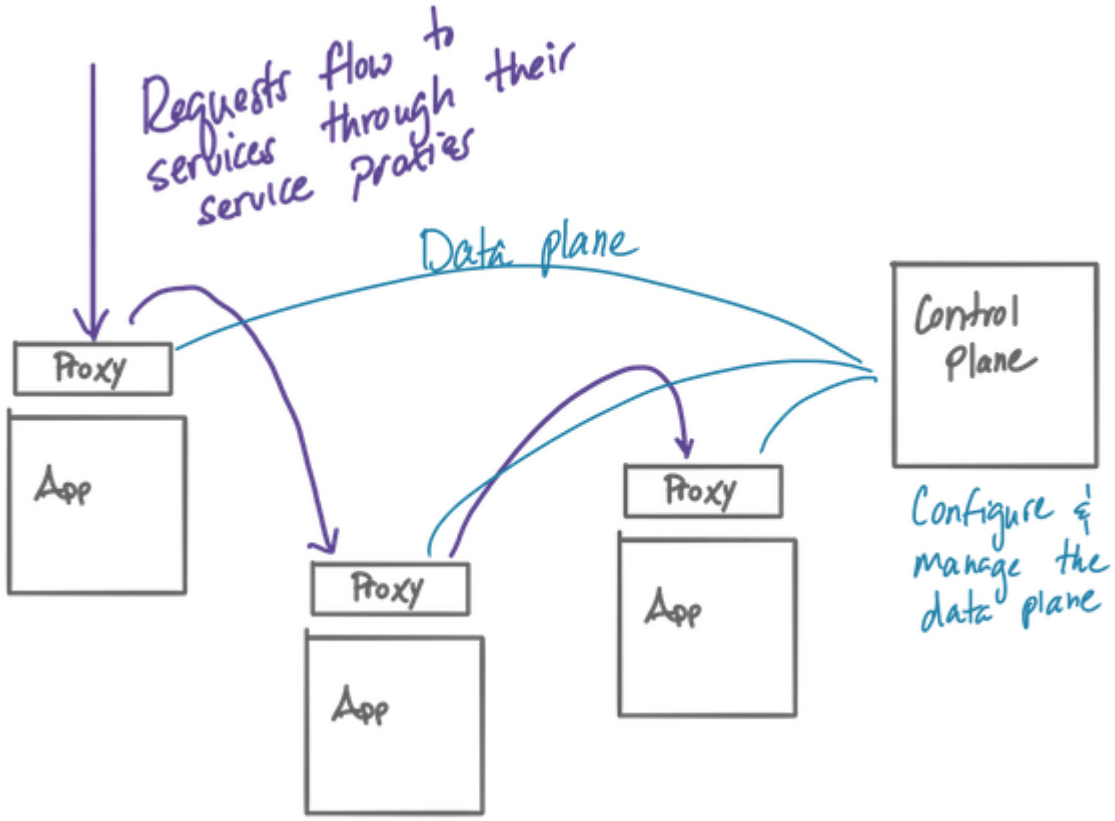


Figure 1.14 Service mesh architecture with co-located application-level proxies (data plane) and management components (control plane)

The service mesh takes on the responsibility of making service communication resilient to failures by implementing capabilities like retries, timeouts, and circuit breakers. It's also capable of handling evolving infrastructure topologies by handling things like service discovery, adaptive and zone-aware load balancing, and health checking. Since all of the traffic flows through the mesh, operators can control and direct traffic explicitly. For example, if we want to deploy a new version of our application, we may want to expose it to only a small fraction, say 1%, of live traffic. With the service mesh in place, we have the power to do that. Of course, the converse of control in the service mesh is understanding its current behavior. Since the traffic flows through the mesh, we're able to capture detailed signals about the behavior of the network by tracking metrics like request spikes, latency, throughput, failures, etc. We can leverage this telemetry to paint a picture of what's actually happening in our system. Lastly, since the service mesh controls both ends of the network communication between applications, it can enforce strong security like transport-level encryption with mutual TLS.

The service mesh provides all of these capabilities to service operators with very little to no application code changes, dependencies, or intrusions. For some of the capabilities, there will have to be minor cooperation with the application code, but we can avoid large complicated library dependencies. With a service mesh, it doesn't matter what application framework or

programming language you've used to build your application; these capabilities are implemented consistently and correctly.

The capabilities that service mesh provides allow our service teams to move quickly, safely, and confidently when implementing and delivering changes to their systems to test their hypotheses and deliver value.

1.6 Introducing Istio service mesh

Istio is an open-source implementation of a service mesh founded by Google, IBM, and Lyft. Istio helps you add resilience and observability to your services architecture in a transparent way. With Istio, applications don't have to know that they're part of the service mesh. Whenever they interact with the outside world, Istio will handle the networking on behalf of the application. This means it doesn't matter if you're doing microservices, monoliths or anything in between, Istio can bring many benefits. Istio's data plane uses Envoy proxy by default out of the box and helps you configure your applications to have an instance of the service proxy (Envoy) deployed alongside it. Istio's control plane is made up of a few components that provide APIs for end users/operators, configuration APIs for the proxies, security settings, policy declarations, and more. We'll cover these control-plane components in future sections of this book.

Istio was originally built to run on Kubernetes, but was written from the perspective of being deployment-platform agnostic. This means you can leverage an Istio-based service mesh across deployment platforms like Kubernetes, OpenShift, Mesos, and Cloud Foundry and even traditional deployment environments like VMs. In later chapters we'll take a look at how powerful this can be for hybrid deployments across combinations of clouds including private data centers. In this book, we'll consider our deployments on Kubernetes until we get to the more advanced chapters.

NOTE	**More nautical references**
	What does *Istio* mean? Istio is the Greek word "to sail" which goes along nicely with the rest of the Kubernetes nautical words

With a service proxy next to each application instance, applications no longer need to have language-specific resilience libraries for circuit breaking, timeouts, retries, service discovery, load balancing, et. al. Moreover, the service proxy also handles metric collection, distributed tracing, and log collection.

Since the traffic in the service mesh is flowing through the Istio service proxy, Istio has control points at each application to influence and direct its networking behavior. This allows a service operator to control traffic flow and implement fine-grained releases with canary releases, dark launches, graduated roll outs, and A/B style testing. We'll explore these capabilities in later chapters.

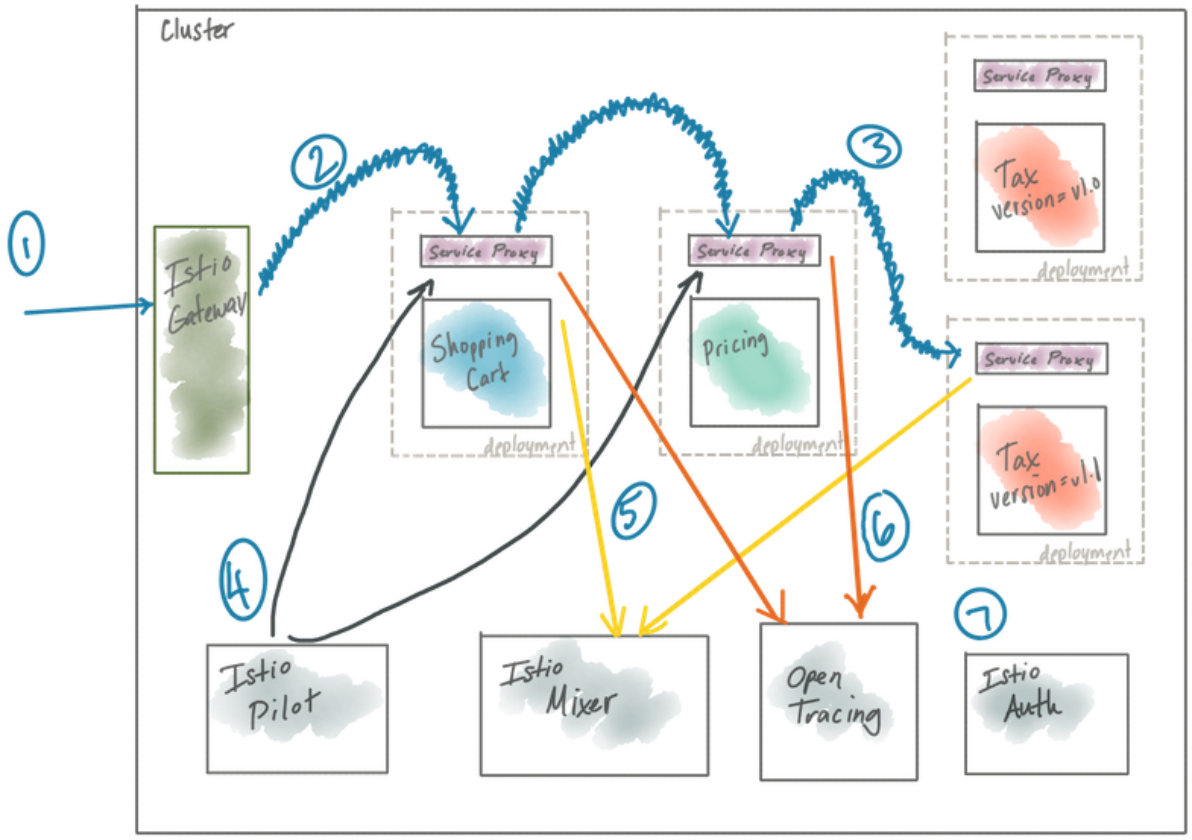


Figure 1.15 Istio is an implementation of a service mesh with a data plane based on Envoy and a control plane

1. Traffic comes into the cluster (ie, a client issues a POST "/checkout" request to our shopping cart service)
2. Traffic goes to the shopping-cart service but goes to the Istio service proxy (Envoy). Note, this traffic (and all inter-service communication in the mesh) is secured with mutual TLS by default. The certificates for establishing mTLS are provided by (7).
3. Istio determines (by looking at the request headers) that this request was initiated by a customer in the North America region, and for those customers, we want to route some of those requests to v1.1 of the Tax service which has a fix for certain tax calculations; Istio routes the traffic to the v1.1 Tax service
4. Istio Pilot is used to configure the istio proxies which handle routing, security, and resilience
5. Request metrics are periodically sent back to the Istio Mixer which stores them to back end adapters (to be discussed later)
6. Distributed tracing spans (like Jaeger or Zipkin) are sent back to an tracing store which can be used to later track the path and latency of a request through the system
7. Istio Auth which manages certificates (expiry, rotation, etc) for each of the istio proxies so mTLS can be enabled transparently

An important requirement for any services-based architecture is security. Istio has security enabled by default. Since Istio controls each end of the application's networking path, it can transparently encrypt the traffic by default. In fact, to take it a step further, Istio can manage key

and certificate issuance, installation, and rotation so that services get mutual TLS (transport layer security) out of the box. If you've ever experienced the pain of installing and configuring certificates for mutual TLS, you'll appreciate both the simplicity of operation and how powerful this capability is. Istio can assign workload identity and embed that into the certificates. Istio can use the identity of the different workloads to further implement powerful access-control policies.

Lastly, but no less important from the previous capabilities, with Istio you can implement quotas, rate limiting and organizational policies. With Istio's policy enforcement, you can create very fine-grained rules about what services are allowed to interact with which other, and which are not. This becomes especially important when deploying services across clouds (public and on premise).

Istio is a powerful implementation of a service mesh. It's capabilities allow you to simplify running and operating a cloud-native services architecture potentially across a hybrid environment. It has a vibrant, open, and diverse community with folks from Google, IBM, Lyft, Red Hat, VMWare, Pivotal, Tigera, and Weaveworks (and many others)! Throughout the rest of this book we'll show you how take advantage of Istio's functionality to operate your microservices in a cloud-native world.

1.6.1 How service mesh relates to Enterprise Service Bus

The Enterprise Service Bus (ESB) from the Service Oriented Architecture (SOA) days have some similarities to service mesh, at least in spirit. If we take a look at how the ESB was originally described in the early days of SOA, we even see some similar language.

> *The enterprise service bus (ESB) is a silent partner in the SOA logical architecture.* [1] *Its presence in the architecture is transparent to the services of your SOA application. However, the presence of an ESB is fundamental to simplifying the task of invoking services – making the use of services wherever they are needed, independent of the details of locating those services and transporting service requests across the network to invoke those services wherever they reside within your enterprise.*

In this description of the ESB, we see it's supposed to be a *silent partner* which means, the applications should not know about it. With the service mesh, we expect similar behavior. The service mesh should be transparent to the application. The ESB also is "fundamental to simplifying the task of invoking services". for the ESB, this included things like protocol mediation, message transformation, and content based routing whereas the service mesh is not responsible for all the things the ESB does. The service mesh does provide request resilience through retries, timeouts, circuit breaking and it does provide services like service discovery and load balancing.

Overall, there are a few significant differences between the service mesh and the ESB:

- The ESB introduced a new silo in organizations that was the gatekeeper for service integrations within the enterprise
- It was a very centralized deployment/implementation
- It mixed application networking, and service mediation concerns
- Often based on complicated proprietary vendor software

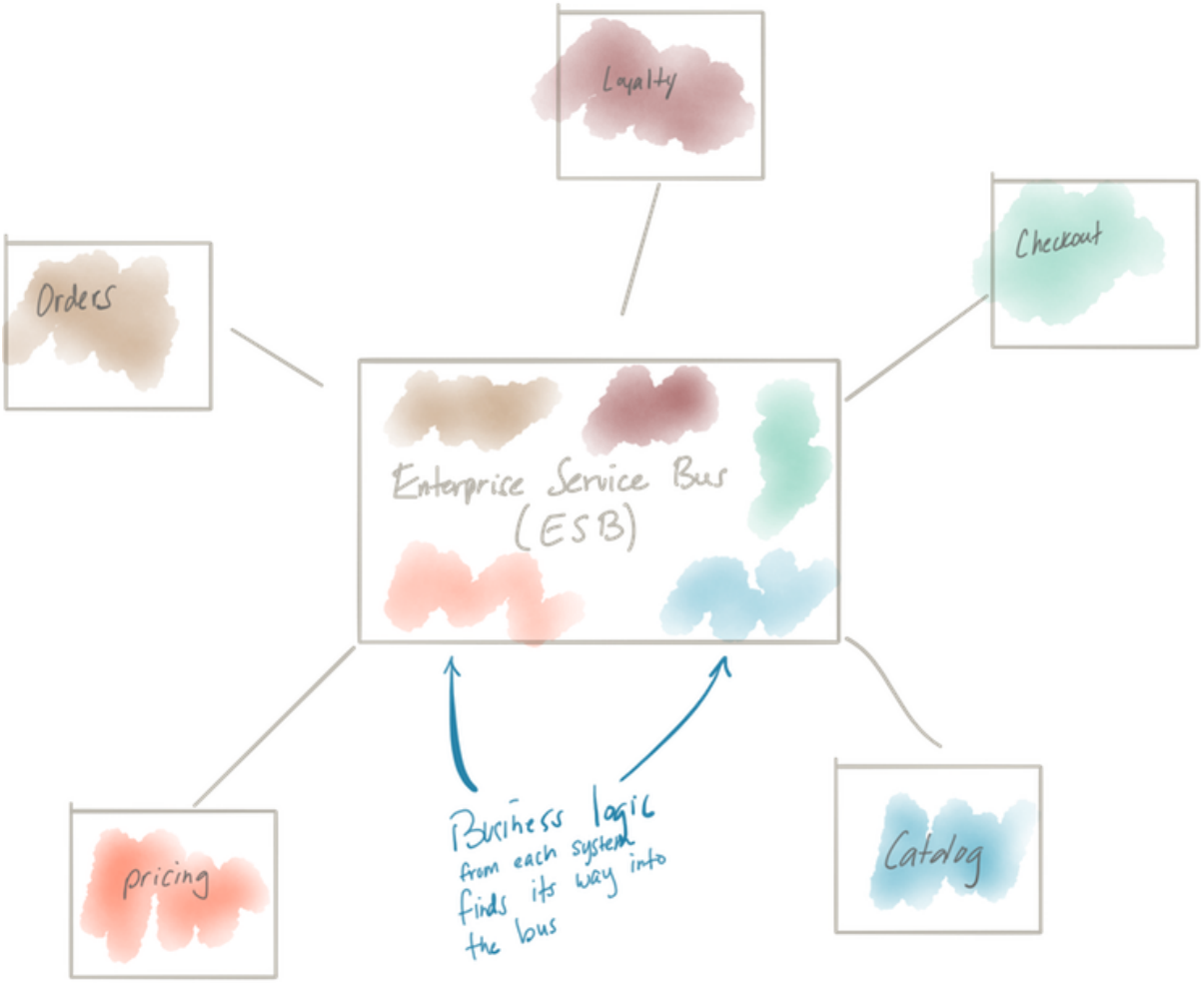


Figure 1.16 An ESB implementation was often a centralized system that comingled application business logic with application routing, transformation and mediation

The service mesh's role is only in application networking concerns. Complex business transformations (X12, EDI, HL7, etc), business process orchestration, process exceptions, service orchestration, etc do not belong in the service mesh. Additionally, the service mesh data plane is highly distributed with its proxies collocated with the applications. This eliminates single points of failure/bottlenecks that often appeared with the ESB architecture. Lastly, both operator and service teams are responsible for establishing service level objectives (SLOs) and configuring the service mesh to support this. The responsibility of integration with other systems is no longer the purview of some centralized team; all service developers share that responsibility

1.6.2 How service mesh relates to API gateway

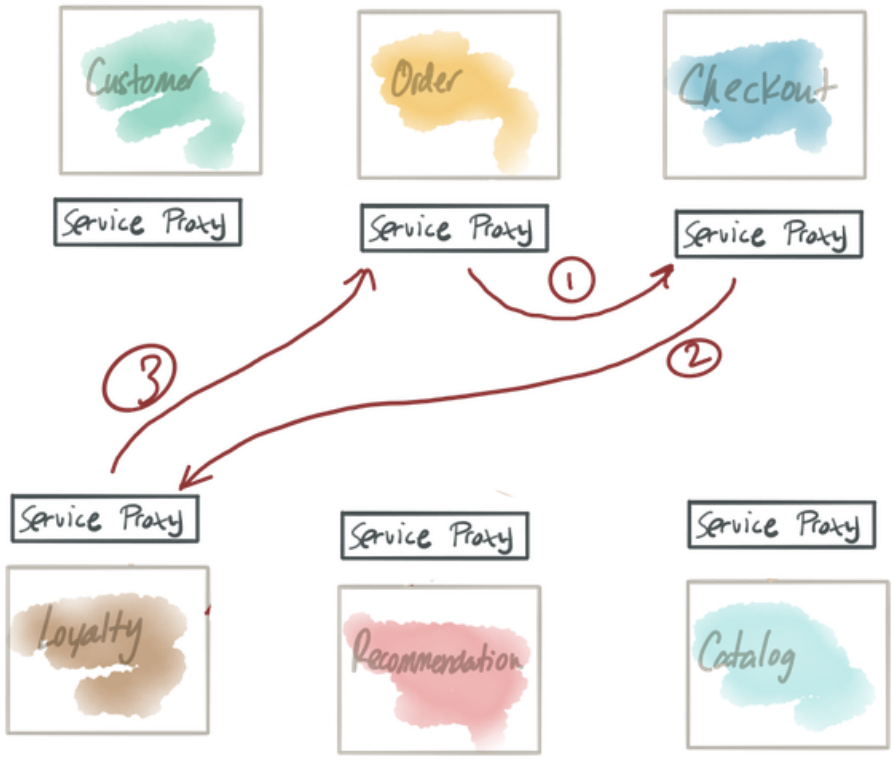
Istio and service-mesh technology also have some similarities and differences with API gateways. API gateway infrastructure (not the microservices pattern from microservices.io/patterns/apigateway.html) is used in API management suites to provide a public facing endpoint for an organization's public APIs. Its role is to provide security, rate limiting, quota management and metric collection for these public APIs and tie into an overall API management solution which includes API plan specification, user registration, billing, and other operational concerns. API gateway architectures vary wildly, but have been used mostly at the edge of an architecture to expose public APIs. They have also been used for internal APIs to provide a centralization of security, policy, and metric collection. The API gateway is where every service would communicate with to get to any other service. This, however, creates a centralized system through which traffic travels and can become a source of bottlenecks as described with the ESB and messaging bus.

Figure 1.17 API gateway for service traffic

Note in this case, all internal traffic between services is traversing the API gateway. This means for each service in our graph, we're taking two hops: one to get to the gateway and one to get to the actual service. This has implications on not just network overhead and latency, but also security. With this multi-hop architecture, the API gateway cannot secure the transport

mechanism with the application unless the application participates in the security configuration. Lastly, in many cases, the API gateway didn't implement resilience capabilities like circuit breakers or bulkheading.

In a service mesh, the proxies are collocated with the services themselves and do not take on additional hops. They're also decentralized so each application can configure its proxy for its particular work loads and not be affected by noisy neighbor scenarios. Since each proxy lives with its corresponding application instance, it can secure the transport mechanism from end to end without the application knowing or actively participating.



The service mesh is quickly becoming a place to enforce and implement API gateway functionality. As the service mesh technologies, like Istio, continue to mature, we'll see API management built on top of the service mesh and not need specialized API gateway proxies.

1.6.3 Can I use Istio for non-microservices deployments?

Istio's power shines as you move to architectures that experience large numbers of services, interconnections, and networks over unreliable cloud infrastructure potentially spanning clusters, clouds and data centers. This does not mean Istio does not provide any benefit to architectures that do not match this description. Since Istio is an out-of-process implementation for its functionality, it can fit existing legacy or brownfield environments just fine.

For example, if you have existing monolith deployments, the Istio service proxy (Envoy) can be deployed alongside each monolith instance and transparently handle the network traffic for it. At

the minimum, this can add request metrics that become very useful in understanding the application's usage, latency, throughput and failure characteristics. It can also participate in higher-level features like policy enforcement about what services are allowed to talk to it. This capability becomes highly important in a hybrid-cloud deployment with monoliths running on premise and cloud services potentially running in a public cloud. With Istio, we can enforce policies such as "cloud services cannot talk to and use data from the on-premise applications."

You may also have an older vintage of microservices implemented with resilience libraries like Netflix OSS. Istio brings powerful capabilities to these deployments as well. Even if both Istio and the application implement functionality like a circuit breaker, you can be safe knowing that the more restrictive policies will kick in and everything should work fine. There can be wonkiness with timeouts and retries that could conflict, but using Istio you can test your service before you ever make it to production to find these conflicts.

1.6.4 What problems does service mesh NOT solve?

You should pick the technology you use in your implementations based on what problems you have and what capabilities you need. Technology like Istio, and service mesh in general, are powerful infrastructure capabilities and touch a lot of areas of a distributed architecture but are not and should not be considered for every problem you may have. An ideal cloud architecture would separate out different concerns from each layer in the implementation.

At the lower level of our architecture we have our deployment automation infrastructure. This is responsible for getting code deployed onto your platform (containers, Kubernetes, public cloud, VMs, etc). Istio does not encroach nor prescribe what deployment automation tools you should use.

At the higher level you have application business logic. This is the differentiating code that a business must write to stay competitive. This code includes the business domain as well as knowing which services to call, in what order, what to do the service interaction responses (how to aggregate them together, etc), and what to do when there are process failures. Istio does not implement or replace any business logic. It does not do service orchestration, business payload transformation, payload enrichment, splitting/aggregating, or any rules computation. These capabilities are best left to libraries and frameworks inside your applications.

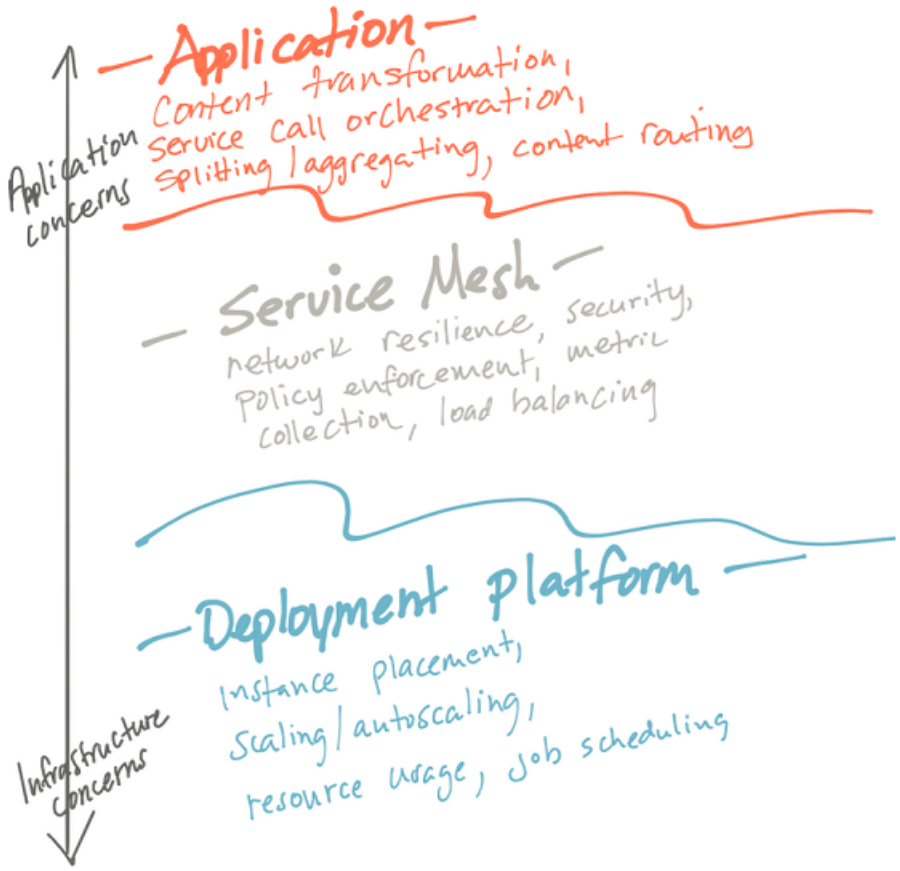


Figure 1.18 An overview of separation of concerns in cloud-native applications where Istio plays a supporting role to the application layer and sits above the lower-level deployment layer

Istio plays the role as connective tissue between the deployment platform and the application code. It's role is to facilitate taking complicated networking code out of the application. It can do content based routing based on external metadata that is part of the request (ie, HTTP headers, etc). It can do fine grained traffic control and routing based on service and request metadata matching. It can secure the transport and offload security token verification. It can enforce quota and usage policies defined by service operators.

Understanding Istio's capabilities, similarities with other systems, and where it fits in the architecture is vital to not making the same mistakes with promising technology as we may have experienced in the past.

Now that we have a basic understanding of what Istio is, the best way to further get acquainted with its power is to use it. We'll take a look at using Istio in Chapter 2 to achieve basic metrics collection, reliability, and traffic control.

1.7 Summary

- Quick wrap up, hitting main points:
  - We need capabilities to deliver software faster
  - We need to balance speed and safety
  - Containers help simplify and organize deployments, but still need to solve for service communication
  - Service reliability, policy, security, control are critical things to solve for service communication
  - Can get a more uniform, consistent, and correct experience by pushing these concerns to infrastructure
  - A service mesh is that infrastructure
  - Istio is an open-source service mesh
  - Istio is not an ESB although there are some superficial similarities

2 First steps with Istio

This chapter covers:

- Installing Istio on Kubernetes
- Understanding the Istio control plane components
- Deploying an application with the Istio proxy
- Controlling traffic with Istio RouteRules

Istio solves some of the difficult challenges of service communication in cloud environments and provides a lot of capabilities to both developers and operators. We'll cover these capabilities and how it all works in subsequent chapters, but to help you get a feel for some of the features of Istio, we're going to do a basic installation (more advanced installation can be found in Appendix XX) and deploy a few services. From there, we'll explore the components that make up Istio and what functionality we can provide to our sample services. Lastly, we'll look at how to do basic traffic routing, and metrics collection, and resilience. Further chapters will dive deeper into the functionality.

2.1 Deploying Istio on Kubernetes

We're going to deploy Istio and our sample applications using containers and we'll use the Kubernetes container platform to do that. Kubernetes is a very powerful container platform capable of scheduling and orchestrating containers over a fleet of host machines known as Kubernetes nodes. These nodes are linux machines capable of running containers, though Kubernetes handles those mechanisms. As we'll see, Kubernetes is a great place to initially kick the tires with Istio, though we should make clear, Istio is intended to support multiple types of workloads including those running on Virtual Machines. We'll explore how to deploy Istio on other platforms in subsequent chapters.

2.1.1 Using Docker for Desktop for our samples

To get started, we're going to need access to a Kubernetes distribution, and for this book, we'll use Docker for Desktop. Docker for Desktop gives a slim virtual machine on your host computer that's capable of running Docker and Kubernetes. Docker for Desktop also has nice integration between the host machine and the virtual machine. As the reader, you're not constrained to using Docker for Desktop to run these samples and follow along in this book. In fact, these samples should run well on any variant of Kubernetes including Google Container Engine (GKE), OpenShift, or your own self-bootstrapped Kubernetes distribution. Please see the Docker for Dekstop documentation for setting up Kubernetes for your machine. After succesfully setting it up, you should be able to connect to your kubernetes clusters like this:

Listing 2.1 Starting and verifying Minikube

```
$  kubectl get nodes
NAME             STATUS   ROLES    AGE     VERSION
docker-desktop   Ready    master   4h30m   v1.16.6-beta.0
```

> **NOTE** **Minimum Kubernetes version**
>
> Istio 1.7.x, used in this book, requires a minimum of Kubernetes version 1.16.x

2.1.2 Getting the Istio distribution

Next, we want to install Istio into our Kubernetes distribution. We will use the `istioctl` command-line tool to install Istio. To do that, we need to download the Istio 1.7.0 distribution from the Istio release page on github.com at https://github.com/istio/istio/releases and download the distribution for your operating system. You can choose Windows, MacOS X / Darwin, and Linux. Alternatively, you could run this handy script:

```
curl -L https://istio.io/downloadIstio | sh -
```

After downloading the distribution for your operating system, you should extract the compressed file to a directory. From there, you can explore the contents of the distribution which includes samples, the installation resources, as well as a binary command-line interface for your OS. For example, on Mac OS X or Linux, you can do the following:

Listing 2.2 Extracting the Istio distribution for Mac OS X:

```
$  tar -xzf istio-1.7.0-osx.tar
$  cd istio-1.7.0
$  ls -l
total 48
-rw-r--r--@  1 ceposta  staff  11348 Aug 21 12:00 LICENSE
-rw-r--r--@  1 ceposta  staff   5756 Aug 21 12:00 README.md
drwxr-x---@  3 ceposta  staff     96 Aug 21 12:00 bin
-rw-r-----@  1 ceposta  staff    815 Aug 21 12:00 manifest.yaml
drwxr-xr-x@  6 ceposta  staff    192 Aug 21 12:00 manifests
drwxr-xr-x@ 21 ceposta  staff    672 Aug 21 12:00 samples
drwxr-x---@  7 ceposta  staff    224 Aug 21 12:00 tools
```

Browse the distribution directories to get an idea of what comes with Istio to help you get started. For example, in the `samples` directory, you'll see a handful of tutorials and applications to help you get your feet wet with Istio. Going through each of these will give you a good initial idea of what Istio can do and how to interact with the Istio components. We'll take a deeper look in the next section. There's also the `tools` directory which contains a few tools for troubleshooting istio deployments as well as bash-completion for `istioctl`. The `manifests` directory contains helm charts Istio profiles for installing Istio into your specific platform including Kubernetes, VMs, and GCE. You likely don't need to use these directly (as we'll see), but they're there for customization purposes. Lastly, there's the `bin` directory which contains binaries (at the time of writing, just one binary) for administering Istio.

Of particular interest in the `bin` directory, you'll find that Istio ships with a simple CLI tool for interacting with the Istio APIs. This binary is similar to `kubectl` for interacting with the Kubernetes API, but includes a handful of user-experience enhancing commands. Run the `istioctl` binary to verify everything works as expected.

Listing 2.3 Verify `istioctl` is able to work on your machine

```
$  ./bin/istioctl version
no running Istio pods in "istio-system"
1.7.0
```

At this point, you can add the `istioctl` CLI to your path so it's available wherever you navigate on the command line.

Lastly, let's verify any prerequisites have been met in our Kubernetes cluster (e.g., version) and identify any issues we my have **before** we begin installation. We can run the following command to do that:

Listing 2.4 Verify our Kubernetes cluster is eligible for an Istio installation

```
$  istioctl x precheck
Checking the cluster to make sure it is ready for Istio installation...

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

#2. Kubernetes-version
-----------------------
Istio is compatible with Kubernetes: v1.16.6-beta.0.

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

#4. Kubernetes-setup
-----------------------
Can create necessary Kubernetes configurations: Namespace,
ClusterRole,ClusterRoleBinding,CustomResourceDefinition,Role,
ServiceAccount,Service,Deployments,ConfigMap.

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

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

At this point we've downloaded the distribution files and verified the `istioctl` CLI tools are a fit for our operating system and Kubernetes cluster. Next, let's do a basic installation of Istio so we can get hands on with its concepts.

2.1.3 Installing the Istio components into Kubernetes

In the distribution you just downloaded and unpacked, there was a directory called `manifests` which contains a collection of charts and resource files for installing Istio into the platform of your choice. The preferred method for any real installation of Istio is to use `istioctl` or the Istio operator. With the operator, you get more flexibility in which components you install and can customize things in detail. For this book, we're going to use `istioctl` and various pre-curated "profiles" to take a step-by-step and incremental approach to adopting istio.

To perform the demo install, use the `istioctl` CLI tool:

Listing 2.5 Use `istioctl` to install Istio into your Kubernetes cluster

```
$  istioctl install --set profile=demo

✓ Istio core installed
✓ Istiod installed
✓ Ingress gateways installed
✓ Egress gateways installed
✓ Installation complete
```

After running this command, you may have to wait a few moments for the Docker images to properly download and for the deployments to succeed. Once things have settled in, you can run the `kubectl` command to list all of the pods in the `istio-system` namespace. You may also see a notification that your cluster doesn't support third-party JWT authentication. This is fine for this part of he book, we'll explain more about this later in Chapter XX.

The `istio-system` namespace is special in that the control plane is deployed into it and is able to act as a cluster-wide control plane for Istio. We'll cover more advanced multi-tenant and split-namespace scenarios later in the book.

Listing 2.6 List the running components installed into Kubernetes

```
$  kubectl get pod -n istio-system
NAME                                    READY   STATUS    RESTARTS   AGE
istio-egressgateway-55d547456b-q2ldq    1/1     Running   0          92s
istio-ingressgateway-7654895f97-2pb62   1/1     Running   0          93s
istiod-5b9d44c58b-vvrpb                 1/1     Running   0          99s
```

So what exactly did we install here? In the first chapter, we introduced the concept of a service mesh and said Istio is an open-source implementation of a service mesh. We also said a service mesh is comprised of a data-plane (i.e., service proxies) and control-plane components. After installing Istio into a cluster, you should see the control plane and supporting components. We've not yet installed any applications so the data-plane service proxies aren't deployed yet, but as soon as we install an application we'll see how to inject the service proxies. If we see the list of Kubernetes pods running from Listing 2.6 we can see those components that make up the control plane.

The astute reader may also notice that for each of the components of the Istio control plane there is only a single replica or instance. You may also be asking the question "this appears to be a single point of failure, what happens if these components fail or go down?". You're asking a great question and one that we'll cover throughout the book, however, for the mean time know that the Istio control plane is intended to be deployed in a highly-available architecture (with multiple replicas of each component). In the event of failures of the control plane components, or even the entire control plane, the data plane is resilient enough to continue on for periods of disconnection from the control plane. Istio is implemented to be highly resilient to the myriad of failures that can occur in a distributed system.

The last thing we want to do is verify the installation. We can run the same `verify-install` command post-install to verify it has completed successfully:

Listing 2.7 Verify our Kubernetes cluster is eligible for an Istio installation

```
$  istioctl verify-install
```

This command will compare the install manifest with what is actually installed and alert to any deviations. We should see a listing of the output ending with:

```
Checked 21 custom resource definitions
Checked 1 Istio Deployments
Istio is installed successfully
```

2.2 Getting to know the Istio control plane

In the previous section we did a "demo installation" of Istio which deployed all of the control-plane components as well as ancillary components to Kubernetes. The control plane provides a way for users of the service mesh to control, observe, manage, and configure the mesh. For Istio, the control plane provides the following functions:

- APIs for operators to specify desired routing/resilience behavior
- APIs for the data plane to consume localized configuration
- Service discovery abstraction for the data plane
- APIs for specifying usage policies
- Certificate issuance and rotation
- Assigning workload identity
- Unified telemetry collection
- Service-proxy sidecar injection
- Specifying network boundaries and how to access them

The bulk of these responsibilities are implemented in a single component of the control plane called `istiod`.

2.2.1 Istiod

Istio's control plane responsibilities are implemented in the `istiod` component. Istiod is responsible for taking higher-level Istio configurations specified by the user/operator and turning them into proxy-specific configurations for each service proxy. Istiod does this by exposing APIs for both the user/operator and APIs for the data plane. Istiod is platform agnostic and leverages platform-specific services such as service registries and configuration formats through the use of platform adapters. For example, running Istiod in Kubernetes like we're doing here, leverages the Kubernetes-native way of doing service registration and discovery. It uses its Kubernetes adapter to interact with the Kubernetes API to watch for changes to the pods/service/configuration constructs. By abstracting away the underlying platform implementation details, Istio's control plane can be run in multiple environments without applications having to know or care from a service-mesh perspective.

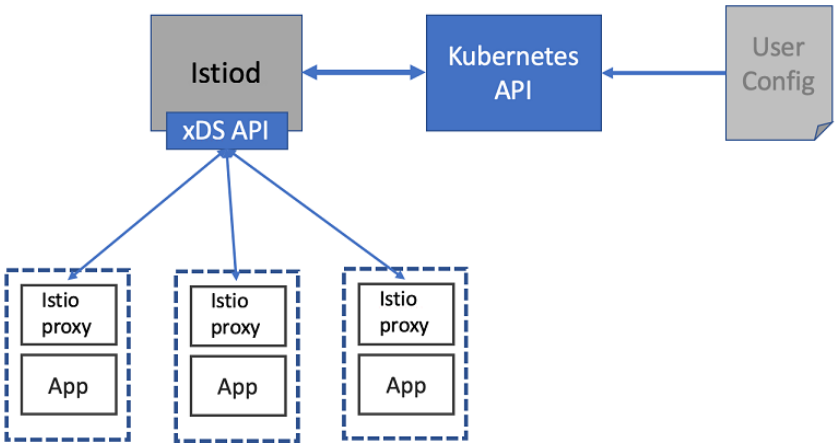


Figure 2.1 Istio control plane: understanding how Istiod takes configuration from operators and exposes to the data plane (istio proxies)

For example, through configuration resources we can specify how traffic is allowed into the cluster, what versions of services should be used based on a request's details, how to shift traffic when doing a new deployment, and how callers of a service should treat resiliency aspects like timeouts, retries, and circuit breaking. Istiod takes these configurations, interprets them, and exposes them as service-proxy-specific configurations. Istio by default uses Envoy as its service proxy, so these configurations are exposed as Envoy configurations. For example, for a service trying to talk to the `catalog` service, we may wish to send traffic to v2 of a service if it has a header `x-dark-launch` in its request. We can express that for Istio with this configuration:

Listing 2.8 Example Istio configuration resource for controlling traffic routing

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: catalog-service
spec:
  hosts:
  - catalog.prod.svc.cluster.local
  http:
  - match:   ❶
    - headers:
        x-dark-lauch:
          exact: "v2"   ❷
    route:
    - destination:   ❸
        host: catalog.prod.svc.cluster.local
        subset: v2
  - route:
    - destination:   ❹
        host: catalog.prod.svc.cluster.local
        subset: v1
```

❶ Request matching

❷ Exact match of header

❸ Where to route on match

❹ All other traffic

For the moment, don't worry about the specifics of the configuration as it's really just to illustrate that this configuration gets exposed to the data plane as proxy-specific configuration. The above configuration specifies that, based on header matching (1) , we would like to route a request to the `v2` deployment (3) of the `catalog` service when there is a header `x-dark-launch` that equals `v2` (2) and for all other requests, we will route to `v1` of the `catalog` service (4). As an operator of Istio running on Kubernetes, we would create this configuration using tools like `kubectl`. For example, if this configuration is stored in a file named `catalog-service.yaml` we could create it in Istio like:

```
$  kubectl apply -f catalog-service.yaml
```

We'll dig deeper into understanding exactly what this configuration does later in this chapter when we get hands on. For now, just know that configuring Istio traffic routing rules will use a similar pattern: describe intent in Istio resource files (yaml) and pass it to the Kubernetes API.

NOTE **Istio uses Kubernetes Custom Resources when deployed on Kubernetes**

Istio's configuration resources are implemented as Kubernetes Custom Resource when running on Kubernetes. Custom Resource Definitions (CRDs) are used to extend the native Kubernetes API to add new functionality to a Kubernetes cluster without having to modify any Kubernetes code. In the case of Istio, we can use Istio's CRs to add Istio functionality to a Kubernetes cluster and use native Kubernetes tools to apply/create/delete the resources. Istio implements a controller that watches for these new CRs to be added and reacts to them accordingly. For example, when we create the `VirtualService` CR from above, Istio recognizes this as an Istio configuration object and translates this into Envoy's native xDS protocol and configuration format.

Istio translates Istio-specific configuration objects, like the `VirtualService` we saw in the previous listing, and translates it into Envoy's native configuration. Istiod will expose this configuration intent to the service proxies as Envoy configuration through its data-plane API like this:

Listing 2.9 Example Istio configuration resource for controlling traffic routing

```
"domains": [
        "catalog.prod.svc.cluster.local"
    ],
"name": "catalog.prod.svc.cluster.local:80",
"routes": [
        {
            "match": {
                    "headers": [
                        {
                                "name": "x-dark-lauch",
                                "value": "v2"
                            }
                        ],
                    "prefix": "/"
                },
            "route": {
                    "cluster": "outbound|80|v2|catalog.prod.svc.cluster.local",
                    "use_websocket": false
                }
            },
        {

            "match": {
                    "prefix": "/"
                },
            "route": {
                    "cluster": "outbound|80|v1|catalog.prod.svc.cluster.local",
                    "use_websocket": false
                }
            }
    ]
```

This data-plane API exposed by Istiod implements Envoy's "discovery APIs". These discovery

APIs, like those for service discovery (Listener Discovery Service - LDS), endpoints (Endpoint Discovery Service - EDS), or routing rules (Route Discovery Service - RDS) are known as the **xDS** APIs. These APIs allow the data plane to separate how it gets configured and dynamically adapt its behavior without having to stop and reload. We'll cover more of these `xDS` APIs from the perspective of Envoy proxy in Chapter 3.

By using this data-plane API, Istio can use proxies not based on Envoy as well. As long as an alternative proxy can use the xDS APIs, Istio can be extended to use other service proxies.

IDENTITY MANAGEMENT

With the Istio service mesh, service proxies run alongside each of the instances of our application and all application traffic goes through this proxy. When an application wishes to issue a request to another service, the proxies on both the sender and receiver actually end up talking to each other directly.

One of Istio's core features is the ability assign identity to each workload instance and encrypt the transport for calls between two services since it sits at both ends of the request path. To do this, Istio uses X.509 certificates to encrypt the traffic. In fact, workload identity is embedded in these certificates following the SPIFFE (Secure Production Identity Framework For Everyone - spiffe.io) which gives the ability to provide strong mutual authentication (mTLS) without the applications having to be aware of certificates, public/private keys, etc. Istiod handles attestation, issuance/mounting of the certificates, and rotation of the certificates used to enable this form of security. We'll cover security in Chapter 8.

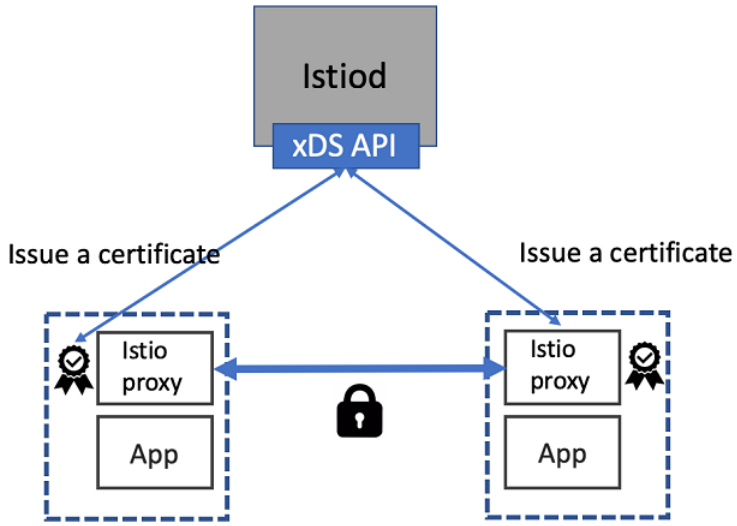


Figure 2.2 Istio control plane: Security with Istiod to manage certificates

2.2.2 Ingress and Egress gateway

For our applications and services to provide anything meaningful, they're going to need to interact with applications that live outside of our cluster. That could be existing monolith applications, off-the-shelf software, messaging queues, databases, and 3rd party partner systems. To do this, operators will need to configure Istio to allow traffic into the cluster and be very specific about what traffic is allowed to leave the cluster. Modeling and understanding what traffic is allowed into and out of the cluster is both good practice as well as improves our security posture.

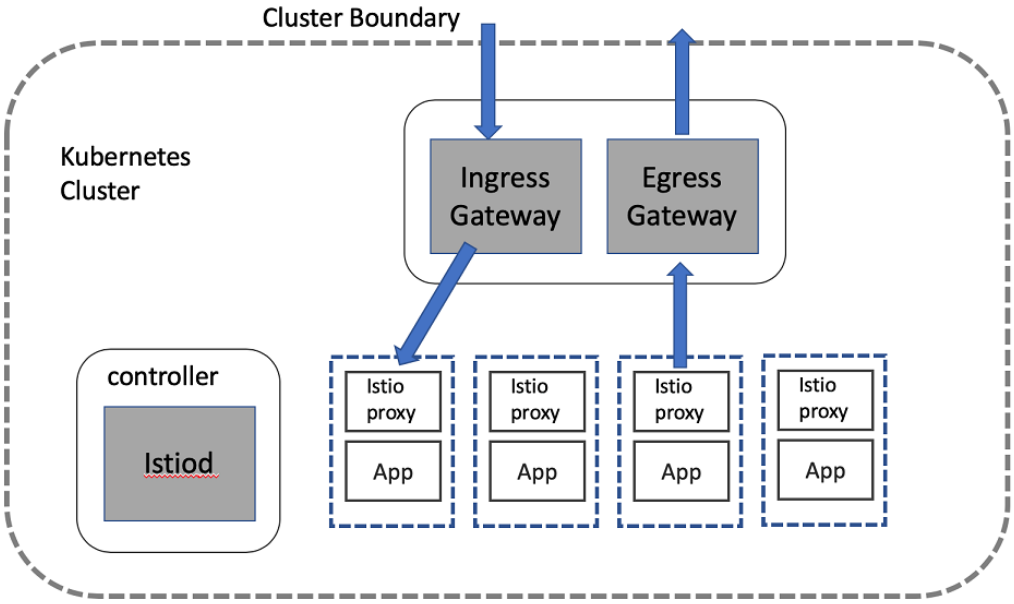


Figure 2.3 Istio control plane: Incoming and Outgoing traffic with Istio Ingress Gateway

The Istio components that provide this functionality are the `istio-ingressgateway` and `istio-egressgateway` as seen in the list of control-plane components from the previous section. Both of these components are really just Envoy proxies that can understand Istio configurations. We'll cover these components in more detail in the next chapter. Although these are not technically part of the control plane, they are instrumental in any real-world usage of a service mesh. These components reside in the data plane and are configured very similarly to istio service proxies that live with the applications. The only real difference is that they're independent of any application workload and are really just to let traffic into and out of the cluster. In future chapters, we'll see how these components play a role in combining clusters and even clouds.

2.3 Deploy your first application in the service mesh

Conference Outfitters (conferenceoutfitters.io), a new startup looking to provide high-quality technology SWAG clothing (think t-shirts, sweatshirts, backpacks, etc) is redoing their website and the systems that power inventory and checkout. They've decided to use Kubernetes as the core of their deployment platform and to build their applications to the Kubernetes API and not a specific cloud vendor. They're looking to solve some of the challenges of service communication in a cloud environment so when their head architect found out about Istio, they decided to leverage it. The Conference Outfitter's application is an online web store comprised of typical enterprise application services. We'll walk through the components that make up the store, but for this first look at Istio's functionality, we'll focus on a smaller subset of the components. The web store is comprised of a web-facing user interface made with AngularJS and NodeJS which communicates with a gateway service that handles coordinating with a backend catalog service and inventory service. For this first example, we'll just deploy the gateway and catalog service.

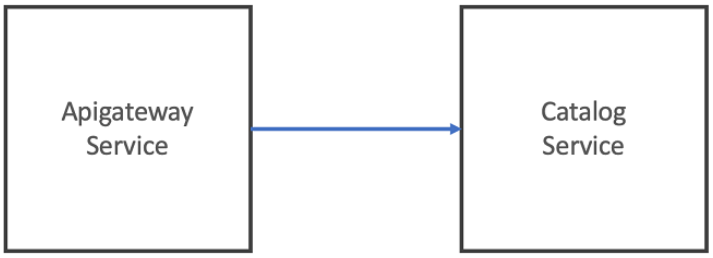


Figure 2.4 Sample application consisting of apigateway and catalog services

To get the source code for this example, download it from the website (http://istioinaction.io) or clone it from https://github.com/istioinaction/book-source-code. In the `install` directory, you should see the Kubernetes resource files that describe the deployment of our components. First thing we want to do is create a namespace in Kubernetes in which we'll deploy our samples:

```
$  kubectl create namespace istioinaction
$  kubectl config set-context $(kubectl config current-context) \
 --namespace=istioinaction
```

Now that we're in the `istioinaction` Kubernetes namespace, let's take a look at what we're going to deploy. The Kubernetes resource files for the `catalog-service` can be found in the `$SRC_BASE/services/catalog/kubernetes/catalog.yaml` file and looks similar to this:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: catalog
  name: catalog
spec:
  ports:
  - name: http
    port: 80
    protocol: TCP
    targetPort: 3000
```

```
  selector:
    app: catalog
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    app: catalog
    version: v1
  name: catalog
spec:
  replicas: 1
  selector:
    matchLabels:
      app: catalog
      version: v1
  template:
    metadata:
      labels:
        app: catalog
        version: v1
    spec:
      containers:
      - env:
        - name: KUBERNETES_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
        image: istioinaction/catalog:latest
        imagePullPolicy: IfNotPresent
        name: catalog
        ports:
        - containerPort: 3000
          name: http
          protocol: TCP
        securityContext:
          privileged: false
```

Before we deploy this, however, we want to inject the Istio service proxy so that this service can participate in the service mesh. From the `coolstore` directory of the source code, run the following command using the `istioctl` command we introduced earlier:

```
$  istioctl kube-inject -f services/catalog/kubernetes/catalog.yaml
```

The `istioctl kube-inject` command takes a Kubernetes resource file and enriches it with the sidecar deployment of the Istio service proxy. Recall from Chapter 1, a **sidecar** deployment is one that packages a complementing process alongside the main application process to work together to deliver some functionality. In the case of Istio, the sidecar process is the service proxy and the main application is your application code. If you look through the output of the previous command you should see the `yaml` now includes a few extra containers as part of the deployment. Most notably, you should see:

```
    - args:
       - proxy
       - sidecar
       - --domain
       - $(POD_NAMESPACE).svc.cluster.local
       - --serviceCluster
       - catalog.$(POD_NAMESPACE)
       - --proxyLogLevel=warning
```

```
        - --proxyComponentLogLevel=misc:error
        - --trust-domain=cluster.local
        - --concurrency
        - "2"
        env:
        - name: JWT_POLICY
          value: first-party-jwt
        - name: PILOT_CERT_PROVIDER
          value: istiod
        - name: CA_ADDR
          value: istiod.istio-system.svc:15012
        - name: POD_NAME
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
...

        image: docker.io/istio/proxyv2:1.7.0
        imagePullPolicy: Always
        name: istio-proxy
```

In Kubernetes, the smallest unit of deployment is called a Pod. A Pod can be one or more containers deployed atomically together. When we ran `kube-inject`, we add another container named 'istio-proxy' to the Pod declaration, though we've not actually deployed anything yet. We can take the yaml file created by the `kube-inject` command and deploy that directly. Let's do that with the `catalog-service`:

```
$  istioctl kube-inject -f services/catalog/kubernetes/catalog.yaml \
| kubectl apply -f -

serviceaccount/catalog created
service/catalog created
deployment.extensions/catalog created
```

If we ask Kubernetes what Pods it has deployed, we should see something like this:

```
$  kubectl get pod
NAME                       READY     STATUS    RESTARTS   AGE
catalog-7c96f7cc66-flm8g   2/2       Running   0          1m
```

If you're running locally on Docker Desktop as outlined earlier, it may take a few moments to download the docker image. After things come to a steady state, you should see the Pod is `Running` as in the previous listing. Also note under the `Ready` column, we see `2/2`. This means there are two containers in the Pod and that two of them are in the Ready state. One of those containers is the application container, `catalog` in this case. The other container is the `istio-proxy` sidecar.

While we wait for the deployment to become ready, let's review an alternative way to inject the sidecar. We manually injected the sidecar using the `istioctl kube-inject` command but we could also automatically inject the sidecar into our deployment using Istio's out-of-the-box sidecar injector. In Kubernetes, this is implemented with an admission webhook but is not

something a user should care about. All the user has to do is label their namespace with `istio-injection=enabled` and when they deploy their application, Istio's sidecar will automatically be injected. We cover this in more detail in Chapter XX.

At this point, we can query the Catalog service within the Kubernetes cluster with `catalog:8080`. Run the following command to verify everything is up and running properly. If it runs successfully and you see the JSON output below, then the service is up and running correctly:

```
$  kubectl run -i --rm --restart=Never dummy \
--image=dockerqa/curl:ubuntu-trusty --command \
-- sh -c 'curl -s catalog/items'

[
  {
    "id": 0,
    "color": "teal",
    "department": "Clothing",
    "name": "Small Metal Shoes",
    "price": "232.00"
  }
]
```

Next, we want to deploy the Gateway API service which provides a facade on top of our other backend services. It also exposes an API that ends up calling the catalog service which we just deployed and verified. Let's follow the same steps to inject the Istio proxy into the Gateway deployment resource and deploy the Gateway:

```
$  istioctl kube-inject -f services/apigateway/kubernetes/apigateway.yaml \
| kubectl apply -f -

serviceaccount/apigateway created
service/apigateway created
deployment.extensions/apigateway created
```

Now if we list the Pods in our Kubernetes cluster, we should see our new Gateway deployment up and running with `2/2` containers running:

```
$  kubectl get pod
NAME                        READY     STATUS    RESTARTS   AGE
apigateway-b689f9c59-8x9ds 2/2       Running   0          47s
catalog-7c96f7cc66-flm8g   2/2       Running   0          1h
```

Lastly, let's call the new Gateway service and verify it works:

```
$  kubectl run -i --rm --restart=Never dummy --image=dockerqa/curl:ubuntu-trusty \
--command -- sh -c 'curl -s apigateway/api/catalog'
```

If that command completes correctly, you should see the same JSON response in the previous call when we called the `catalog-service` directly.

So far all we've done is deploy the `catalog` service and `apigateway` service with the Istio service proxies. Each service has its own proxy and all traffic to or from the individual services

go through each respective proxy.

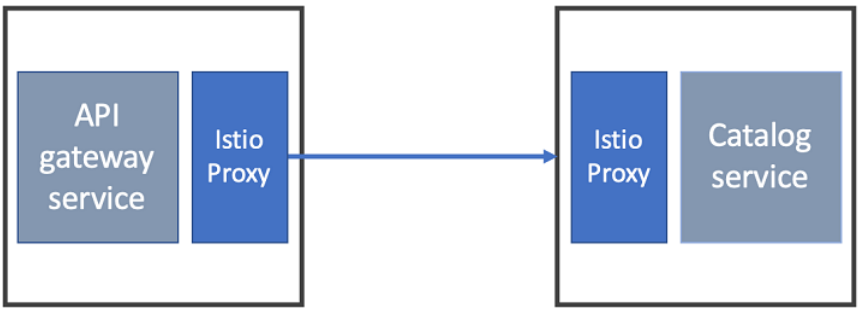


Figure 2.5 Apigateway service calling Catalog service both with istio-proxy injected

2.4 Exploring the power of Istio with resilience, observability, and traffic control

So far we've deployed a couple of sample services, but we have no way of getting traffic into the cluster. With Kubernetes, we typically use a Kubernetes Ingress resource like Nginx, or an API Gateway like Solo.io Gloo, to do that. With Istio, we can use an Istio Gateway resource to get traffic into the cluster so we can call our `apigateway` service. In chapter 4 we'll look at why the out of the box Kubernetes Ingress resource is not sufficient for typical enterprise workloads and how Istio has a concept of a `Gateway` to solve those challenges. For now, we'll use the Istio Gateway to expose our `apigateway` service.

```
$  kubectl apply -f chapters/chapter2/ingress-gateway.yaml
gateway.networking.istio.io/coolstore-gateway created
virtualservice.networking.istio.io/apigateway-virtualservice created
```

At this point, we've made Istio aware of the `apigateway` service at the edge of the Kubernetes cluster and we can call into it and connect to the `apigateway` service. Let's see whether we can reach our service. First we need to get the endpoint on which the Istio Gateway is listening. On Docker for Desktop, it will default to `localhost:80`. On a cloud provider, it will be the public load balancer assigned to the Kubernetes service.

```
$  curl http://localhost:80/api/catalog
```

If you're running on your own Kubernetes cluster, for example on a public cloud, you can find the public cloud's external endpoint by listing the Kubernetes services in the `istio-system` namespace:

```
$  URL=$(kubectl -n istio-system get svc istio-ingressgateway\
-o jsonpath='{.status.loadBalancer.ingress[0].ip}')
$  curl $URL/api/catalog
```

After hitting the endpoint with `curl` like we did here, you should see the same output you saw in the previous steps where we hit the services individually. If you encountered an error up to this point, go back and make sure you successfully completed all of the steps up to this point.

If you still encounter errors, make sure that the Gateway properly has a route set up to our API gateway. To do that, we can use Istio's debugging tools to check the configuration of the Gateway proxy. We can use the same technique to check any Istio proxy deployed with any application, but we'll come back to that. In the meantime check whether your Gateway has a route:

```
$  istioctl -n istio-system proxy-config routes $(make ingress-pod)
```

You should see something similar to this:

```
NOTE: This output only contains routes loaded via RDS.
NAME        DOMAINS     MATCH                  VIRTUAL SERVICE
http.80     *           /*                     apigateway-virtualservice.istioinaction
            *           /healthz/ready*
            *           /stats/prometheus*
```

If not, your best bet is to double check the Gateway and VirtualService got installed:

```
$  kubectl get gateway
$  kubectl get virtualservice
```

2.4.1 Istio observability

Since the Istio service proxy sits in the call path on both sides of the connection (each service has its own service proxy), Istio can collect a lot of telemetry and insight into what's happening between the applications. Istio's service proxy is deployed as a sidecar alongside each application, so the insight it collects is from "out of process" of the application. This means, for the most part, the applications do not need to have library or framework specific implementations to accomplish this level of observability. It's "black box" and is focused on the behavior of the application as the network sees it.

There are two major categories of observability that Istio can collect. The first is top-line metrics or things like requests/second, number of failures, and tail-latency percentiles. Knowing these values can give great insight into where problem areas of a system are starting to arise. Secondly, Istio can facilitate distributed tracing like OpenTracing.io. Istio can send spans to the distributed-tracing backends without the applications having to worry about it. This way, we can dig into what happened during a particular service interaction, where latency occurred and get information about overall call latency. Let's explore these capabilities hands on with our example application.

TOP-LEVEL METRICS

The first capability of Istio we'll take a look at is some of the observability features we can get out of the box. In the previous section, we added two Kubernetes deployments and injected them with the Istio sidecar proxies. Then we added an Istio Gateway so we can reach our service from outside the cluster. To get metrics, we will install Prometheus and Grafana.

Istio by default comes with some sample addons that can install Prometheus and Grafana. Navigate to the Istio installation directory (from earlier this chapter) and check the `$ISTIO_BASE/samples/addons` folder. You should see something like this:

```
total 920
-rw-r--r--@ 1 ceposta  staff    4892 Aug 21 12:00 README.md
drwxr-xr-x@ 4 ceposta  staff     128 Aug 21 12:00 extras
-rw-r--r--@ 1 ceposta  staff  398854 Aug 21 12:00 grafana.yaml
-rw-r--r--@ 1 ceposta  staff    1960 Aug 21 12:00 jaeger.yaml
-rw-r--r--@ 1 ceposta  staff   38508 Aug 21 12:00 kiali.yaml
-rw-r--r--@ 1 ceposta  staff   12951 Aug 21 12:00 prometheus.yaml
```

Let's apply the Prometheus and Grafana addons:

```
$  kubectl apply -f ./samples/addons/prometheus.yaml
$  kubectl apply -f ./samples/addons/grafana.yaml
```

Once these components get installed correctly we should see something like this in the `istio-system` namespace:

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

NAME                                     READY   STATUS    RESTARTS   AGE
grafana-767c5487d6-cr5n7                 1/1     Running   0          2m25s
istio-egressgateway-55d547456b-q2ldq     1/1     Running   0          2d4h
istio-ingressgateway-7654895f97-2pb62    1/1     Running   0          2d4h
istiod-5b9d44c58b-vvrpb                  1/1     Running   0          2d4h
prometheus-788c945c9c-4k799              2/2     Running   0          2m30s
```

Let's use `istioctl` to port forward Grafana to our local machine so we can see the dashboards:

```
$  istioctl dashboard grafana -p 3000
http://localhost:3000
```

Now if we use our browser and go to http://localhost:3000, we should be taken to a Grafana home screen. In the upper left-hand corner, you can select the "Home" dashboard to expose a drop-down or a list of other dashboards we can switch to.

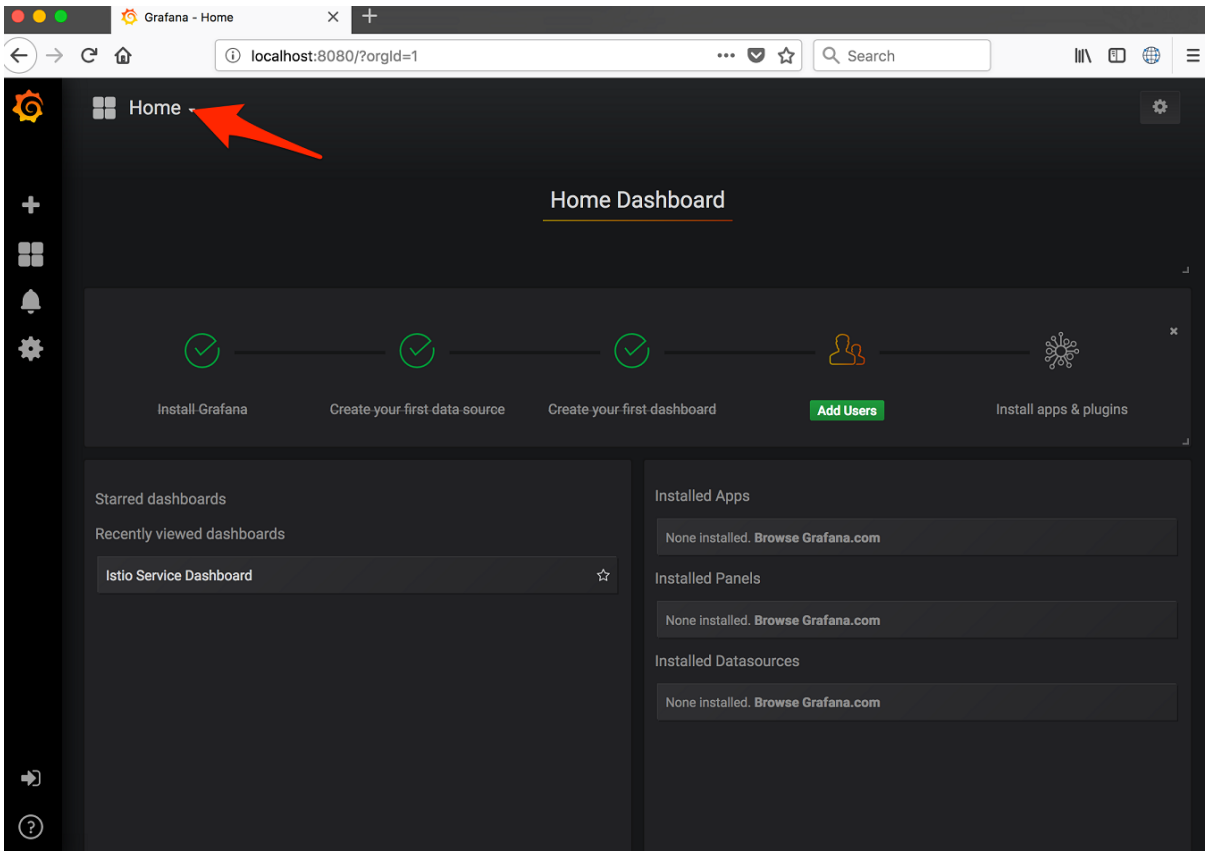


Figure 2.6 Grafana home screen

Istio has a set of out-of-the-box dashboards that give some basic details about the services running in Istio. With the out-of-the-box dashboards, we can see those services we have installed and running in the mesh and some of the Istio control-plane components. In the list of dashboards is a dashboard named "Istio Service Dashboard". Click that one.

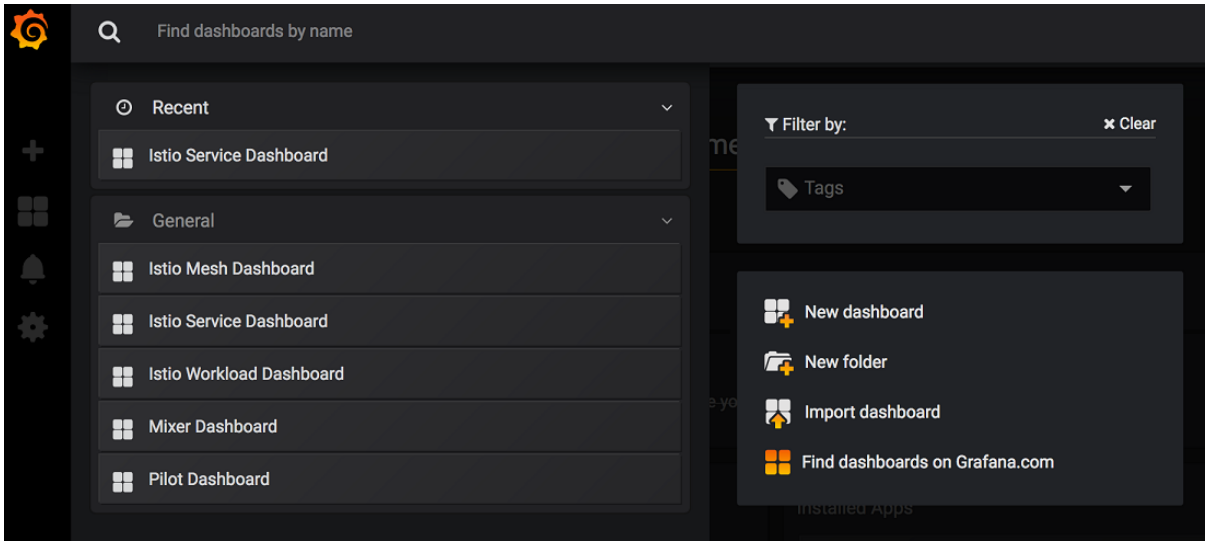


Figure 2.7 List of installed Grafana dashboards including the Istio out of the box dashboards

The dashboard should show some top-level metrics of the particular service selected. In the `Service` dropdown box toward the top of the dashboard, make sure the `apigateway.istioinaction.svc.cluster.local` service is selected. It should look similar to this:

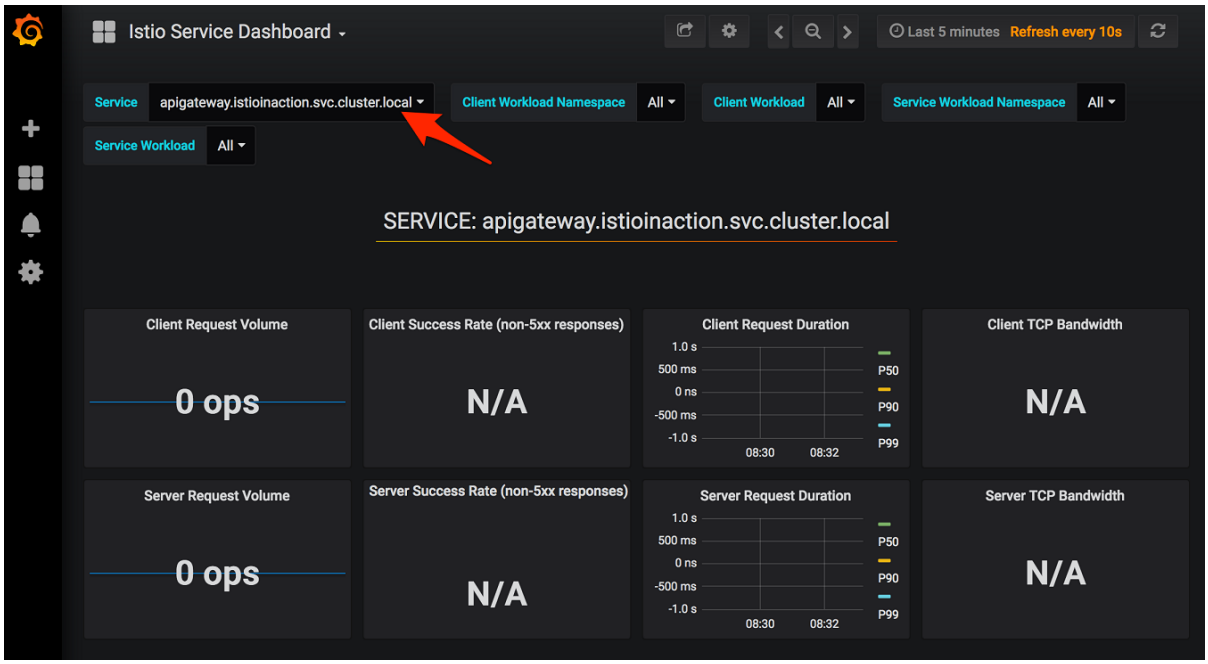


Figure 2.8 Dashboard for the apigateway service

We see metrics like `Client Request Volume` and `Client Success Rate`, but the values are mostly empty or "N/A". In your command-line shell, let's add some traffic to the services and watch what happens:

```
$  while true; do curl http://localhost/api/catalog; sleep .5; done
```

You can press CTRL+C to exit out of this while loop.

If you go back and look at the Grafana dashboard, now we should see some interesting traffic:

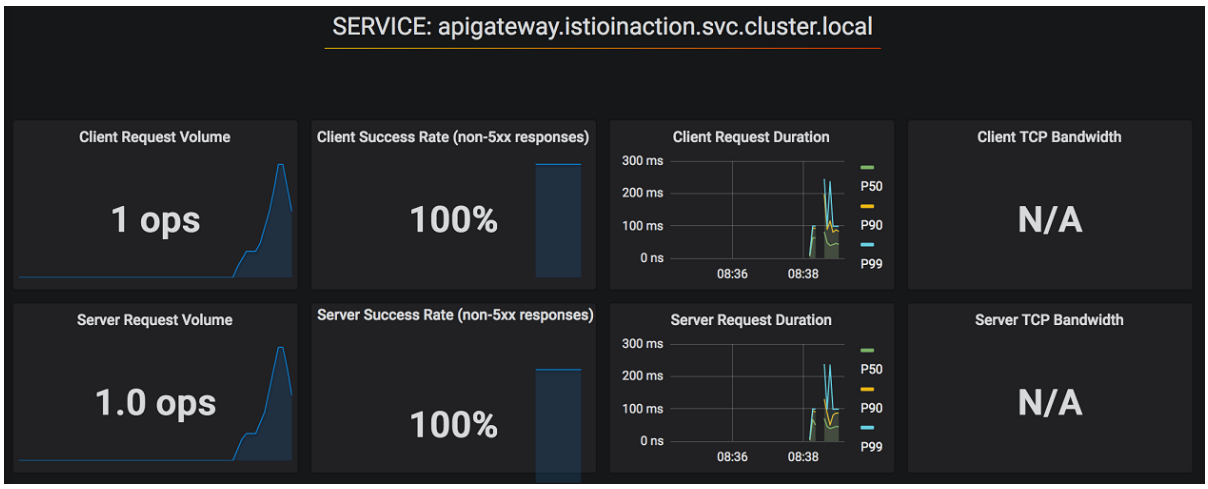


Figure 2.9 Top-level metrics for our apigateway service as seen in Grafana

We can see that our service received some traffic, we had 100% success rate, and we can even see the P50, P90, and P99 tail latencies we experienced. Scroll down the dashboard and you can see some more interesting metric collection about what services and clients are calling the `apigateway` service and what that behavior looks like.

An important thing to point out here is that we did not add any instrumentation to our application code. Although, we should always heavily instrument our application, what we see here is what the application actually did over the network regardless of what the application **thinks** happened. We are able to observe from a "black-box" perspective how the applications and their collaborators are behaving in the mesh and all we did was add the Istio sidecar proxies. If we want to get more wholistic view of individual calls through the cluster we can look at things like distributed tracing to follow a single request as it hits multiple services.

DISTRIBUTED TRACING WITH OPENTRACING

We can leverage Istio to take care of most of the heavy lifting to get distributed tracing out of the box. One of the addons that come with Istio's installation is the Jaeger tracing dashboard. Let's install it from the same directory of `addons` as the Prometheus and Grafana addons:

```
$  kubectl apply -f ./samples/addons/jaeger.yaml
```

After installing Jaeger, let's open the dashboard:

```
$  istioctl dashboard jaeger -p 16686

http://localhost:16686
```

Now, let's use our web browser to navigate to http://localhost:16686 which should take us to the Jaeger web console:

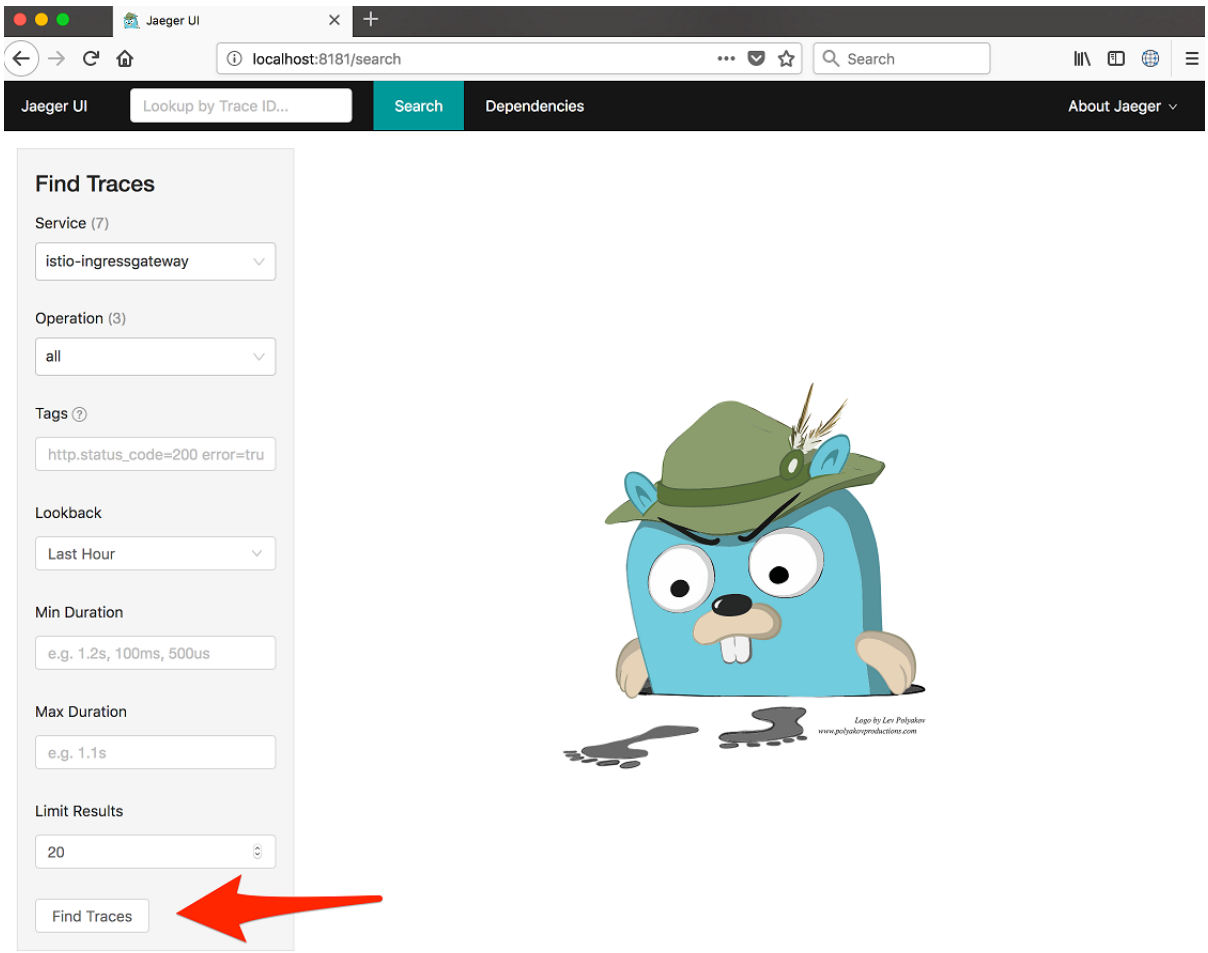


Figure 2.10 Jaeger distributed-tracing engine web console home page

The service in the top left-hand pane "Service" drop-down should be the `istio-ingressgateway` service. If it's not, then click the drop-down and select the `istio-ingressgateway` option. Then click "Find Traces" on the lower left-hand of the side pane. You should see some distributed tracing entries. If you don't, re-run the traffic-generation client from your command line:

```
$  while true; do curl http://localhost/api/catalog; sleep .5; done
```

You can press CTRL+C to exit out of this while loop.

You should be able to see the most recent calls that came into the cluster and the distributed tracing spans they generated. If you click on one of the span entries, you can see the details of a particular call. You can see that from the `istio-ingresssgateway`, the call went to the `apigateway` and then the `catalog` service.

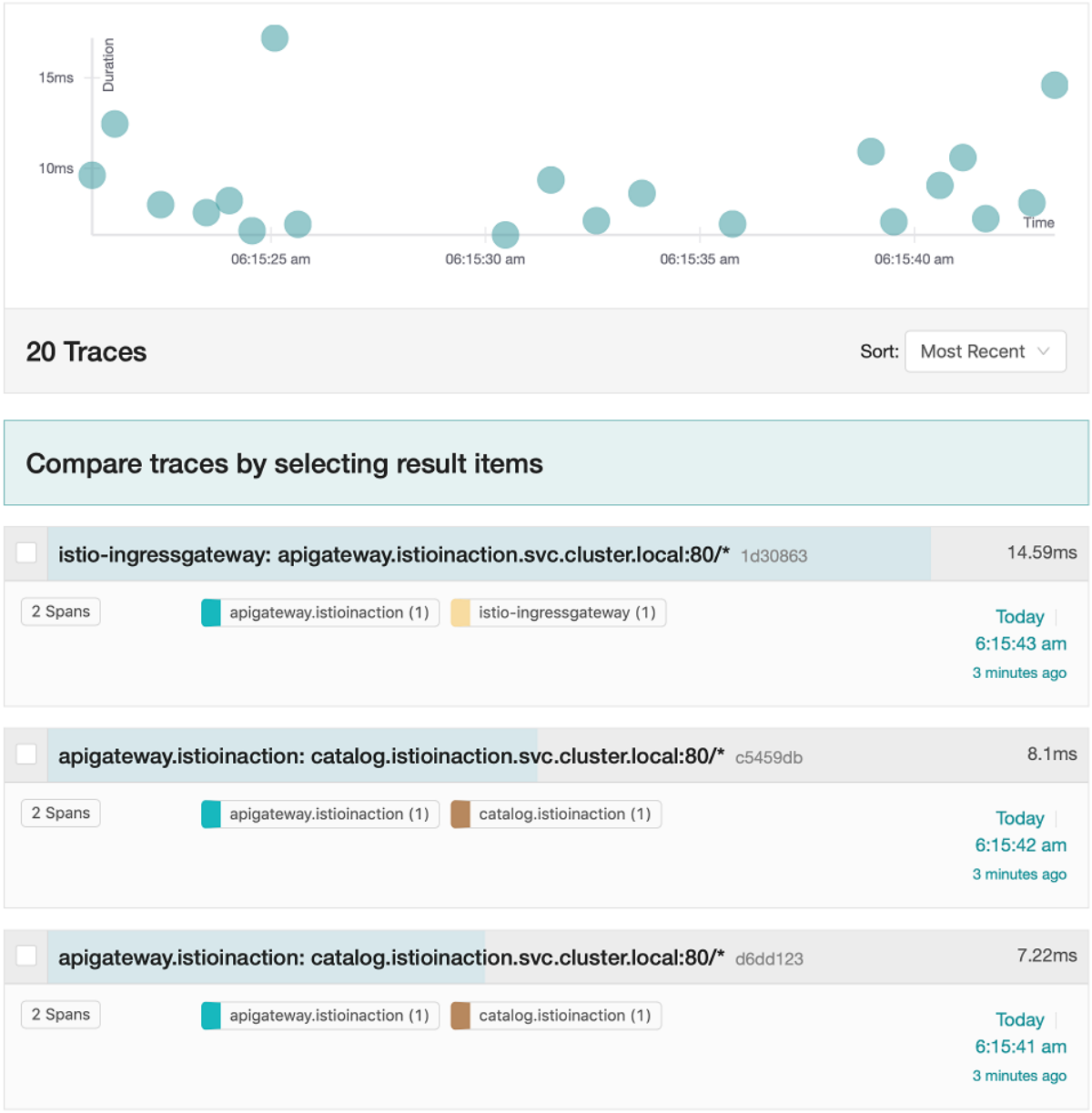


Figure 2.11 A collection of distributed traces gathered using Istio

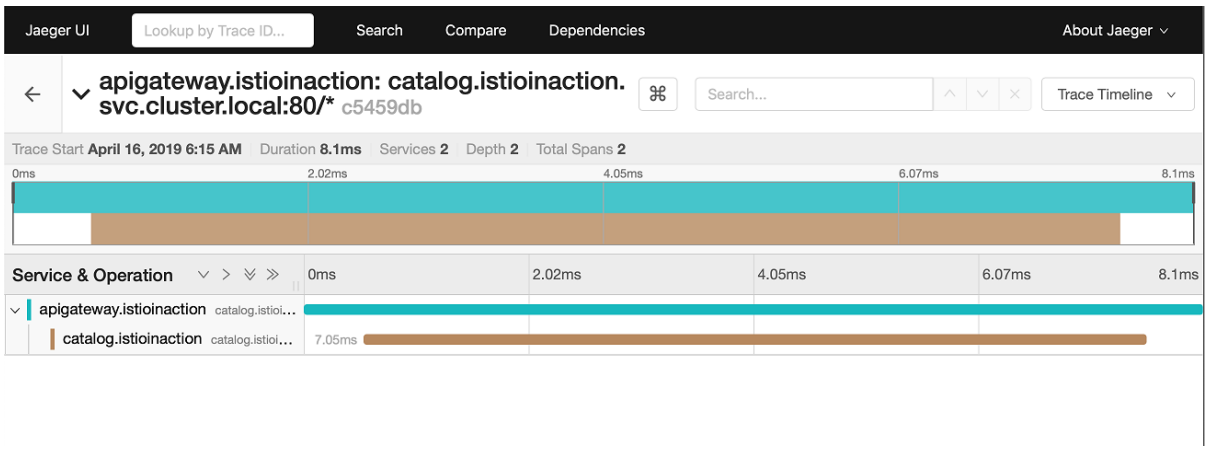


Figure 2.12 Detailed spans for a specific call

In subsequent chapters, we'll explore how this all works, but for now you should understand that the Istio service proxy propagated the tracing IDs and metadata between services and also sent tracing span information to a tracing engine (like Zipkin or Jaeger). We don't want to gloss over the fact, however, that the application does have to play some small part in this overall capability.

Although Istio can propagate the traces **between** services and to the tracing engine, the applications themselves are responsible for propagating the tracing metadata **inside** the applications. The tracing metadata usually consist of a set of HTTP headers (if doing HTTP) and it's up to the application to correlate the incoming headers with any outgoing requests. Said another way, Istio cannot know what happens inside a particular service or application so it cannot know that for a specific request that came in, it should be associated with a specific outgoing request (causation). It relies on the application to know that and properly inject the headers into any outgoing request. From there, Istio can capture those spans and send them to the tracing engine.

2.4.2 Istio for resiliency

As we've discussed, applications that communicate over the network to help complete their business logic must be aware of and account for the fallacies of distributed computing. Namely, they need to deal with network unpredictability. In the past, we tried to code a lot of this networking workaround code into our applications by doing things like retries, timeouts, circuit-breaking, etc. Istio can alleviate application developers from having to code this networking code directly into their application and provide a consistent, default, expectation of resilience for all of the applications in the service mesh.

One such resiliency aspect is retrying requests in the midst of intermittent/transient network errors. For example, if the network experiences some hardware failures but the network has enough redundant paths, we may experience these errors, but the application can continue if they just try again. In our example architectures, we'll simulate this by driving the behavior from our `Catalog` service.

If we make a call to our `apigateway` service endpoint like we did the in previous section, we see that the call returns successfully. If we want all calls to fail, however, we can use a script that injects some bad behavior into the application. If we run the following command from the root of our source code, we can cause all calls to fail with an HTTP 500 100% of the time:

Listing 2.10 Using a built-in API to inject bad behavior into the catalog service 100% of the time

```
$  ./bin/chaos.sh 500 100
```

If you try running the command from Listing X, you can see that an HTTP 500 gets returned:

```
$  curl -v http://localhost/api/catalog
*   Trying 192.168.64.67...
* TCP_NODELAY set
* Connected to 192.168.64.67 (192.168.64.67) port 31380 (#0)
> GET /api/catalog HTTP/1.1
> Host: 192.168.64.67:31380
> User-Agent: curl/7.54.0
> Accept: */*
>
< HTTP/1.1 500 Internal Server Error
< content-type: text/plain; charset=utf-8
< x-content-type-options: nosniff
< date: Wed, 17 Apr 2019 00:13:16 GMT
< content-length: 30
< x-envoy-upstream-service-time: 4
< server: istio-envoy
<
error calling Catalog service
* Connection #0 to host 192.168.64.67 left intact
```

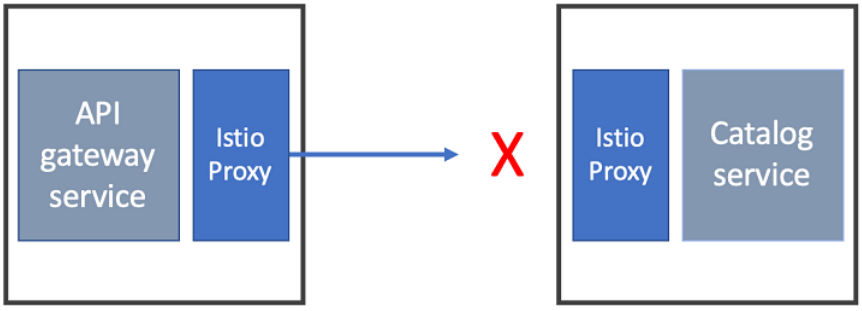


Figure 2.13 Catalog service can be provoked to fail by injecting bad behavior

To demonstrate Istio's ability to automatically perform a retry for your application, let's issue a command to generate errors 50% of the time when we call our `apigateway` service endpoint.

Listing 2.11 Using a built-in API to inject bad behavior into the catalog service 50% of the time

```
$  ./bin/chaos.sh 500 50
```

Listing 2.12 Test the service responses

```
$  while true; do curl http://localhost/api/catalog ; \
sleep .5; done
```

You can press CTRL+C to exit out of this while loop.

The output from that command should be intermittent success and failures from the `apigateway` service. Actually the failures are being caused when the `apigateway` service talks with the `catalog` service (it's the catalog service misbehaving). Let's see how we can use Istio to make the network more resilient between `apigateway` and `catalog`.

Using an Istio `VirtualService` we can specify rules about interacting with services in the mesh. The following is an example of the `catalog` `VirtualService` definition:

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

With this definition, we specify for requests to the `catalog` service to be eligible for retry up to three times with each try having a timeout of two seconds. If we put this rule into place, we can leverage Istio to automatically retry when we experience failures (like we did in the previous step). Let's create this rule and re-run our test client script:

```
$  kubectl apply -f chapters/chapter2/catalog-virtualservice.yaml
virtualservice.networking.istio.io/catalog created
```

Now try running the client script again:

```
$  while true; do curl http://localhost/api/catalog ; \
sleep .5; done
```

You can press CTRL+C to exit out of this while loop.

You should see no more exceptions bubbling up to the client. Using Istio, and without touching any application code, we're able to add a level of resilience when communicating over the network.

Let's disable the failures in our catalog service:

```
$  ./bin/chaos.sh 500 delete
```

This should disable any misbehaving responses from the `catalog` service.

2.4.3 Istio for traffic routing

The last Istio capability we'll look at in this chapter is the ability to have very fine-grained control over requests in the service mesh no matter how deep they are in a call graph. So far we've taken a look at a simple architecture consisting of the `apigateway` service providing a simple facade over any of the services it communicates in the backend. The one service it talks to at this moment is the `catalog` service. Let's say we want to add some new functionality the the `catalog` service. For this example, we'll enhance the functionality to add a flag to the payload to indicate whether a discount is available for a particular item in the catalog. We want to expose this information to end callers (like a user interface capable of understanding this flag, or an apigateway service that can then use that flag to decide whether to enrich an item with more **discount** information, etc) that are able to handle this change.

If we take a look at the response from the `catalog` service, we can see that v1 of `catalog` has the following properties in its response:

```
{
  "id": 0,
  "color": "teal",
  "department": "Clothing",
  "name": "Small Metal Shoes",
  "price": "232.00"
}
```

For v2 of the `catalog` service, we will add a new property named `imageUrl`:

```
{
  "id": 0,
  "color": "teal",
  "department": "Clothing",
  "name": "Small Metal Shoes",
  "price": "232.00",
  "imageUrl": "http://lorempixel.com/640/480"
}
```

When we make requests to the `catalog` service, for the version v2, we'll expect this new `imageUrl` field in the response.

In principle, what we want to do is deploy the new version of our `catalog` service, but we want to finely control to whom this gets exposed (released). It's important to be able to separate **deployment** from **release** in such a way to reduce the chances of breaking things in production and having our paying customers be at the forefront of our risky behavior. Specifically, a deployment is when we bring new code to production. When it's in production we can run tests against it, and evaluate whether it's fit for any production usage. When we **release** code, we bring live traffic to it. We can exercise a phased approach to a release wherin we only route certain classes of traffic to the new deployment. One such strategy could be to only route internal employees to new deployments and watching how the deployment and overall system behaves. We could then graduate the traffic up to non-paying customers, silver-level customers, and so on.

We'll cover more of this principle in Chapter XX when we look deeper at Istio's request-routing functionality.

Using Istio, we can finely control which traffic goes to v1 of our service and which requests should go to v2 of our service. We'll use a concept from Istio called a `DestinationRule` to split up our services by version like the following:

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

With this `DestinationRule` we denote two different versions of the `catalog` service. We specify the group based on labels of the deployments in Kubernetes. From above, we can see that any Kubernetes Pods labeled with `version: v2` will belong to the `v2` group of the catalog service that Istio knows about. Before we actually create the `DestinationRule`, let's deploy a second version of our `catalog` service

```
$  istioctl kube-inject -f ./services/catalog/kubernetes/catalog-deployment-v2.yaml \
| kubectl apply -f -
deployment.extensions/catalog-v2 created
```

When the new deployment is ready, you should see a second `catalog` pod like this:

```
$  kubectl get pod
NAME                          READY     STATUS    RESTARTS   AGE
apigateway-bd97b9bb9-q9g46    2/2       Running   0          17m
catalog-5dc749fd84-fwcl8      2/2       Running   0          10m
catalog-v2-64d758d964-rldc7   2/2       Running   0          38s
```

If we call our service a handful of times, we should notice that some of the responses have our new `imageUrl` field in the response, and some do not. By default, Kubernetes is able to do a limited form of load balancing between the two different versions. However, since we want to safely deploy software to production without impacting end users and also have the option to test it in production before releasing it, we want to restrict traffic to the v1 version of `catalog` for now.

The first thing we'll do, is let Istio know how to identify different versions of our catalog service. We'll use the `DestinationRule` from the previous listing to do that:

```
$  kubectl apply -f chapters/chapter2/catalog-destinationrule.yaml
destinationrule.networking.istio.io/catalog created
```

Next we'll create a rule in the `catalog` `VirtualService` that says to route all traffic to v1 of catalog:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: catalog
spec:
  hosts:
  - catalog
  http:
  - route:
    - destination:
        host: catalog
        subset: version-v1
```

Let's update the `catalog` virtual service with our new traffic routing rule:

```
$   kubectl apply -f chapters/chapter2/catalog-virtualservice-all-v1.yaml
virtualservice.networking.istio.io/catalog created
```

Now if we send traffic to our `apigateway` endpoint, we should see only v1 responses:

```
$  while true; do curl http://localhost/api/catalog; sleep .5; done
```

You can press CTRL+C to exit out of this while loop.

Let's say for certain users, we do want to expose the functionality of v2 of the `catalog` service. Istio gives us the power to control the routing for individual requests and match on things like request path, headers, cookies, etc. Let's say for users that pass in a specific header, we will allow them to hit the new `catalog` v2 service. Using a revised `VirtualService` definition for the `catalog` service, let's match on a header called `x-dark-launch` and for any requests with that header, send them to `catalog` version two:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: catalog
spec:
  hosts:
  - catalog
  http:
  - match:     ❶
    - headers:
        x-dark-launch:
          exact: "v2"
    route:     ❷
    - destination:
        host: catalog
        subset: version-v2
  - route:
    - destination:   ❸
        host: catalog
        subset: version-v1
```

❶ A match clause

❷ A route to v2 that gets activated when matched

❸ Default route

Let's create this new routing rule inside our `VirtualService`:

```
$  kubectl apply -f chapters/chapter2/catalog-virtualservice-dark-v2.yaml
virtualservice.networking.istio.io/catalog configured
```

Try calling the `apigateway` endpoint once again. You should notice only v1 responses from the `catalog` service in the response.

```
$  while true; do curl http://localhost/api/catalog; sleep .5; done
```

Now, let's call the endpoint with our new special header `x-dark-launch`

```
$  curl http://localhost/api/catalog -H "x-dark-launch: v2"
[
  {
    "id": 0,
    "color": "teal",
    "department": "Clothing",
    "name": "Small Metal Shoes",
    "price": "232.00",
    "imageUrl": "http://lorempixel.com/640/480"
  }
]
```

When we include the `x-dark-launch: v2` header in our call, we will see the response from the `catalog-v2` service while all other traffic will go to `catalog-v1`. Here we've used Istio to finely control the traffic to our services based on individual requests.

In this next chapter, we're going to take a deeper look at Envoy Proxy, Istio's default data-plane proxy, to understand it as a standalone component and then show how Istio leverages it to achieve the functionality desired by a service mesh.

2.4.4 Clean up and prepare Istio for rest of the book

Before we move on, we will delete the sample applications as we'll re-install the individual components as we go along.

Listing 2.13 Delete the demo applications

```
$  kubectl delete all --all -n istioinaction
```

kubectl delete gateway -A --all kubectl delete virtualservice -A --all

2.5 Summary

- Istio is a powerful, open-source service mesh
- We see how to deploy Istio and supporting components on Kubernetes including
  - Istio Pilot for managing configuration of the cluster
  - Istio Policy for enforcing quota and access-control policy
  - Istio Telemetry for syndicating telemetry to backend systems
  - Istio Gateway for allowing traffic into the cluster
- Kubernetes is a natural fit for Istio
- Applications become part of the mesh by installing the istio sidecar proxy
- Since Istio is in the data plane, it sees and understands all traffic between services
- This gives us powerful capabilities like routing control, resilience, observability, security, etc.

3 Istio's data plane: Envoy Proxy

This chapter covers:

- Understanding stand-alone Envoy Proxy and how it contributes to Istio
- Hands on with Envoy configuration to get an appreciation for how it works
- Envoy's capabilities like traffic routing, resilience, and metric collection are core to a service mesh like Istio
- How to configure Envoy and how Istio makes it easier to do so in a cluster
- Exploration of Envoy's Admin API to get a sense for how to introspect and debug a proxy

When we introduced the idea of a service mesh in Chapter 1, we established the concept of a service proxy and how this proxy is built to understand application-level constructs (e.g., application protocols like HTTP and gRPC) and supplement an application's business logic with non-differentiating application-networking logic. This service proxy runs collocated and out of process with the application and the application talks through the service proxy whenever it wants to communicate with other services. With Istio, Envoy Proxy is the default out-of-the-box service proxy and gets deployed collocated with all application instances participating in the service mesh thus forming the service-mesh data plane. Since Envoy is such a critical component in the data plane, and in the overall service-mesh architecture, we'll spend this chapter getting familiar with Envoy in hopes you better understand Istio and how to dig into specific components when putting together a bigger architecture or when you need to debug or troubleshoot your deployments.

3.1 What is Envoy Proxy

Envoy was developed at Lyft to solve some of the difficult networking problems that crop up when building distributed systems. It was contributed as an open-source project in September 2016 and a year later (September 2017) joined the Cloud Native Computing Foundation (CNCF). Envoy is written in C++ in an effort to increase performance, but more importantly, make it more stable and deterministic at higher load echelons. Envoy was created out of the following two critical principles:

> *The network should be transparent to applications. When network and application problems do occur it should be easy to determine the source of the problem.*
>
> *– Envoy Announcement https://lft.to/2MxcVXl*

Envoy is a proxy, so before we go any further, we should make very clear what a proxy is. A proxy is a intermediary component in a network architecture that inserts itself in the middle of communication to provide additional features like security, privacy, or policy. For example, an internet proxy in your organization is where all traffic intended for the internet first flows. Traffic reaches the proxy, any organizational policy gets applied by the proxy (ie, web sites, protocols, data, etc that cannot be visited) and the proxy then sends the traffic out to the internet.

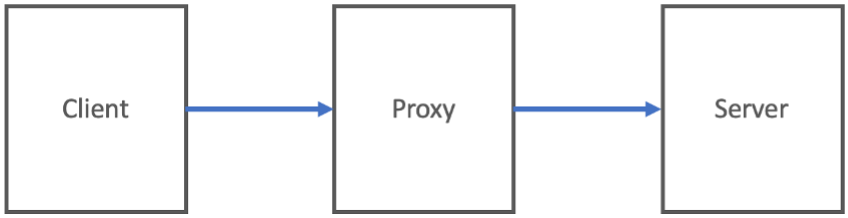


Figure 3.1 A proxy is an intermediary adding additional functionality the flow of traffic

Proxies can simplify what a client needs to know when talking to a service and protect backend services from becoming overloaded. For example, Service A may actually be implemented as a set of identical instances (cluster) in the backend each able to handle a certain amount of load. How should the client know which instance or IP address to use when talking to service A? A proxy can stand in the middle with a single identifier or single IP address and clients can just use that to talk to the service. The proxy then handles load balancing across the instances of the service without the client knowing any details of how things are actually deployed. Another common function of this type of "reverse proxy" is checking health of the instances in the cluster and routing traffic around failing or misbehaving back end instances. This way the proxy can protect the client from having to know and understand which backends are overloaded or are failing.

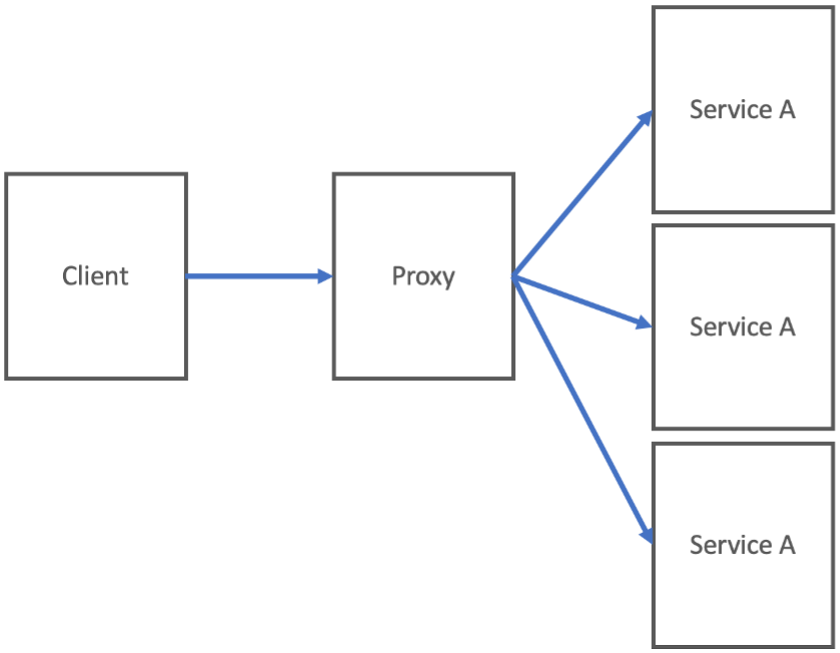


Figure 3.2 A proxy can hide backend topology from clients and implement algorithms to fairly distribute traffic (load balancing)

Envoy proxy is specifically an "application proxy" that we can insert into the request path of our applications to provide things like service discovery, load balancing, and health checking, but Envoy can do more than just load balance connections and shuttle bytes and packets across network cards and routers. We've hinted at some of these more enhanced capabilities in earlier chapters, and we'll cover them more in this chapter. Envoy can understand Layer 7 protocols that an application may speak when communicating with other services. For example, out of the box Envoy understands HTTP 1.1, HTTP 2, gRPC, etc protocols and can add behavior like request-level timeouts, retries, per-retry timeouts, circuit breaking, and other resilience features. Something like this cannot be accomplished with basic connection (L3/L4) level proxies that only understand packets and bytes.

Envoy can also be extended to understand more protocols than just the out of the box defaults. Filters have been written for databases like Mongodb, Dynamodb, and even asynchronous protocols like AMQP. Reliability, and the goal of "network transparency" for applications is a worthwhile endeavor, but just as important if not more so, is the ability to quickly understand what's happening in a distributed architecture especially when things are not working as expected. Since Envoy understands application-level protocols, and application traffic flows through Envoy, the proxy can collect lots of telemetry about the requests flowing through the system, how long they're taking, how many requests certain services are seeing (throughput), and what kind of error rates the services are seeing.

As a proxy, Envoy is designed to be able to shield our developers from networking concerns by running out of process from the applications. This means any application written in any programming language or with any framework can take advantage of these features. Moreover,

although services architectures (SOA, microservices, etc) are the architecture **de jour**, Envoy doesn't really care if you're doing microservices or if you have monoliths and legacy applications written in any language. As long as they speak protocols that Envoy can understand (ie, HTTP), Envoy can be used to provide benefits.

Lastly, Envoy can be used as a proxy at the edge of your cluster (as an ingress point), as a shared proxy for a single host or group of services, and even as a per-service proxy like we see with Istio. With Istio, a single Envoy proxy is deployed per service to achieve the most flexibility, performance, and control. Just because you use one type of deployment pattern (i.e., sidecar service proxy), doesn't mean you cannot also have the edge served with Envoy. In fact, having the proxy be the same implementation at the edge as well as located within the application traffic can make it easier to operate and reason about your infrastructure. As we'll see in the next chapter, Envoy can be used at the edge for ingress and tie into the service mesh to give full control and observation of traffic from the point it enters the cluster all the way down to the individual services in a call graph for a particular request.

3.1.1 Envoy's core features

Envoy has many features useful for inter-service communication. To help understand Envoy's features and capabilities, you should be familiar with Envoy `listeners`, `routes`, and `clusters`:

- Listeners - expose a port to the outside world into which application can connect; for example, a listener on port 8080 would accept traffic and apply any configured behavior to that traffic
- Routes - rules for how to handle traffic that came in on **listeners**; for example, if a request comes in and matches `/catalog`, then direct that traffic to the catalog **cluster**
- Clusters - specific upstream services to which Envoy can direct traffic; for example, `catalog-v1` and `catalog-v2` can be separate clusters and **routes** can specify rules about how traffic can be directed to either v1 or v2 of the catalog service

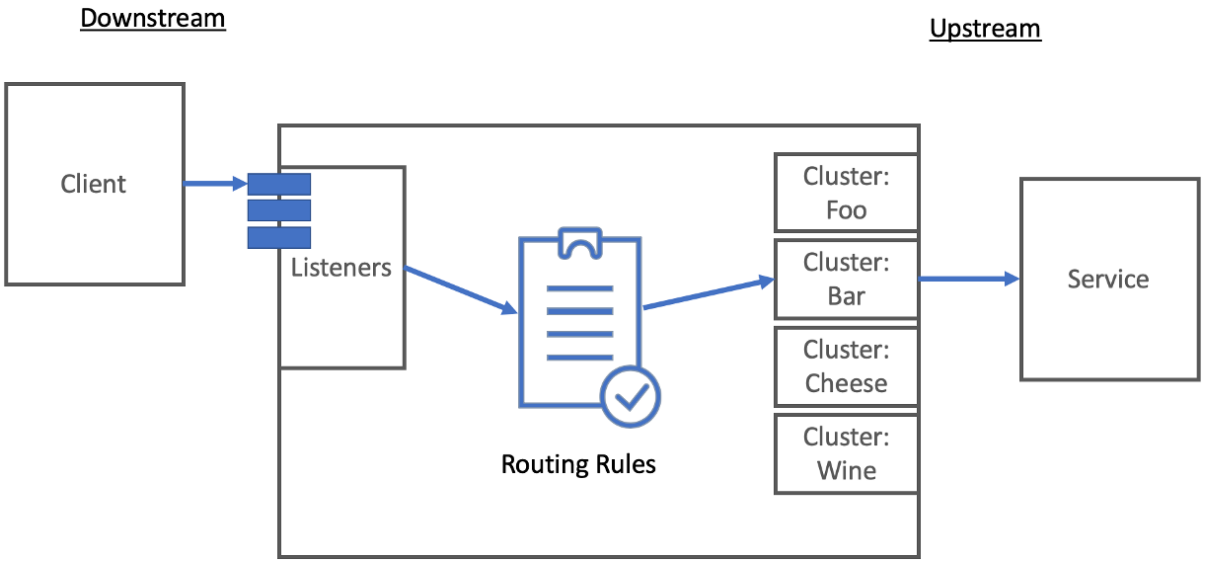


Figure 3.3 A request comes in from a downstream system through the listeners, then goes through the routing rules, and ends up going to a cluster which sends to an upstream service

Envoy uses similar terminology to that of other proxies when conveying traffic directionality. For example, traffic coming into Envoy comes into a `listener` and is coming from a `downstream` system. This traffic gets routed to one of Envoy's `clusters` which is responsible for sending that traffic to an `upstream` system (as shown in Figure 3.3). Downstream to upstream is how traffic flows through Envoy. Now on to some of it's features:

SERVICE DISCOVERY

Instead of using runtime-specific libraries for client-side service discovery, Envoy can do this automatically for an application. By configuring Envoy to look for service endpoints from a simple discovery API, our application can be agnostic to how service endpoints are found. The discovery API is a simple REST API that can be used to wrap other common service-discovery APIs (like HashiCorp Consul, Apache Zookeeper, Netflix Eureka, etc). Istio's control plane implements this API out of the box.

Envoy specifically is built to rely on **eventually consistent** updates to the service-discovery catalog. This means in a distributed system we cannot expect to know the **exact** status of all services with which we can communicate and whether they're available or not. The best we can do is use the knowledge at hand, employ active and passive health checking, and expect those results may not be the most up-to-date (nor could they be).

Istio abstracts away a lot of this detail by providing a higher-level set of resources that drive the configuration of Envoy's service-discovery mechanisms. We'll be taking a closer look throughout the book.

LOAD BALANCING

Envoy implements a few advanced load-balancing algorithms of which applications can take advantage. One of the more interesting capabilities of Envoy's load balancing algorithms is the ability to do **zone-aware load balancing**. In this situation, Envoy will be smart enough to keep traffic from crossing any zone boundaries unless it meets certain criteria and will provide a better balance of traffic. For example, Service A may request data from Service B and Envoy will make sure that requests are routed to instances of Service B in the same zone as Service A unless doing so would create an unbalanced situation. Envoy provides out of the box load balancing algorithms for the following:

- random
- round robin
- weighted, least request
- consistent hashing

TRAFFIC/REQUEST ROUTING

Because Envoy can understand application protocols like HTTP 1.1 and HTTP 2, Envoy can use sophisticated routing rules to direct traffic to specific backend clusters. Envoy can do basic reverse-proxy routing like mapping virtual hosts and context-path routing, it can also do header and and priority based routing, retry and timeouts for routing, as well as fault injection. As part of doing deployment techniques like canary releases, Envoy can be used to finely control which requests go to which versions of a deployment. Envoy supports traffic splitting and traffic shifting for those usecases. One particularly interesting feature of Envoy's routing capabilities is the ability to do traffic shadowing which we'll cover specifically in the next item. We'll take a closer look at these capabilities in Chapter 5.

TRAFFIC SHADOWING

Envoy can do sophisticated percentage/weighted traffic routing for splitting and shifting usecases (usually doing multiple deployments/canaries), but those deal with live user traffic. Envoy can also make copies of the traffic and **shadow** that traffic in a **fire and forget** mode to an Envoy `cluster`. You can think of this shadowing capability as doing something like traffic splitting, but that the requests that the `upstream cluster` sees is actually just a copy of the live traffic; it's not really acting on live production traffic and is not in the request path. This is a very powerful capability for testing production changes with production traffic without impacting customers. We'll see more of this in Chapter 5.

NETWORK RESILIENCE

Envoy can be used to offload certain classes of resilience problems, but note that it's the application's responsibility to fine-tune and configure these parameters. Envoy can automatically do request timeouts as well as request-level retries (with per-retry timeouts). This type of retry behavior is very useful when a request experiences intermittent network instability. On the other hand, retry amplification can lead to cascading failures; Envoy allows you to limit retry behavior. Also note, application-level retries may still be needed and cannot be completely offloaded to Envoy. Additionally, when Envoy calls `upstream` clusters, Envoy can be configured with bulkheading characteristics like limiting the number of connections or outstanding requests in flight and to fast-fail any that exceed those thresholds (with some jitter on those thresholds). Lastly, Envoy can perform "outlier detection" which behaves like a circuit breaker and eject endpoints from the load-balancing pool when they misbehave.

HTTP/2 AND GRPC

HTTP/2 is a major improvement to the HTTP protocol which allows multiplexing requests over a single connection, server-push interactions, streaming interactions, and request back pressure. Envoy was built from the beginning to be a HTTP/1.1 and HTTP/2 proxy with proxying capabilities for each protocol on both `downstream` and `upstream`. This means, for example, Envoy can accept HTTP/1.1 connections and proxy to HTTP/2 - or vice versa - or proxy incoming HTTP/2 to `upstream` HTTP/2 clusters. gRPC is an RPC protocol using Google Protobufs that lives on top of HTTP/2 and is also natively supported by Envoy. These are powerful features (and difficult to get correct in an implementation) and really differentiates Envoy from other service proxies.

OBSERVABILITY WITH METRICS COLLECTION

As we saw in the Envoy announcement from Lyft back in September 2016, one of the goals of Envoy is to help make the network understandable. Envoy collects a large set of metrics to help achieve this goal. Envoy tracks a lot of dimensions around the `downstream` systems that call it, the server itself, and the `upstream` clusters to which Envoy sends requests. Envoy's stats are tracked as counters, gauges, or histograms. Here's an example of the type of statistics tracked for an `upstream` cluster:

Statistic	Description
downstream_cx_total	Total connections
downstream_cx_http1_active	Total active HTTP/1.1 connections
downstream_rq_http2_total	Total HTTP/2 requests
cluster.<name>.upstream_cx_overflow	Total times that the cluster's connection circuit breaker overflowed
cluster.<name>.upstream_rq_retry	Total request retries
cluster.<name>.ejections_detected_consecutive_5xx	Number of detected consecutive 5xx ejections (even if unenforced)

Envoy can emit stats using configurable adapters and formats. Out of the box Envoy supports:

- statsd
- datadog / dogstatsd
- hystrix formatting
- generic metrics service

OBSERVABILITY WITH DISTRIBUTED TRACING

Envoy can report trace spans to Open Tracing (http://opentracing.io) engines for the purpose of visualizing traffic flow, hops, and latency in a call graph. This means, you don't have to install special OpenTracing libraries. On the other hand, the application is responsible for propagating the necessary Zipkin headers which can be done with thin wrapper libraries. In reality, you'll want to complement Envoy's tracing capability with some of the feature-rich libraries of a tracing engine which we'll discuss further in Chapter XX.

Envoy generates a `x-request-id` header to correlate calls across services and can also generate the initial `x-b3*` headers when tracing is triggered. The headers that the application is responsible for propagating are:

- `x-b3-traceid`
- `x-b3-spanid`
- `x-b3-parentspanid`
- `x-b3-sampled`
- `x-b3-flags`

AUTOMATIC TLS TERMINATION AND ORIGINATION

Envoy can terminate Transport Level Security (TLS/SSL) traffic destined to a specific service both at the edge of a cluster as well as deep within a mesh of service proxies. A more interesting capability is that Envoy can be used to **originiate** TLS traffic to an `upstream` cluster on behalf of an application. For enterprises developers and operators this means we don't have to muck with language-specific settings and keystores/truststores. By having Envoy in our request path, we can automatically get TLS and even mutual TLS.

RATE LIMITING

An important aspect of resiliency is the ability to restrict or limit access to resources that are protected. Resources like databases or caches or shared services may be protected for various reasons such has:

- Expensive to call (per-invocation cost)
- Slow or unpredictable latency
- Need fairness algorithms to protect starvation

Especially as services are configured for retries, we don't want to magnify the effect of certain failures in the system. To help throttle requests in these scenarios, we can use a global rate

limiting service. Envoy can integrate with a rate limiting service at both the network (per connection) and HTTP (per request) level. We'll see how Istio helps connect to a rate-limiting service.

EXTENDING ENVOY

At its core, Envoy is really a byte-processing engine on which protocol (layer 7) codecs (called **filters**) can be built. Envoy makes building additional filters a first-class use case and represents an exciting way to extend Envoy for specific use cases. Envoy filters are written in C++ and compiled into the Envoy binary. Additionally, Envoy supports Lua (https://www.lua.org) scripting for a less invasive approach to extending Envoy functionality.

3.1.2 Envoy compared to other proxies

Envoy's sweet spot is playing the role of application or service proxy where Envoy facilitates applications talking to each other through the proxy and solves the problems of reliability and observability. Other proxies have evolved from their origins as load balancers and web servers into more capable and performant proxies. Some of these communities don't move all that fast, or are closed-source and have taken a while to evolve to the point they can be used in application-to-application scenarios. The main areas Envoy shines with respect to other proxies are:

- open community
- modular codebase built for maintenance and extension
- HTTP/2 support (upstream and downstream)
- deep protocol metric collection
- C++ / non garbage collected
- configuration reloads / dynamic configuration

For a more specific and detailed comparison please take a look at the following:

- Envoy's documentation and comparison http://bit.ly/2U2g7zb
- Turbine Labs switch from NGINX to Envoy http://bit.ly/2nn4tPr
- Cindy Sridharan's initial take on Envoy http://bit.ly/2OqbMkR
- Why Datawire chose Envoy over HAProxy and NGINX http://bit.ly/2OVbsvz

3.2 Configuring Envoy

Envoy is driven by a configuration file in either JSON or YAML format. The configuration file specifies `listeners`, `routes`, and `clusters` as well as server-specific settings like whether to enable the Admin API, where access logs should go, tracing engine configuration and so on. For any folks already familiar with Envoy or Envoy configuration, you may know there are different versions of the Envoy config. The initial version, v1, was the original way of configuring Envoy when it launched. That version has since been deprecated in favor of v2 of the Envoy configuration. The reference documentation for Envoy (https://www.envoyproxy.io/docs) also has the explicit distinction of v1 and v2 docs. We will be looking only at v2 configuration in this book as that's the go forward version and also is what Istio uses.

Envoy's v2 configuration API is built on gRPC. Envoy and implementors of the v2 API can take advantage of streaming capabilities when calling the API and lower the amount of time it takes for the Envoys to converge on the correct configuration. In practice this eliminates the need to poll the API and allows the server to push updates to the Envoys instead of the proxies polling at periodic intervals.

3.2.1 Static configuration

We can specify listeners, route rules, and clusters using Envoy's configuration file. We can see a very simple Envoy configuration here:

```
static_resources:
  listeners:   ❶
  - name: httpbin-demo
    address:
      socket_address: { address: 0.0.0.0, port_value: 15001 }
    filter_chains:
    - filters:
      - name: envoy.http_connection_manager   ❷
        config:
          stat_prefix: egress_http
          route_config:   ❸
            name: httpbin_local_route
            virtual_hosts:
            - name: httpbin_local_service
              domains: ["*"]   ❹
              routes:
              - match: { prefix: "/" }
                route:
                  auto_host_rewrite: true
                  cluster: httpbin_service   ❺
          http_filters:
          - name: envoy.router
  clusters:
    - name: httpbin_service   ❻
      connect_timeout: 5s
      type: LOGICAL_DNS
      # Comment out the following line to test on v6 networks
      dns_lookup_family: V4_ONLY
      lb_policy: ROUND_ROBIN
      hosts: [{ socket_address: { address: httpbin, port_value: 8000 }}]
```

❶ listener definitions

❷ HTTP filter

❸ route rules

❹ wildcard virtual hosts

❺ route to cluster

❻ upstream clusters

In this simple Envoy configuration file, we declare a listener that opens a socket on port `15001` and attaches a chain of filters to it. The filters configure the `http_connection_manager` in Envoy with routing directives. The simple routing directive we see in this example is to match on the wildcard `*` for all virtual hosts, and route all traffic to the `httpbin_service` cluster. The last section of the configuration defines the connection properties to the `httpbin_service` cluster. In this example, we specify `LOGICAL_DNS` for endpoint service discovery and `ROUND_ROBIN` for load balancing when talking to the upstream `httpbin` service. See Envoy's documentation (www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/service_discovery#logical-dns) for more.

This is a simple configuration file that creates a `listener` to which incoming traffic can connect and routes **all** traffic (see note #4) to the `httpbin` cluster. It also specifies what load-balancing settings to use and what kind of connect timeout to use. If we call this proxy, we would expect our request to get routed to an `httpbin` service.

You'll notice that a lot of the configuration is specified explicitly (ie, what listeners there are, what the routing rules are, what clusters we can route to etc). This is an example of a fully static configuration file. In previous sections, we pointed out that Envoy has the ability to dynamically configure its various settings. For the hands-on section of Envoy, we'll use the static configurations, but we'll first cover the dynamic services and how Envoy uses its "xDS" APIs for dynamic configuration.

3.2.2 Dynamic configuration

Envoy can leverage a set of APIs to do in-line configuration updates without any downtime or restarts. Envoy just needs a simple bootstrap configuration file that points the configuration to the correct discovery service APIs and the rest is configured dynamically. The APIs that Envoy leverages for dynamic configuration are the following:

- Listener Discovery Service (LDS) - an API that allows Envoy to query what `listeners` should be exposed on this proxy
- Route Discovery Service (RDS) - a part of the configuration for `listeners` that specifies which `routes` to use; this is a subset of LDS for when static and dynamic configuration should be used
- Cluster Discovery Service (CDS) - an API that allows Envoy to discover what clusters

and respective configuration for each cluster this proxy should have

- Endpoint Discovery Service (EDS) - a part of the configuration for `clusters` that specifies which `endpoints` to use for a specific cluster; this is a subset of CDS
- Secret Discovery Service (SDS) - an API used to distribute certificates
- Aggregate Discovery Service (ADS) - a serialized stream of all the changes to the rest of the APIs; you can use this single API to get all of the changes in order

Collectively, these APIs are referred to as the **xDS** services. A configuration can use one or some combination of them; you don't have to use all of them. One thing to note is that Envoy's xDS APIs are built on a premise of "eventual consistency" and that correct configurations will converge eventually. For example, Envoy could end up getting an update to the RDS with a new route that routes traffic to a cluster **foo** that may not have been updated in CDS yet. This means, the route could introduce routing errors until the CDS is updated. Envoy introduced the Aggregated Discovery Service (ADS) to account for this ordering race conditions. Istio implements the Aggregated Discovery Service and uses ADS for proxy configuration changes.

For example, to dynamically discover the `listeners` for an Envoy proxy, we can use a configuration like the following:

```
dynamic_resources:
  lds_config:   ❶
    api_config_source:
      api_type: GRPC
      grpc_services:
        - envoy_grpc:   ❷
            cluster_name: xds_cluster

clusters:
- name: xds_cluster   ❸
  connect_timeout: 0.25s
  type: STATIC
  lb_policy: ROUND_ROBIN
  http2_protocol_options: {}
  hosts: [{ socket_address: { address: 127.0.0.3, port_value: 5678 }}]
```

❶ Configuration for Listeners (LDS)

❷ Go to this cluster for listener API

❸ gRPC cluster that implements LDS

With the above configuration, we don't need to explicitly configure each listener in the configuration file. We're telling Envoy to use the `LDS` API to discovery the correct listener configuration values at run time. We do, however, configure one cluster explicitly. This cluster is where the `LDS` API lives (named `xds_cluster` in this example).

For a more concrete example, Istio uses a `bootstrap` configuration for its service proxies similar to the following:

```
bootstrap:
  dynamicResources:
    ldsConfig:
      ads: {}   ❶
```

```
    cdsConfig:
      ads: {}  ❷
    adsConfig:
      apiType: GRPC
      grpcServices:
      - envoyGrpc:
          clusterName: xds-grpc   ❸
      refreshDelay: 1.000s
  staticResources:
    clusters:
    - name: xds-grpc   ❹
      type: STRICT_DNS
      connectTimeout: 10.000s
      hosts:
      - socketAddress:
          address: istio-pilot.istio-system
          portValue: 15010
      circuitBreakers:   ❺
        thresholds:
        - maxConnections: 100000
          maxPendingRequests: 100000
          maxRequests: 100000
        - priority: HIGH
          maxConnections: 100000
          maxPendingRequests: 100000
          maxRequests: 100000
      http2ProtocolOptions: {}
```

❶ ADS for listeners

❷ ADS for clusters

❸ Use cluster named xds-grpc

❹ Definition of xds-grpc cluster

❺ Reliability, circuit-breaking settings

Let's tinker with a simple static Envoy configuration file to see Envoy in action.

3.3 Envoy in action

Envoy is written in C++ and compiled to a native/specific platform. The best way to get started with Envoy is to just use Docker and run a docker container with it. We've been using Docker for Desktop for this book, but access to any Docker daemon can be used for this section. For example, on a Linux machine you can directly install docker.

At this point, you should have access to a docker daemon. You can test access like this:

```
$  docker ps
```

You should see a list of containers in your console. You may see a long list of containers if you've followed the hands-on parts of the book up to this point as we deployed a handful of applications and the Istio control plane. You should start by pulling three docker images that we'll use to explore Envoy's functionality:

```
$  docker pull istioinaction/envoy:v1.15.0
$  docker pull tutum/curl
```

```
$  docker pull citizenstig/httpbin
```

To get started, we'll create a simple `httpbin` service. If you're not familiar with httpbin, you can go to httpbin.org and explore the different endpoints available. It basically implements a simple service that can return us headers that were used to call it, delay an HTTP request, or even throw an error all depending which endpoint you call. For example, navigate to httpbin.org/headers. Once we start the `httpbin` service, we'll start up Envoy and configure it to proxy all traffic to the `httpbin` service. Then we'll start up a client app and call the proxy. The simplified architecture of this example looks like this:

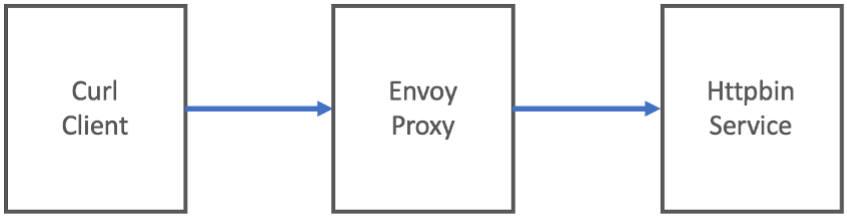


Figure 3.4 The sample applications we'll use to exercise some of Envoy's functionality

Run the following command to set up our `httpbin` service running in Docker:

```
$  docker run -d --name httpbin citizenstig/httpbin
787b7ec9365ff01841f2525cdd4e74e154e9d345f633a4004027f7ff1926e317
```

Let's test that our new `httpbin` service was correctly deployed by querying the `/headers` endpoint:

```
$  docker run -it --rm --link httpbin tutum/curl \
curl -X GET http://httpbin:8000/headers

{
  "headers": {
    "Accept": "*/*",
    "Host": "httpbin:8000",
    "User-Agent": "curl/7.35.0"
  }
}
```

You should see the output similar to above; we see the response return with the headers we used to call the `/headers` endpoint.

Now let's run our Envoy proxy and pass `--help` to the command and explore some of its flags and command line parameters:

```
$  docker run -it --rm istioinaction/envoy:v1.15.0 envoy --help
```

Some of the interesting flags are the `-c` flag for passing in a configuration file, the `--service-zone` flag for specifying into what availability zone the proxy is deployed, and `--service-node` which gives the proxy a unique name. You may also be interested in the `--log-level` flag which controls how verbose the logging is from the proxy.

Let's try and run Envoy:

```
$  docker run -it --rm istioinaction/envoy:v1.15.0 envoy

[2018-08-09 22:51:47.214][6][critical][main] source/server/server.cc:78]
error initializing configuration '': unable to read file:
[2018-08-09 22:51:47.214][6][info][main] source/server/server.cc:437] exiting
```

What happened here? We tried to run the proxy, but we did not pass in a valid configuration file. Let's fix that and pass in a simple configuration file. The config file we'll pass in is based on the sample configuration we saw earlier and has a structure like this:

```
static_resources:
  listeners:   ❶
  - name: httpbin-demo
    address:
      socket_address: { address: 0.0.0.0, port_value: 15001 }
    filter_chains:
    - filters:
      - name: envoy.http_connection_manager
        config:
          stat_prefix: egress_http
          route_config:
            name: httpbin_local_route
            virtual_hosts:
            - name: httpbin_local_service
              domains: ["*"]
              routes:
              - match: { prefix: "/" }
                route:   ❷
                  auto_host_rewrite: true
                  cluster: httpbin_service
          http_filters:
          - name: envoy.router
  clusters:
    - name: httpbin_service   ❸
      connect_timeout: 5s
      type: LOGICAL_DNS
      # Comment out the following line to test on v6 networks
      dns_lookup_family: V4_ONLY
      lb_policy: ROUND_ROBIN
      hosts: [{ socket_address: { address: httpbin, port_value: 8000 }}]
```

❶ A listener on port 15001

❷ A simple route rule

❸ A cluster for httpbin

Basically, we're exposing a single listener on port `15001` and we'll route all traffic to our `httpbin` cluster. Let's start up Envoy with this configuration file (NOTE: this configuration file is automatically included in the `istioinaction/envoy` Docker image. There is no need to mount it into the container yourself.)

```
$  docker run -d --name proxy --link httpbin \
istioinaction/envoy:v1.15.0 envoy -c /etc/envoy/simple.yaml
5d32538c078a6e14ba0d4072d6ff10592a8a439714e7c9ac9c69e1ff71aa54f2

$ docker logs proxy
[2018-08-09 22:57:50.769][5][info][config]
 all dependencies initialized. starting workers
```

```
[2018-08-09 22:57:50.769][5][info][main]
 starting main dispatch loop
```

Now we should see the proxy has started successfully and is listening on port `15001`. Let's use a simple command line client, `curl`, to call the proxy:

```
$  docker run -it --rm --link proxy tutum/curl \
   curl  -X GET http://proxy:15001/headers

{
  "headers": {
    "Accept": "*/*",
    "Content-Length": "0",
    "Host": "httpbin",
    "User-Agent": "curl/7.35.0",
    "X-Envoy-Expected-Rq-Timeout-Ms": "15000",
    "X-Request-Id": "45f74d49-7933-4077-b315-c15183d1da90"
  }
}
```

We see the traffic was correctly sent to the `httpbin` service even though we called the proxy. We also see that the headers used to call the `httpbin` service are slightly different insofar we have some new headers:

- `X-Envoy-Expected-Rq-Timeout-Ms`
- `X-Request-Id`

It may seem insignificant, but Envoy is already doing a lot for us here. It generated a new `x-request-id` which can be used to correlate requests across a cluster and potentially multiple hops across services to fulfill the request. The second header `x-envoy-expected-rq-timeout-ms` is a hint to upstream services that the request is expected to timeout after `15000ms`. Upstream systems, and any other hops it takes, can use this hint to implement a **deadline**. A deadline allows us to communicate timeout intentions to upstream systems, and allow them to cease processing if the deadline has passed. This frees up resources after a timeout has been executed.

Let's alter this configuration a little bit. Let's try to set the expected request timeout to 1s. In our configuration file, we'd update the route rule:

```
- match: { prefix: "/" }
    route:
        auto_host_rewrite: true
        cluster: httpbin_service
        timeout: 1s
```

For this example, we've already updated the configuration file, and it's available in the docker image. It's named `simple_change_timeout.yaml` and we can pass it as an argument to Envoy. Let's stop our existing proxy and restart it with this new configuration file:

```
$  docker rm -f proxy
proxy
```

```
$   docker run -d --name proxy --link httpbin \
istioinaction/envoy:v1.15.0 envoy -c /etc/envoy/simple_change_timeout.yaml
26fb84558165ae9f9d9afb67e9dd7f553c4d412989904542795a82cc721f1ce5
```

Now, let's call the proxy again:

```
$  docker run -it --rm --link proxy tutum/curl \
curl  -X GET http://proxy:15001/headers

{
  "headers": {
    "Accept": "*/*",
    "Content-Length": "0",
    "Host": "httpbin",
    "User-Agent": "curl/7.35.0",
    "X-Envoy-Expected-Rq-Timeout-Ms": "1000",
    "X-Request-Id": "c7e9212a-81e0-4ac2-9788-2639b9898772"
  }
}
```

Now we see the expected request timeout value changed to `1000ms`. Let's do something a little bit more exciting than changing the deadline hint headers.

3.3.1 Envoy's Admin API

To explore more of Envoy's functionality, let's first get familiar with Envoy's Admin API. The Admin API gives us insight into how the proxy is behaving, access to its metrics, and access to its configuration. Let's start by running `curl` against `proxy:15000/stats`

```
$  docker run -it --rm --link proxy tutum/curl \
curl -X GET http://proxy:15000/stats
```

The response should be a long list of statistics and metrics for the listeners, clusters, and server itself. Let's trim the output using `grep` and only show those statistics with the word `retry` in it:

```
$  docker run -it --rm --link proxy tutum/curl \
curl -X GET http://proxy:15000/stats | grep retry

cluster.httpbin_service.retry_or_shadow_abandoned: 0
cluster.httpbin_service.upstream_rq_retry: 0
cluster.httpbin_service.upstream_rq_retry_overflow: 0
cluster.httpbin_service.upstream_rq_retry_success: 0
```

If you call the Admin API directly, without the `/stats` context path, you should see a list of other endpoints you can call. Some endpoints to explore include:

- `/certs` - the certificates on the machine
- `/clusters` - the clusters Envoy is configured with
- `/config_dump` - dump the actual Envoy config
- `/listeners` - the listeners Envoy is configured with
- `/logging` - can view and change logging settings
- `/stats` - Envoy statistics
- `/stats/prometheus` - Envoy statistics as prometheus records

3.3.2 Envoy request retries

Let's cause some failures in our request to `httpbin` and watch how Envoy can automatically retry a request for us. First, we'll update the configuration file to use a `retry_policy`:

```
- match: { prefix: "/" }
  route:
    auto_host_rewrite: true
    cluster: httpbin_service
    retry_policy:   ❶
        retry_on: 5xx   ❷
        num_retries: 3   ❸
```

❶ retry in effect

❷ when 5xx

❸ # of times

Just like in the previous example, we won't have to actually update the configuration file; an updated version of the file is already available on the docker image named `simple_retry.yaml`. Let's pass in the configuration file this time when we start Envoy:

```
$  docker rm -f proxy
proxy

$   docker run -d --name proxy --link httpbin \
istioinaction/envoy:v1.15.0 envoy -c /etc/envoy/simple_retry.yaml
4f99c5e3f7b1eb0ab3e6a97c16d76827c15c2020c143205c1dc2afb7b22553b4
```

Now call our proxy with the `/status/500` context path. Calling `httpbin` (which the proxy does) with that context path will force an error. Let's try it:

```
$  docker run -it --rm --link proxy tutum/curl \
curl -X GET http://proxy:15001/status/500
```

When the call completes we shouldn't see any response. What happened here? Let's ask Envoy's Admin API what happened:

```
$  docker run -it --rm --link proxy tutum/curl \
curl -X GET http://proxy:15000/stats | grep retry

cluster.httpbin_service.retry.upstream_rq_500: 3
cluster.httpbin_service.retry.upstream_rq_5xx: 3
cluster.httpbin_service.retry_or_shadow_abandoned: 0
cluster.httpbin_service.upstream_rq_retry: 3
cluster.httpbin_service.upstream_rq_retry_overflow: 0
cluster.httpbin_service.upstream_rq_retry_success: 0
```

We see that Envoy encountered a `500` HTTP response when talking to the upstream cluster `httpbin`. This is as we expected. We also see that Envoy automatically retried the request for us as indicated by this stat `cluster.httpbin_service.upstream_rq_retry: 3`.

We just demonstrated some very basic capabilities of Envoy to automatically give us some reliability in our application networking. We used some static configuration files to reason about

and demonstrate these basic capabilities, but as we saw in the previous section Istio leverages the dynamic configuration capabilities. Doing so allows Istio to manage a large fleet of Envoy proxies each with their own and potentially complex configurations. Please refer to the Envoy documentation (https://www.envoyproxy.io) or a series of blogs going into more detail on Envoy's capabilities (http://bit.ly/2M6Yld3)

3.4 How Envoy fits with Istio

Envoy provides the bulk of the heavy lifting for most of the Istio features we covered in Chapter 1 and 2. As a proxy, Envoy is a great fit for the service-mesh use case, however, to get the most value out of Envoy, it needs supporting infrastructure or components. As we've mentioned a few times now, Envoy forms the data plane of a service mesh. The supporting components, which Istio provides, creates the control plane.

A couple of examples should make this clear. We saw with Envoy that we can configure a fleet of service proxies using static configuration files or using a set of **discovery services** for discovering listeners, endpoints, and clusters at runtime. Istio implements these XDS APIs in Istio Pilot. Another related example is Envoy's service discovery relies on a service registry of some sort to discover endpoints. Istio Pilot implements this API but also abstracts Envoy away from any particular service-registry implementation. When Istio is deployed on Kubernetes, Kubernete's service registry is what Istio uses for service discovery. Other registries can also be used like HashiCorp's Consul. The Envoy data plane is completely shielded from those implementation details.

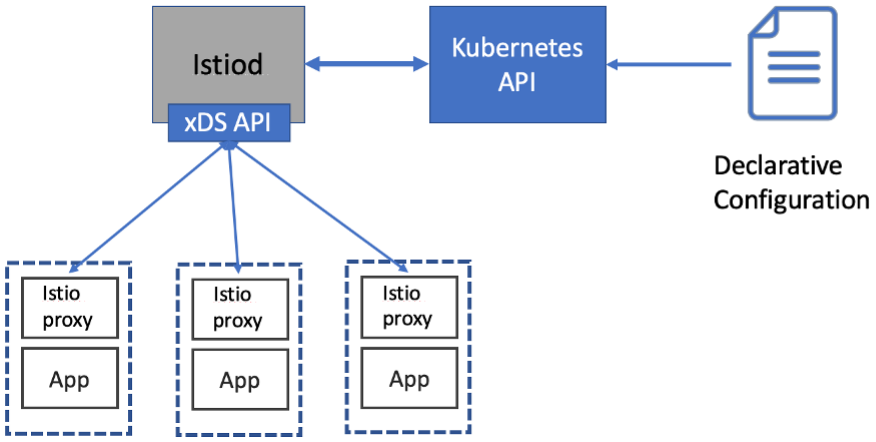


Figure 3.5 Istio abstracts away service registry and provides an implementation of Envoy's xDS API

Another example: Envoy can emit a lot of metrics and telemetry. This telemetry needs to go somewhere and Envoy must be configured to send it there. Istio provides telemetry sinks as part of its control plane to which Envoy can send these data. We also saw how Envoy can send distributed tracing spans to an OpenTracing engine. Istio can handle installing a compliant OpenTracing engine and configuring Envoy to send its spans to that location. For example, Istio

comes with the Jaeger tracing engine https://www.jaegertracing.io, although Zipkin can be used as well https://zipkin.io.

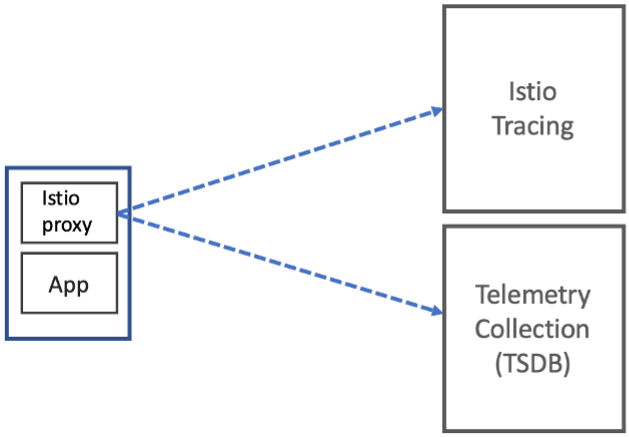


Figure 3.6 Istio helps configure and integrate with metrics-collection and distributed-tracing infrastructure

Lastly, Envoy can terminate and originate TLS traffic to services in our mesh. To do this, you need supporting infrastructure to create, sign and rotate certificates. Istio provides this with the Istio Citadel component.

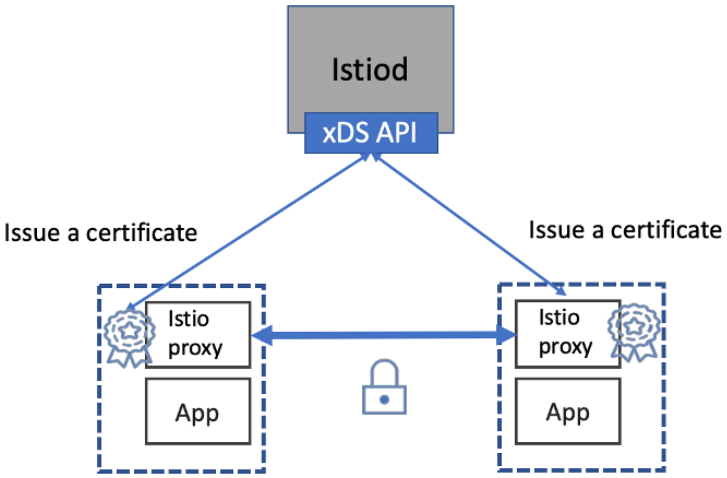


Figure 3.7 Istio Citadel delivers application-specific certificates which can be used to establish mutual TLS to secure the traffic between services

Together Istio's control plane and Envoy's data plane make for a compelling service-mesh implementation. Both have thriving and vibrant communities and are geared toward next-generation services architectures. The rest of the book assumes Envoy as a data plane, so all of your learning from this chapter is transferable to the rest of the chapters in this book. From here on, we'll refer to Envoy as the Istio "service proxy" and any of its capabilities will be seen through Istio'sAPIs but understand that a lot of those are actually coming from and implemented by Envoy.

In the next chapter, we'll look at how we can begin to get traffic into our service-mesh cluster by

going through an edge gateway/proxy that controls traffic. When client applications outside of our cluster wish to communicate with services running inside our cluster, we need to be very clear and explicit about what traffic is allowed in and what is not allowed. We'll look at Istio's Gateway and how it provides the functionality we need to establish a controlled ingress point, and all of the knowledge you learned in this chapter will apply: Istio's default gateway is built on Envoy proxy.

3.5 Summary

- Envoy is a proxy that applications can leverage for application-level behavior
- Envoy is Istio's default data plane
- Envoy can help solve cloud reliability challenges (network failures, topology changes, elasticity) consistently and correctly
- Envoy leverages a dynamic API for runtime control (which Istio uses)
- Envoy exposes a lot of powerful metrics and information about application usage and proxy internals

4 Istio Gateway: getting traffic into your cluster

This chapter covers:

- Defining entry points into a cluster
- Routing ingress traffic to deployments in your cluster
- Securing ingress traffic
- Routing non HTTP/S traffic

As we'll see throughout the rest of this book, Istio will allow us to solve some difficult challenges in service-to-service communication. For most of the book, we'll assume a single cluster with a single Istio control-plane deployment, but in reality Istio's capabilities are not limited to a single or homogeneous cluster. But even before we look at multi-cluster or hybrid deployments, we should understand how to connect different networks together. This chapter will consider two different networks: the cluster in which the service mesh is deployed and where user services are deployed, and anything outside of the cluster.

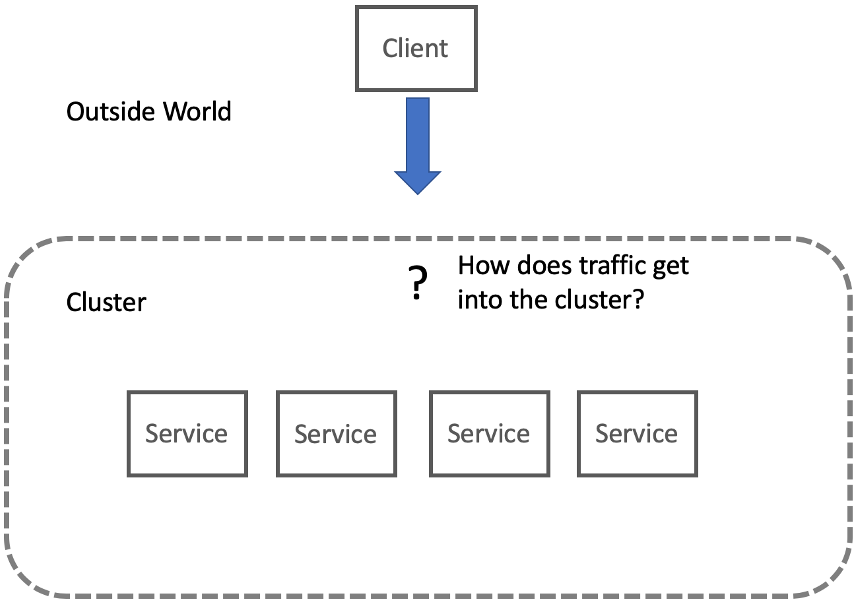


Figure 4.1 We want to connect networks: clients running outside of our cluster to services running inside our cluster

We will most likely run interesting services and applications **inside** our cluster. We will most likely have intra-service communication within the cluster and that's where Istio shines. But what about those clients that are deployed or exist outside of the cluster? In this chapter, we'll take a look at connecting those clients that live **outside** the cluster to services running **inside** the cluster.

4.1 Traffic ingress concepts

The networking community has a term for connecting networks via well established entry points called **ingress points**. Ingress refers to traffic that originates outside the local network and is intended for an endpoint within the local network. This traffic is first routed to exposed ingress points for the purpose of enforcing rules and policies about what traffic is allowed into the local network. If traffic does not go through these ingress points, it cannot connect with any of the services inside the cluster. If the ingress point allows the traffic, it proxies it to the correct endpoint in the local network. If the traffic is not allowed, the ingress point refuses to proxy the traffic.

4.1.1 Virtual IPs: simplifying service access

At this point, it's useful to dig in a little bit more to how traffic is routed to a network's ingress points, at least how it relates to the type of clusters at which we'll be looking in this book. Let's say we have an service that we wish to expose at `api.istioinaction.io/v1/products` for external systems to get a list of products in our catalog. When our client tries to query that endpoint, the client's networking stack first tries to resolve the `api.istioinaction.io` domain name to an IP address. This is done with DNS servers. Their networking stack would query the DNS servers for what the IP address is which means we need to first map our service's IP to this name when we register it in DNS. For a public address, we could use a service like Amazon Route 53 or Google Cloud DNS and map a domain name to an IP address. In our own datacenters, we'd use our internal DNS servers to do the same thing. But to what IP address should we map the name?

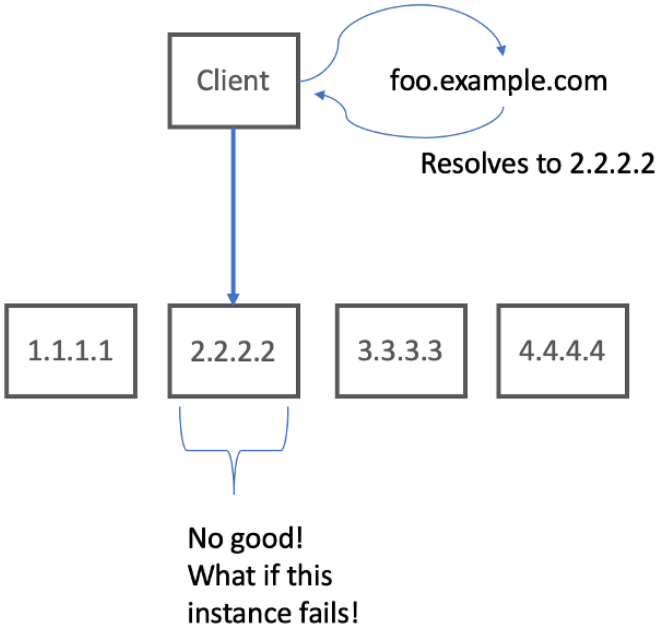


Figure 4.2 We don't want to map domain names to specific instances and IPs of a service

We are probably not going to map the name directly to a single instance or single endpoint of our service (single IP) as that can be very fragile. What would happen if that one specific service instance goes down? Clients would see a lot of errors until we changed the DNS mapping to a new IP address with a working endpoint. But doing this any time a service goes down is slow, error prone, and low availability.

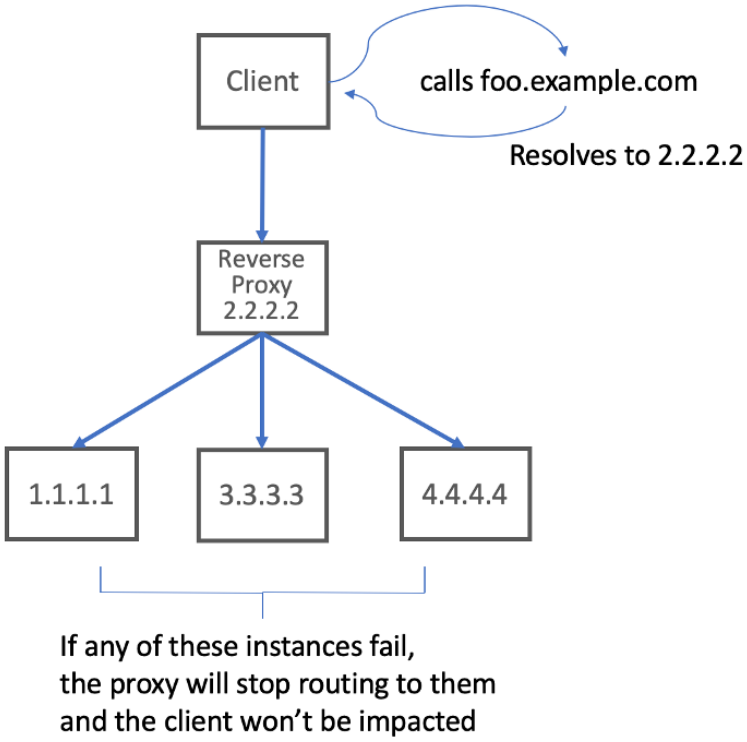


Figure 4.3 Let's map a virtual IP to a reverse proxy that handles load balancing across service instances

What we want to do is map the domain name to an IP address that **represents** our service. This single "virtual IP" would **represent** our service to be able to forward traffic to our actual service instances. We will map the domain name to a **virtual IP** that's bound to a type of ingress point known as a **reverse proxy**. The reverse proxy is an intermediary component that's responsible for distributing requests to backend services and does not correspond to any specific service. The reverse proxy can also provide capabilities like load balancing so requests don't overwhelm any one specific backend.

4.1.2 Virtual Hosting: multiple services from a single access point

Foo: In the previous section we saw how a single virtual IP can be used to address a service that may be comprised of many actual service instances with their own IP, however, the client only uses the virtual IP. We can also represent multiple different services using a single virtual IP. For example, we could have both `prod.istioinaction.io` and `api.istioinaction.io` resolve to the same virtual IP address. This means requests for both URIs would end up going to the same virtual IP and thus the same ingress reverse proxy. If the reverse proxy was smart enough, it could use the `Host` HTTP header to further delineate which requests should go to which group of services.

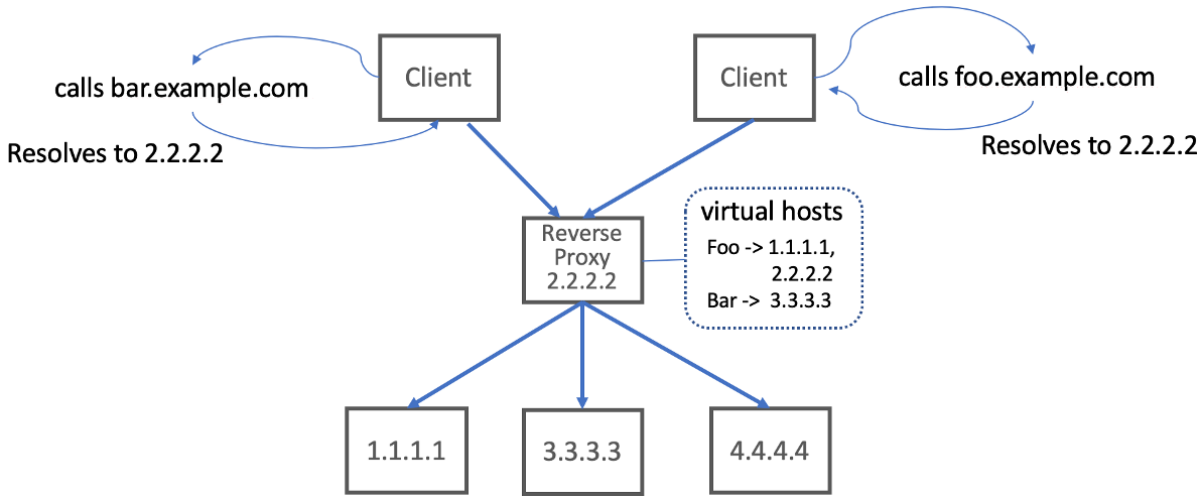


Figure 4.4 Virtual hosting lets us map multiple services to a single Virtual IP

Hosting multiple different services at a single entry point is known as **virtual hosting**. We need some way to decide to which virtual-host group a particular request should be routed. With HTTP/1.1, we can use the `Host` header, with HTTP/2 we can use the `:authority` header, and with TCP connections we can rely on Server Name Indication (SNI) with TLS. We'll take a closer look at SNI later in this chapter. The important thing to note is that the edge ingress functionality we'll see in Istio uses both virtual IP routing as well as virtual hosting to route service traffic into the cluster.

4.2 Istio Gateway

Istio has a concept of an ingress `Gateway` that plays the role of the network-ingress point and is responsible for guarding and controlling access to the cluster from traffic that originates outside of the cluster. Additionally, Istio's Gateway also plays the role of load balancing and virtual-host routing.

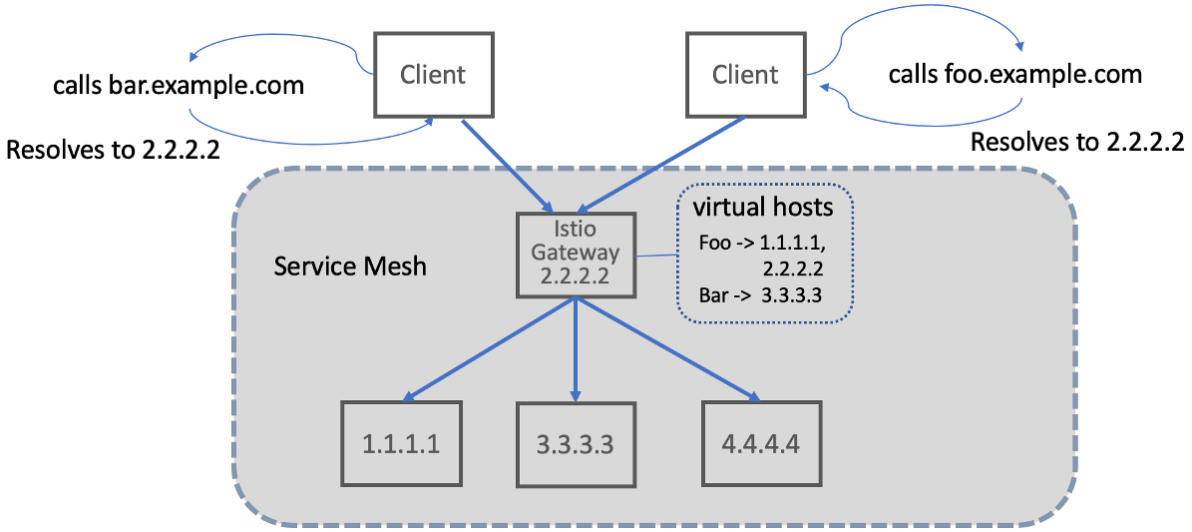


Figure 4.5 Istio Gateway plays the role of network ingress and uses Envoy Proxy to do the routing and load balancing

In Figure XX we see that by default Istio uses an Envoy proxy as the ingress proxy. We saw in Chapter 3 that Envoy is a capable service-to-service proxy, but it can also be used to load balance and route proxy traffic from outside the service mesh to services running inside of it. All of the features of Envoy that we saw in the previous chapter are also available in the ingress gateway.

Let's take a closer look at how Istio uses Envoy to implement an ingress gateway. When we installed Istio in Chapter 2, we saw the listing of components that make up the control plane and any additional components that support the control plane.

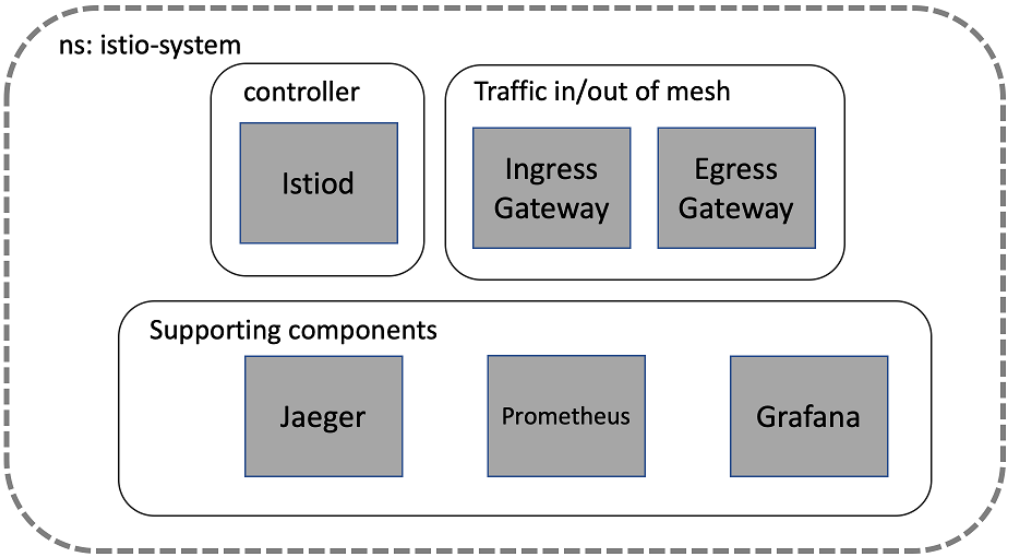


Figure 4.6 Review of the components installed in chapter 2; some comprise the Istio control plane while others support it

If you'd like to verify that the Istio service proxy is indeed running in the Istio ingress gateway, you can run something like this from the root directory of the source code for this book:

```
$  INGRESS_POD=$(make ingress-pod)
$  kubectl -n istio-system exec $INGRESS_POD -- ps aux
```

We should see a process listing as the output showing the Istio service proxy command line with both the `pilot-agent` and the `envoy` processes.

To configure Istio's `Gateway` to allow traffic into the cluster and through the service mesh, we'll start by exploring two concepts: `Gateway` and `VirtualService`. Both are fundamental in general to getting traffic to flow in Istio, but we'll look at them only within the context of allowing traffic into the cluster. We will cover `VirtualService` more fully in the next chapter about traffic management and routing.

4.2.1 Specifying Gateway resources

To configure a `Gateway` in Istio, we use the `Gateway` resource and specify which ports we wish to open on the Gateway, and what virtual hosts to allow for those ports. The example `Gateway` we'll explore is quite simple and exposes an HTTP port on port `80` that will accept traffic destined for virtual host `apiserver.istioinaction.io`:

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: coolstore-gateway   ❶
spec:
  selector:
    istio: ingressgateway   ❷
  servers:
  - port:
      number: 80    ❸
      name: http
      protocol: HTTP
    hosts:
    - "apiserver.istioinaction.io"   ❹
```

❶ name of gateway

❷ which gateway implementation

❸ ports to expose

❹ host(s) for this port

This `Gateway` definition is intended for the `istio-ingressgateway` that was created when we set up Istio initially, but we could have used our own definition of Gateway. We can define to which gateway the configuration applies by using the labels in the `selector` section of the `Gateway` configuration. In this case, we're selecting the gateway implementation with the label `istio: ingressgateway` which will match the default `istio-ingressgateway`. The Gateway for `istio-ingressgateway` is really just an instance of Istio's service proxy (Envoy), and its Kubernetes deployment configuration looks something like this:

```
containers:
- name: ingressgateway
  image: "gcr.io/istio-release/proxyv2:1.7.0"   ❶
  imagePullPolicy: IfNotPresent
  ports:
    - containerPort: 80
    - containerPort: 443
  args:
  - proxy   ❷
  - router
  - --domain
  - $(POD_NAMESPACE).svc.cluster.local
  - --proxyLogLevel=warning
  - --proxyComponentLogLevel=misc:error
  - --log_output_level=default:info
  - --serviceCluster
  - istio-ingressgateway
  - --trust-domain=cluster.local
```

❶ Image to use

❷ Configuration for Envoy

Our `Gateway` resource configures Envoy to listen on port 80 and expect HTTP traffic. Let's create that resource and see what it does. From the root of the source code that accompanies this book, you should see a `chapters/chapter4/coolstore-gw.yaml` file. To create the configuration, navigate to the `chapters/chapter4` folder and run the following:

```
$  kubectl -n istioinaction apply -f coolstore-gw.yaml
```

Let's see whether our settings took affect.

```
$  istioctl proxy-config listener $INGRESS_POD  -n istio-system

ADDRESS PORT  MATCH DESTINATION
0.0.0.0 8080  ALL   Route: http.80
0.0.0.0 15021 ALL   Inline Route: /healthz/ready*
0.0.0.0 15090 ALL   Inline Route: /stats/prometheus*
```

So we've exposed our HTTP port (ie, port `80`) correctly! If we take a look at the routes for virtual services, we see that the Gateway doesn't have any at the moment (you may see another route for Prometheus, but we can ignore that for now):

```
$  istioctl proxy-config route $INGRESS_POD -o json \
--name http.80  -n istio-system

[
    {
        "name": "http.80",
        "virtualHosts": [
            {
                "name": "blackhole:80",
                "domains": [
                    "*"
                ],
                "routes": [
                    {
                        "name": "default",
                        "match": {
                            "prefix": "/"
```

```
                        },
                        "directResponse": {
                            "status": 404
                        }
                    }
                ]
            }
        ],
        "validateClusters": false
    }
]
```

We see our listener is bound to a "blackhole" default route that just routes everything to HTTP 404. In the next section, we'll take a look at setting up a virtual host for routing traffic from port 80 to a service within the service mesh.

Before we go to the next section there's an important last point to be made here. The pod running the gateway, whether that's the default `istio-ingressgateway` or your own custom gateway, will need to be able to listen on a port or IP that is exposed outside the cluster. For example, on our local Docker for Desktop that we're using for these examples, the ingress gateway is listening on port `80`.

If you're deploying on a cloud service like GKE, you'll want to make sure you use a `LoadBalancer` that gets an externally routable IP address. More information can be found at istio.io/docs/tasks/traffic-management/ingress/.

Additionally, the default `istio-ingressgateway` does not need privileged access to open any ports as it does not listen on any system ports (ie, `80` for HTTP). The `istio-ingressgateway` will by default listen on port `8080`, however whatever service or load balancer you use to expose the gateway will be the actual port. In our examples with Docker for Desktop, we expose the service on port `80`.

4.2.2 Gateway routing with Virtual Services

So far, all we've done is configured the Istio Gateway to expose a specific port, expect a specific protocol on that port, and define specific hosts to serve from the port/protocol pair. When traffic comes into the gateway, we need a way to get it to a specific service within the service mesh and to do that, we'll use the `VirtualService` resource. In Istio, a `VirtualService` resource defines how a client talks to a specific service through its fully qualified domain name, which versions of a service are available, and other routing properties (like retries and request timeouts). We'll cover `VirtualService` in more depth in the next chapter when we explore traffic routing more deeply, but in this chapter it's sufficient to know that `VirtualService` allows us to route traffic from the ingress gateway to a specific service.

An example of a `VirtualService` that routes traffic for the virtual host `apigateway.istioinaction.io` to services deployed in our service mesh looks like this:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: apigateway-vs-from-gw   ❶
spec:
  hosts:
  - "apiserver.istioinaction.io"   ❷
  gateways:
  - coolstore-gateway   ❸
  http:
  - route:
    - destination:   ❹
        host: apigateway
        port:
          number: 8080
```

❶ name of virtual service

❷ virtual host name(s)

❸ gateways to which this applies

❹ where to route traffic

With this `VirtualService` resource, we define what to do with traffic when it comes into the gateway. In this case, as you can see with `spec.gateways` field, these traffic rules apply only to traffic coming from the `coolstore-gateway` gateway definition which we created in the previous section. Additionally, we're specifying a virtual host of `apiserver.istioinaction.io` for which traffic must be destined for these rules to match. An example of matching this rule is a client querying `apiserver.istioinaction.io` which resolves to an IP that the Istio gateway is listening on. Additionally, a client could explicitly set the `Host` header in the HTTP request to be `apiserver.istioinaction.io` which we'll show through an example.

Again, verify you're in the `chapters/chapter4` directory:

```
$  kubectl apply -n istioinaction -f coolstore-vs.yaml
```

After a few moments (it may take a few for the configuration to sync; recall in previous sections that configuration in the Istio service mesh is **eventually consistent**), we can re-run our commands to list the `listeners` and `routes` :

```
$  istioctl proxy-config listener $INGRESS_POD  -n istio-system

ADDRESS     PORT     TYPE
0.0.0.0     80        HTTP
0.0.0.0     15090     HTTP

$  istioctl proxy-config route $INGRESS_POD -o json \
--name http.80  -n istio-system

[
  {
    "name": "http.80",
    "virtualHosts": [
      {
      "name": "apigateway-vs-from-gw:80",
      "domains": [
```

```
          "apiserver.istioinaction.io"   ❶
      ],
      "routes": [
        {
          "match": {
            "prefix": "/"
          },
          "route": {   ❷
            "cluster": "outbound|8080||apigateway.istioinaction.svc.cluster.local",
            "timeout": "0.000s"
          }
        }
      ]
      }
    ]
  }
]
```

❶ Domains to match

❷ Where to route

The output for the `route` should look similar to the previous listing, although it may contain other attributes and information. The critical part is we can see how defining a `VirtualService` created an Envoy route in our Istio Gateway that routes traffic matching domain `apiserver.istioinaction.io` to service `apiserver` in our service mesh.

This configuration assumes you have the sample application deployed from Chapter 2, but if you do not, run this to install the `apigateway` and `catalog` services with Istio's service proxy injected alongside the services. This command is meant to be run from the root of the source code for this book:

```
$  kubectl config set-context $(kubectl config current-context) \
 --namespace=istioinaction
$  istioctl kube-inject -f services/catalog/kubernetes/catalog.yaml \
| kubectl apply -f -
$  istioctl kube-inject -f services/apigateway/kubernetes/apigateway.yaml \
| kubectl apply -f -
```

Once all the pods are ready, you should see something like this:

```
$  kubectl get pod
NAME                          READY     STATUS    RESTARTS   AGE
apigateway-bd97b9bb9-q9g46    2/2       Running   18         19d
catalog-786894888c-8lbk4      2/2       Running   8          6d
```

Verify that your `Gateway` and `VirtualService` are installed correctly:

```
$  kubectl get gateway
NAME                CREATED AT
coolstore-gateway   2h

$  kubectl get virtualservice
NAME                     CREATED AT
apigateway-vs-from-gw    11m
```

Now, let's try to call the gateway and verify the traffic is allowed into the cluster. Remember we

are using the Docker for Desktop approach where the Istio ingress gateway is available on port `80` on `localhost`. If using a cloud service or NodePort service, you'll need to figure out what that external IP is. For example in chapter 2 we saw one way to get the correct host for the ingress gateway exposed on a public load balancer looks like this:

```
$  URL=$(kubectl -n istio-system get svc istio-ingressgateway\
-o jsonpath='{.status.loadBalancer.ingress[0].ip}')
```

Once you have the correct endpoint, you can run something similar to this (remember, `localhost` is on Docker for Desktop):

```
$  curl http://localhost/api/catalog
```

We should actually see no response. Why is that? If we take a closer look at the call by printing the headers, we should see that the `Host` header we sent in is **not** a host that the gateway recognizes.

```
$  curl -v $URL/api/catalog
*   Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 80 (#0)
> GET /api/catalog HTTP/1.1
> Host: localhost
> User-Agent: curl/7.54.0
> Accept: */*
>
< HTTP/1.1 404 Not Found   ❷
< date: Tue, 21 Aug 2018 16:08:28 GMT
< server: envoy
< content-length: 0
<
* Connection #0 to host 192.168.64.27 left intact
```

❶ Host

❷ Not found

The Istio Gateway, nor any of the routing rules we declared in `VirtualService` knows anything about `Host: localhost:80` but it does know about virtual host `apiserver.istioinaction.io`. Let's override the `Host` header on our command line and then the call should work:

```
$  curl http://localhost/api/catalog -H "Host: apiserver.istioinaction.io"
```

Now you should see a successful response.

4.2.3 Overall view of traffic flow

In the previous subsections, we got hands on with the `Gateway` and `VirtualService` resources from Istio. The `Gateway` resource defines our ports, protocols, and virtual hosts that we wish to listen for at the edge of our service mesh cluster. The `VirtualService` resources define where traffic should go once it's allowed in at the edge. In Figure 4.7 we see the full end-to-end flow:

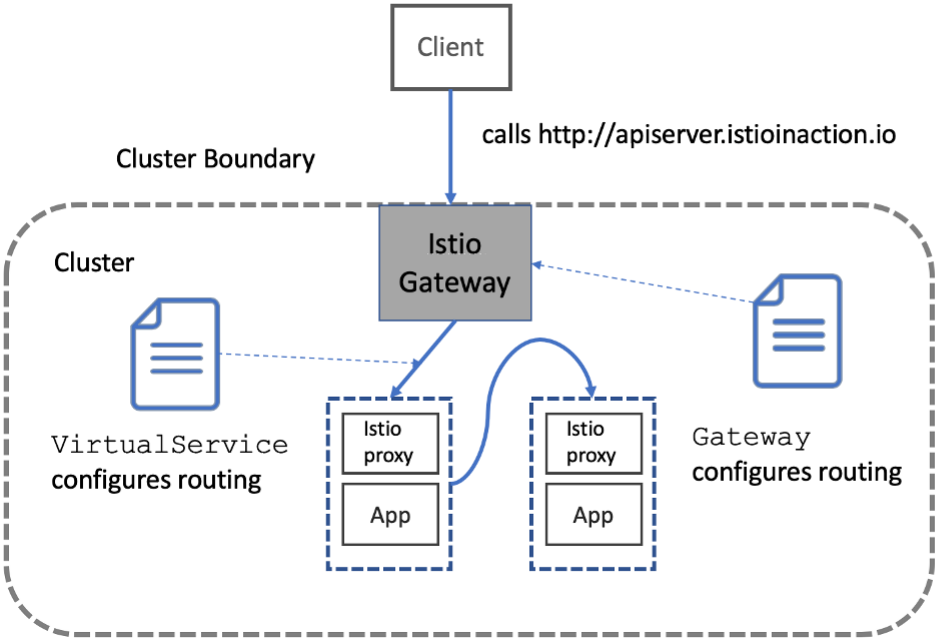


Figure 4.7 Flow of traffic from client outside of service mesh/cluster to services inside the service mesh through the ingress gateway

4.2.4 Istio Gateway vs Kubernetes Ingress

When running on Kubernetes, you may ask "why doesn't Istio just use the Kubernetes Ingress resource to specify ingress?". In some of Istio's early releases there was support for using Kubernetes Ingress, but there are significant drawbacks with the Kubernetes Ingress specification.

The first issue is that the Kubernetes Ingress is a very simple specification geared toward HTTP workloads. There are implementations of Kubernetes Ingress (like NGINX, Heptio Contour, etc) however each of them is geared toward HTTP traffic. In fact, Ingress specification really only considers port `80` and port `443` as ingress points. This severely limits the types of traffic a cluster operator can allow into the service mesh. For example, if you have Kafka or NATS.io workloads, you may wish to expose direct TCP connections to these message brokers. Kubernetes Ingress doesn't allow for that.

Second, the Kubernetes Ingress resource is severely underspecified. There is no common way to specify complex traffic routing rules, traffic splitting, or things like traffic shadowing. The lack of specification in this area causes each vendor to re-imagine how best to implement configurations for each type of Ingress implementation (HAProxy, Nginx, etc).

Lastly, since things are underspecified, the way most vendors chose to expose configuration is through bespoke annotations on deployments. The annotations between vendors varied and were not portable, and if Istio continued that trend there would have been many more annotations to account for all the power of Envoy as an edge gateway.

Ultimately, Istio decided on a clean slate for building ingress patterns and specifically separating out the layer 4 (transport) and layer 5 (session) properties from the layer 7 (application) routing concerns. Istio `Gateway` handles the L4 and L5 concerns while `VirtualService` handles the L7 concerns. In fact many mesh or gateway providers have also built their own API for ingress and the Kubernetes community is also working on a revised ingress API. When the new ingress spec comes to Kubernetes, Istio will support that as well.

4.3 Securing Gateway traffic

So far, we've shown how to expose basic HTTP services with the Istio gateway using the `Gateway` and `VirtualService` resources. When connecting services from outside of a cluster (let's say, the public internet) to those running inside a cluster, one of the basic capabilities of the ingress gateway in a system is to secure traffic and help to establish trust in the system. One way we can begin to secure our traffic is to give clients confidence that the service they're hoping to communicate with is indeed the service it claims to be. Additionally, we want to exclude anyone from eavesdropping on our communication, so we should encrypt the traffic.

Istio's gateway implementation allows us to terminate incoming TLS/SSL traffic, pass it through to the backend services, redirect any non-TLS traffic to the proper TLS ports as well as implement mutual TLS. We'll take a look at each of these capabilities in this section.

4.3.1 HTTP traffic with TLS

To prevent man-in-the-middle (MITM) attacks and to encrypt all traffic coming into the service mesh, we can set up TLS on the Istio gateway so that any incoming traffic is served over HTTPS (for HTTP traffic; we'll cover non-HTTP traffic in the next sections). MITM attacks occur when a client connects to a service, but instead of the service it intends, it connects to a impostor service claiming to be the intended service. The impostor service can gain access to the communication including sensitive information. TLS helps to mitigate this attack.

To enable TLS/HTTPS for our ingress traffic, we need to specify the correct private keys and certificates that the gateway should use. As a quick reminder, the certificate that the server presents is how it announces its identity to any clients. The certificate is basically the server's public key that has been signed by a reputable authority also known as a Certificate Authority (CA). For a client to trust that the server's certificate is indeed valid, it must have first installed the CA issuer's certificate. This way a client can verify that the certificate is signed by a CA that it trusts. The private key that the server uses is how it encrypts traffic it sends to the client who can then decrypt the traffic with the server's public key (which can be found in the certificate).

NOTE **TLS encryption is beyond the scope of this book**

The previous statement wasn't entirely correct: the client doesn't technically decrypt the traffic with the server's public key. The TLS handshake includes a more sophisticated protocol that combines the public/private keys (asymemntric) for initial communication and then create a session key (symmetric) that is used for the TLS session to encrypt/decrypt traffic. See the book "Bulletproof SSL and TLS" for a more in-depth treatment of TLS.

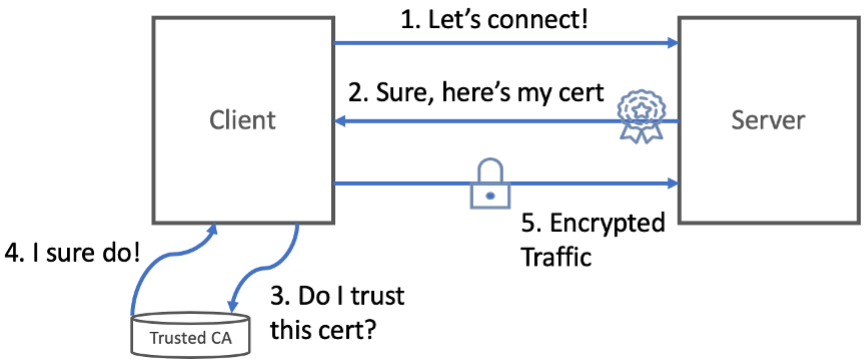


Figure 4.8 Basic model of how TLS is established between a client and server

Before we can configure the default `istio-ingressgateway` to use certificates and keys, we need to create them as Kubernetes secrets.

Let's start by creating the `apiserver-credential` secret. Double check you're in the `chapters/chapter4` directory and run:

```
$  kubectl create -n istio-system secret tls apiserver-credential \
--key certs/3_application/private/apiserver.istioinaction.io.key.pem \
--cert certs/3_application/certs/apiserver.istioinaction.io.cert.pem

secret/apiserver-credential created
```

In this step, we create the secret in the `istio-system` namespace. At the time of writing (Istio 1.7), the secret that's used for TLS in the gateway can only be retrieved if it's in the same namespace as the Istio ingress gateway. The default gateway is run in the `istio-system` namespace, so that's where we put the secret. We could run the ingress gateway in a different namespace, but the secret would still have to be in that namespace. In future versions of Istio this constraint may be relaxed.

Now we can configure our Istio `Gateway` resource to use the certificates and keys:

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: coolstore-gateway
spec:
  selector:
```

```
    istio: ingressgateway
  servers:
  - port:
      number: 80   ❶
      name: http
      protocol: HTTP
    hosts:
    - "apiserver.istioinaction.io"
  - port:
      number: 443   ❷
      name: https
      protocol: HTTPS
    tls:
      mode: SIMPLE   ❸
      credentialName: apiserver-credential
    hosts:
    - "apiserver.istioinaction.io"
```

❶ simple HTTP

❷ secured HTTPS

❸ standard server TLS pointing to Kubernetes secret

As we can see in our `Gateway` resource in listing XX, we've opened port `443` on our ingress gateway, and we've specified its protocol to be HTTPS. Additionally, we've added a `tls` section to our gateway configuration where we've specified the locations to find the certificates and keys to use for TLS. Note, these are the same locations that were mounted into the `istio-ingressgateway` that we saw earlier.

Let's replace our gateway with this new `Gateway` resource:

```
$  kubectl apply -f coolstore-gw-tls.yaml

gateway.networking.istio.io/coolstore-gateway replaced
```

NOTE **Use the correct host and ports for your environment**

The commands in this book assumes we're using Docker for Desktop but if you're using your own Kubernetes cluster (or a public-cloud hosted one), you can use those values directly. For example, on GKE, you can figure out the HOST IP by using the cloud loadbalancer's public IP as shown when looking at the Kubernetes services:

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

NAME                    TYPE           CLUSTER-IP     EXTERNAL-IP
istio-ingressgateway    LoadBalancer   10.12.2.78     35.233.243.32
istio-pilot             ClusterIP      10.12.15.206   <none>
```

In this case `35.233.243.32` would be used for `HTTPS_HOST`. You can then just use the real ports (ie, `80` and `443`) for HTTP and HTTPS respectively.

If we call the service like we did in the previous section, by passing in the proper `Host` header, we should see something like this (note, we use `https://` in the URL):

```
$  curl -v -H "Host: apiserver.istioinaction.io" https://localhost/api/catalog

*   Trying 192.168.64.27...
* TCP_NODELAY set
* Connected to 192.168.64.27 (192.168.64.27) port 31390 (#0)
* ALPN, offering http/1.1
* Cipher selection: ALL:!EXPORT:!EXPORT40:!EXPORT56:!aNULL:!LOW:!RC4:@STRENGTH
* successfully set certificate verify locations:
*   CAfile: /usr/local/etc/openssl/cert.pem  (1)
  CApath: /usr/local/etc/openssl/certs
* TLSv1.2 (OUT), TLS header, Certificate Status (22):
* TLSv1.2 (OUT), TLS handshake, Client hello (1):
* OpenSSL SSL_connect: SSL_ERROR_SYSCALL in connection to 192.168.64.27:31390
* Closing connection 0
curl: (35) OpenSSL SSL_connect: SSL_ERROR_SYSCALL in connection to 192.168.64.27:31390
```

(1) default cert chain

This means the Certificate presented by the server cannot be verified using the default CA certificate chains. Let's pass in the proper CA certificate chain to our `curl` client:

```
$  curl -v -H "Host: apiserver.istioinaction.io" https://localhost/api/catalog \
--cacert certs/2_intermediate/certs/ca-chain.cert.pem

*   Trying 192.168.64.27...
* TCP_NODELAY set
* Connected to 192.168.64.27 (192.168.64.27) port 31390 (#0)
* ALPN, offering http/1.1
* Cipher selection: ALL:!EXPORT:!EXPORT40:!EXPORT56:!aNULL:!LOW:!RC4:@STRENGTH
* successfully set certificate verify locations:
*   CAfile: certs/2_intermediate/certs/ca-chain.cert.pem
  CApath: /usr/local/etc/openssl/certs
* TLSv1.2 (OUT), TLS header, Certificate Status (22):
* TLSv1.2 (OUT), TLS handshake, Client hello (1):
* OpenSSL SSL_connect: SSL_ERROR_SYSCALL in connection to 192.168.64.27:31390
* Closing connection 0
curl: (35) OpenSSL SSL_connect: SSL_ERROR_SYSCALL in connection to 192.168.64.27:31390
```

The client still cannot verify the certificate! This is because the server certificate is issued for `apiserver.istioinaction.io` and we're calling the Docker for Desktop host (localhost in this case). We can use a `curl` parameter called `--resolve` that lets us call the service as though it was at `apiserver.istioinaction.io` but then tell curl to use localhost:

```
$  curl -H "Host: apiserver.istioinaction.io" \
https://apiserver.istioinaction.io:443/api/catalog \
--cacert certs/2_intermediate/certs/ca-chain.cert.pem \
--resolve apiserver.istioinaction.io:443:127.0.0.1
```

Note, we use the `--resolve` flag to map the hostname and port in the certificate to the real IP that we're using. With Docker for Desktop, the ingress runs on `localhost` as we've seen. If you are using a cloud-provided load balancer you can replace `12.0.0.1` with the appropriate IP. Now we should see a proper `HTTP/1.1 200` response and the JSON payload for the products list. As a client we're verifying the server is who it says it is by trusting the CA that signed the certificate and we're able to encrypt the traffic to the server by using this certificate.

NOTE **Will curl work for you?**

Note, for curl to work in this section, you need to make sure it supports TLS and you can add your own CA certificates to override the default. Not all builds of curl support TLS. For example, in some versions of curl on Mac OS X, CA certificates can only come from the Apple Keychain. Newer builds of `curl` should have the proper SSL libraries and should work for you, but what you want to see is something about your SSL library (OpenSSL, LibreSSL, etc) when you type this:

```
curl --version | grep -i SSL
```

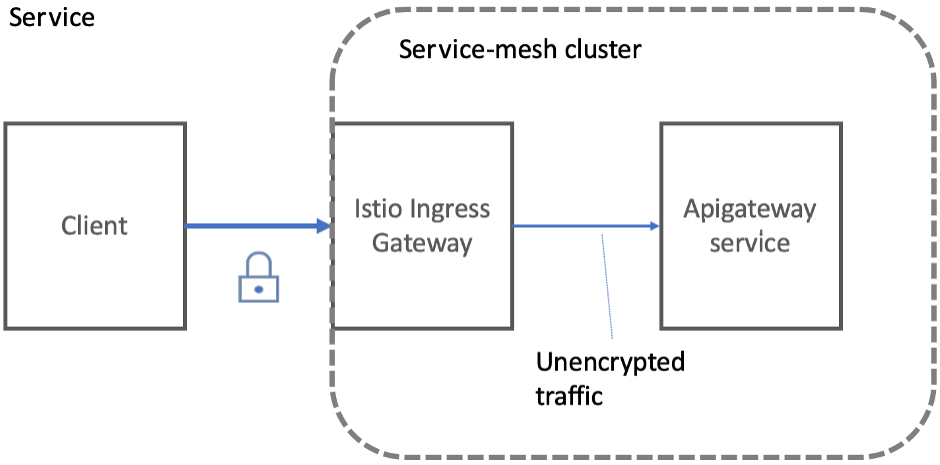


Figure 4.9 Secured traffic from outside world to the Istio ingress gateway component; traffic within the mesh is not secured yet

At this point, we've secured traffic by encrypting it to the Istio ingress gateway, which terminates the TLS connection and then sends the traffic to the backend `apigateway` service running in our service mesh. The hop between the `istio-ingressgateway` component and the `apigateway` service is **not** encrypted or secured in anyway **yet**. We will cover securing internal service-mesh traffic in Chapter XX.

4.3.2 HTTP redirect to HTTPS

We set up TLS in the previous section, but what if we want to force all traffic to always use TLS? In the previous section we could have used both `http://` and `https://` to access our service through the ingress gateway, but in this section we want to force all traffic to use `HTTPS`. To do that, we have to modify our `Gateway` resource slightly to force a redirect for `HTTP` traffic.

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: custom-coolstore-gateway
spec:
  selector:
    istio: custom-ingressgateway
  servers:
```

```
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "apiserver.istioinaction.io"
    tls:
      httpsRedirect: true   ❶
  - port:
      number: 443
      name: https
      protocol: HTTPS
    tls:
      mode: SIMPLE
      serverCertificate: /etc/istio/ingressgateway-certs/tls.crt
      privateKey: /etc/istio/ingressgateway-certs/tls.key
    hosts:
    - "apiserver.istioinaction.io"
```

❶ Redirect HTTP to HTTPS

If we update our `Gateway` to use the above configuration, we can limit all traffic to only `HTTPS`.

```
$  kubectl apply -f coolstore-gw-tls-redirect.yaml
gateway.networking.istio.io/coolstore-gateway replaced
```

Now if we call the ingress gateway on the `HTTP` port, we should see something like this:

```
$  curl -v http://localhost/api/catalog -H "Host: apiserver.istioinaction.io"
*   Trying 192.168.64.27...
* TCP_NODELAY set
* Connected to 192.168.64.27 (192.168.64.27) port 31380 (#0)
> GET /api/catalog HTTP/1.1
> Host: apiserver.istioinaction.io
> User-Agent: curl/7.61.0
> Accept: */*
>
< HTTP/1.1 301 Moved Permanently   ❶
< location: https://apiserver.istioinaction.io/api/catalog
< date: Wed, 22 Aug 2018 21:01:29 GMT
< server: envoy
< content-length: 0
<
* Connection #0 to host 192.168.64.27 left intact
```

❶ HTTP 301 redirect

Now we can expect all traffic going to our ingress gateway to always be encrypted.

4.3.3 HTTP traffic with mutual TLS

In the previous section, we used standard TLS to allow the server to prove its identity to the client, but what if we want our cluster to verify who the clients are before we accept any traffic from outside the cluster? In the simple TLS scenario, the server sends its public certificate to the client and the client verifies that it trusts the CA that signed the server's certificate. What we want to do is have the client send its public certificate and let the server verify that it trusts it. When the client and server each verify the other's certificates and use these to encrypt traffic, we call this mutual TLS (mTLS).

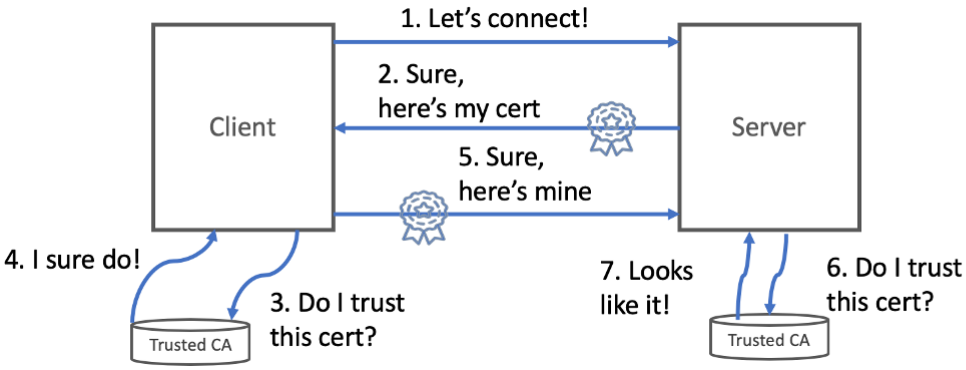


Figure 4.10 Basic model of how mTLS is established between a client and server

To configure the default `istio-ingressgateway` to participate in a mutual TLS connection, we need to give it a set of Certificate Authority (CA) certificates to use to verify a client's certificate. Just like we did in the previous section, we need to make this CA certificate (or certificate chain more specifically) available to the `istio-ingressgateway` with a Kubernetes secret.

Let's start by configuring the `istio-ingressgateway-ca-certs` secret with the proper CA certificate chain. To do so, verify you're in the `chapters/chapter4` folder and run the following:

```
$  kubectl create -n istio-system secret generic apiserver-credential-mtls \
--from-file=tls.key=certs/3_application/private/apiserver.istioinaction.io.key.pem \
--from-file=tls.crt=certs/3_application/certs/apiserver.istioinaction.io.cert.pem \
--from-file=ca.crt=certs/2_intermediate/certs/ca-chain.cert.pem

secret/apiserver-credential-mtls created
```

Now let's update the Istio `Gateway` resource to point to the location of the CA certificate chain as well as configure the expected protocol to be mutual TLS:

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: coolstore-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "apiserver.istioinaction.io"
  - port:
      number: 443
      name: https
      protocol: HTTPS
    tls:
      mode: MUTUAL    ❶
      credentialName: apiserver-credential-mtls    ❷
    hosts:
    - "apiserver.istioinaction.io"
```

1. Configured for mTLS
2. Credentials with a trusted CA configured

Let's replace the `Gateway` configuration with this new updated version:

```
$  kubectl apply -f coolstore-gw-mtls.yaml

gateway.networking.istio.io/coolstore-gateway replaced
```

Now if we try to call the ingress gateway the same way we did in the previous section (i.e., assuming simple TLS), we should see the call rejected:

```
$  curl -H "Host: apiserver.istioinaction.io" \
https://apiserver.istioinaction.io:443/api/catalog \
--cacert certs/2_intermediate/certs/ca-chain.cert.pem \
--resolve apiserver.istioinaction.io:443:127.0.0.1

curl: (35) error:14094410:SSL routines:ssl3_read_bytes:sslv3 alert handshake failure
```

NOTE **Istio Gateway SDS**

Istio Gateway gets the certificates from the SDS (Secret Discovery Service) that's built into the `istio-agent` process that's used to start the `istio-proxy`. SDS is a dynamic API that should automatically propagate the updates. Note, if you're not seeing the new caCertificates configuration take effect, you may wish to "bounce" the istio-ingressgateway pod. To do so

```
kubectl delete po -n istio-system -l app=istio-ingressgateway
```

We see this call getting rejected because the SSL handshake wasn't successful. We are only passing the CA certificate chain to the `curl` command, we need to also pass the client's certificate and private key. With `curl` we can do it by passing the `--cert` and `--key` parameters like this:

```
$  curl -H "Host: apiserver.istioinaction.io" \
https://apiserver.istioinaction.io:443/api/catalog \
--cacert certs/2_intermediate/certs/ca-chain.cert.pem \
--resolve apiserver.istioinaction.io:443:127.0.0.1 \
--cert certs/4_client/certs/apiserver.istioinaction.io.cert.pem \
--key certs/4_client/private/apiserver.istioinaction.io.key.pem
```

Now we should see a proper `HTTP/1.1 200` response and the JSON payload for the products list. The client is both validating the server's certificate as well as sending its own certificate for validation to achieve mutual TLS.

4.3.4 Serving multiple virtual hosts with TLS

Istio's ingress gateway can serve multiple virtual hosts each with its own certificate and private key from the same HTTPS port (i.e., port 443). To do that, we just add multiple entries for the same port and the same protocol. For example, we can add multiple entries for both the `apiserver.istioinaction.io` and `catalog.istioinaction.io` services each with their own certificate and key pair. An Istio `Gateway` resource that describes multiple virtual hosts served with HTTPS looks like this:

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: coolstore-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 443   ❶
      name: https-apiserver
      protocol: HTTPS
    tls:
      mode: SIMPLE
      credentialName: apiserver-credential
    hosts:
    - "apiserver.istioinaction.io"
  - port:
      number: 443   ❷
      name: https-catalog
      protocol: HTTPS
    tls:
      mode: SIMPLE
      credentialName: catalog-credential
    hosts:
    - "catalog.istioinaction.io"
```

❶ first entry

❷ second entry

Notice that both of the entries each listen on port `443`, each serve the `HTTPS` protocol, but they have different names. One is named `https-apiserver`, and the other is named `https-catalog`. Each has its own unique certificates and keys that are used for the specific virtual host it serves. To put this into action, we need to add these new certificates and keys. Let's create them:

```
$  kubectl create -n istio-system secret tls catalog-credential \
--key certs2/3_application/private/catalog.istioinaction.io.key.pem  \
--cert certs2/3_application/certs/catalog.istioinaction.io.cert.pem
```

Now let's update the gateway configuration:

```
$  kubectl apply -f coolstore-gw-multi-tls.yaml
gateway.networking.istio.io/coolstore-gateway replaced
```

Lastly, we need to add a `VirtualService` for the `catalog` service that we'll be exposing through this ingress gateway:

```
$  kubectl apply -f catalog-vs.yaml
```

Now that we've updated the `istio-ingressgateway`, let's give it a try. Calling `apiserver.istioinaction.io` should work just like it did in the simple TLS section:

```
$  curl -H "Host: apiserver.istioinaction.io" \
https://apiserver.istioinaction.io:443/api/catalog \
--cacert certs/2_intermediate/certs/ca-chain.cert.pem \
--resolve apiserver.istioinaction.io:443:127.0.0.1
```

Now when we call the `catalog` service through the Istio gateway, let's use different certificates:

```
$  curl -H "Host: catalog.istioinaction.io" \
https://catalog.istioinaction.io:$HTTPS_PORT/items \
--cacert certs2/2_intermediate/certs/ca-chain.cert.pem \
--resolve catalog.istioinaction.io:$HTTPS_PORT:$HTTPS_HOST

curl -H "Host: catalog.istioinaction.io" \
https://catalog.istioinaction.io:443/items \
--cacert certs2/2_intermediate/certs/ca-chain.cert.pem \
--resolve catalog.istioinaction.io:443:127.0.0.1
```

Both calls should succeed with the same response. You may be wondering how does the Istio ingress gateway know which certificate to present depending who's calling? In other words, there's only a single port opened for these connections, how does it know which service the client is trying to access and which certificate corresponds with that service? The answer lies in an extension to TLS called Server Name Indication (SNI). Basically, when a HTTPS connection is created, the client first identifies which service it's trying to reach using the **ClientHello** part of the TLS handshake. Istio's Gateway (Envoy specifically) implements SNI on TLS which is how it's able to present the correct cert and route to the correct service. For more on SNI, see XXX

In this section we successfully exposed to different virtual hosts and served each with its own unique certificate through the same HTTPS port. In the next section, we'll take a look at TCP traffic.

4.4 TCP traffic

Istio's gateway is powerful enough to serve not only HTTP/HTTPS traffic, but any traffic accessible via TCP. For example, we can expose a database (like MongoDB) or a message queue like Kafka through the ingress gateway. When Istio treats the traffic as plain TCP, we do not get as many useful features like retries, request-level circuit breaking, complex routing, etc. This is simply because Istio cannot tell what protocol is being used (unless a specific protocol that Istio understands is used — like MongoDB). Let's take a look at how to expose TCP traffic through the Istio Gateway so that clients on the outside of the cluster can communicate with those running inside the cluster.

4.4.1 Exposing TCP ports on the Istio Gateway

The first thing we need to do is create a TCP-based service within our service mesh. For this example, we'll use the simple `echo` service from https://github.com/cjimti/go-echo/. This simple TCP service will allow us to login with a simple TCP client like `telnet` and issue commands that should be displayed back to us.

Let's deploy the TCP service and inject the Istio service proxy next to it. Recall that we're pointing to the `istioinaction` namespace.

```
$  kubectl config set-context $(kubectl config current-context) \
 --namespace=istioinaction
$  istioctl kube-inject -f echo.yaml | kubectl apply -f -

deployment.apps/tcp-echo-deployment created
service/tcp-echo-service created
```

Next, we should create an Istio `Gateway` resource that exposes a specific non-HTTP port for this service. In the following example, we expose port `31400` on the default `istio-ingressgateway` . Just like with the HTTP ports, 80 and 443, this TCP port `31400` must be made available either as a NodePort or as a cloud LoadBalancer. In our examples running on minikube, this is exposed as a NodePort running on `31400`:

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: echo-tcp-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 31400   ❶
      name: tcp-echo
      protocol: TCP   ❷
    hosts:
    - "*"   ❸
```

❶ Port to expose

❷ Expected protocol

❸ For any hosts

Let's create the gateway:

```
$  kubectl apply -f gateway-tcp.yaml

gateway.networking.istio.io/echo-tcp-gateway created
```

Now that we've exposed a port on our ingress gateway, we need to route the traffic to the `echo` service. To do that, we'll use the `VirtualService` resource like we did in the previous sections. Note, for tcp traffic like that, we must match on the incoming port, in this case port `31400`:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: tcp-echo-vs-from-gw
spec:
  hosts:
  - "*"
  gateways:
  - echo-tcp-gateway   ❶
  tcp:
  - match:
    - port: 31400   ❷
    route:
    - destination:
        host: tcp-echo-service   ❸
        port:
          number: 2701
```

❶ Which gateway

❷ Match on port

❸ Where to route

Let's create the virtual service:

```
$  kubectl apply -f echo-vs.yaml

virtualservice.networking.istio.io/tcp-echo-vs-from-gw created
```

NOTE **If you're running in a public cloud**

If you're running in a public cloud, or a cluster that creates a LoadBalancer for the `istio-ingressgateway` service, and you're not able to connect as shown below, you may need to explicitly add a port to the `istio-ingressgateway` service on port 31400 and targetPort 31400 for this to work correctly. By default, Istio 1.7 does add this port to the `istio-ingressgateway` service, but you may want to double check.

Now that we have exposed a port on our ingress gateway, and set up routing, we should be able to connect with a very simple `telnet` command:

```
$  telnet localhost 31400

Trying 192.168.64.27...
Connected to kubebook.
Escape character is '^]'.
Welcome, you are connected to node minikube.
Running on Pod tcp-echo-deployment-6fbccd8485-m4mqq.
In namespace istioinaction.
With IP address 172.17.0.11.
Service default.
```

As you type anything into the console and hit Return/Enter, you should see your text replayed back to you:

```
hello there
hello there
```

```
by now
by now
```

To quit from `telnet` press `CTRL+]` and type `quit` and then hit Return/Enter.

4.4.2 Traffic routing with SNI Passthrough

In the previous section, we learned how to use the Istio gateway functionality to accept and route non HTTP traffic, specifically, applications that may communicate over a TCP protocol that is very application specific. Earlier we saw how to route HTTPS traffic and present certain certificates depending on the SNI hostname. In this last section, we'll look at a combination of these two capabilities. We will see how to route TCP traffic based on SNI hostname without terminating the traffic on the Istio ingress gateway. All the gateway will do is inspect the SNI headers and route the traffic to the specific backend which will then terminate the TLS connection. The connection will "pass through" the gateway and be handled by the actual service, not the gateway.

This opens the door for a much wider swath of applications that can participate in the service mesh including TCP over TLS services like databases, message queues, caches, etc and even legacy applications that expect to handle and terminate HTTPS/TLS traffic itself. To see this in action, let's take a look at a Gateway definition that is configured to use `PASSTHROUGH` as its routing mechanism:

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: sni-passthrough-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 31400   ❶
      name: tcp-sni
      protocol: TLS
    hosts:
    - "simple-sni-1.istioinaction.io"   ❷
    tls:
      mode: PASSTHROUGH   ❸
```

❶ Open a specific, non HTTP, port

❷ Associate this host with the port

❸ Treat this as passthrough traffic

In our sample application we configure the application to terminate TLS for the HTTPS connection using certificates. This means we don't need the ingress gateway to do anything with the connection. We won't need to configure any certificates on the gateway like we did in previous section.

Let's get started by deploying our sample applications that terminates TLS. Remember we are in

the `chapters/chapter4` directory of the source code for this book and we are defaulting to the `istioinaction` namespace in Kubernetes:

```
$  istioctl kube-inject -f sni/simple-tls-service-1.yaml \
| kubectl apply -f -
```

Next, let's deploy our Gateway resource that opens port `31400`. Before we do that, since we'll be using the same port as the TCP section earlier in this chapter, let's make sure to delete any Gateways that already use that port:

```
$  kubectl delete gateway echo-tcp-gateway -n istioinaction
```

Now let's apply the pass through Gateway:

```
$  kubectl apply -f sni/passthrough-sni-gateway.yaml
```

At this point, we've opened the `31400` port on the Istio ingress gateway, but as you'll recall from previous sections, we will also need to specify routing rules with a VirtualService to get the traffic from the gateway to the service. Let's take a look at what that VirtualService would look like:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: simple-sni-1-vs
spec:
  hosts:
  - "simple-sni-1.istioinaction.io"
  gateways:
  - sni-passthrough-gateway
  tls:
  - match:   ❶
    - port: 31400
      sniHosts:
      - simple-sni-1.istioinaction.io
    route:
    - destination:   ❷
        host: simple-tls-service-1
        port:
          number: 80   ❸
```

❶ Matching clause on specific port and host

❷ Routing destination if traffic matched

❸ Route to the service port

Let's create the VirtualService:

```
$  kubectl apply -f sni/passthrough-sni-vs-1.yaml
```

Now let's try call the Istio ingress gateway on port `31400`

```
$  curl -H "Host: simple-sni-1.istioinaction.io" \
 https://simple-sni-1.istioinaction.io:31400/ \
 --cacert sni/simple-sni-1/2_intermediate/certs/ca-chain.cert.pem \
```

```
 --resolve simple-sni-1.istioinaction.io:31400:127.0.0.1

{
  "name": "simple-tls-service-1",
  "uri": "/",
  "type": "HTTP",
  "ip_addresses": [
    "10.1.0.63"
  ],
  "start_time": "2020-09-03T20:09:08.129404",
  "end_time": "2020-09-03T20:09:08.129846",
  "duration": "441.5µs",
  "body": "Hello from simple-tls-service-1!!!",
  "code": 200
}
```

Our call from `curl` went to the Istio ingress gateway, traversed through without termination, and ended up on the simple example service `simple-tls-service-1`. To make the routing more apparent, let's deploy a second service with different certificates and route based on the SNI host:

```
$  istioctl kube-inject -f sni/simple-tls-service-2.yaml \
| kubectl apply -f -
```

Let's see what the Gateway looks like:

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: sni-passthrough-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 31400
      name: tcp-sni-1
      protocol: TLS
    hosts:
    - "simple-sni-1.istioinaction.io"
    tls:
      mode: PASSTHROUGH
  - port:
      number: 31400
      name: tcp-sni-2
      protocol: TLS
    hosts:
    - "simple-sni-2.istioinaction.io"
    tls:
      mode: PASSTHROUGH
```

Let's apply this Gateway and VirtualService

```
$  kubectl apply -f sni/passthrough-sni-gateway-both.yaml
$  kubectl apply -f sni/passthrough-sni-vs-2.yaml
```

Now we'll call again to the same ingress gateway port with different hostname and watch how it gets routed to the correct service:

```
$  curl -H "Host: simple-sni-2.istioinaction.io" \
```

```
 https://simple-sni-2.istioinaction.io:31400/ \
 --cacert sni/simple-sni-2/2_intermediate/certs/ca-chain.cert.pem \
 --resolve simple-sni-2.istioinaction.io:31400:127.0.0.1

{
  "name": "simple-tls-service-2",
  "uri": "/",
  "type": "HTTP",
  "ip_addresses": [
    "10.1.0.64"
  ],
  "start_time": "2020-09-03T20:14:13.982951",
  "end_time": "2020-09-03T20:14:13.984547",
  "duration": "1.5952ms",
  "body": "Hello from simple-tls-service-2!!!",
  "code": 200
}
```

Notice how this response in the "body" field indicates this request is served by the `simple-tls-service-2` service.

4.5 Summary

In this chater we looked at why it's important to have very fine-grained control over what traffic enters your service-mesh cluster and how to use the Istio `Gateway` and `VirtualService` constructs to do this. Specifically we saw these ways to restrict traffic:

- By specific host over HTTP
- By simple TLS/HTTPS to encrypt traffic and prevent man-in-the-middle attacks
- Mutual TLS to provide identity to both server and client about who's making the connections
- Basic TCP routing for non-HTTP traffic
- SNI/TLS for TCP connections for securing TCP connections

We started to explore `VirtualService` resource files to configure how routing happens at the ingress of our cluster. In the next chapter, we'll expand on our understanding of `VirtualService` resources for the purposes of more powerful routing within the service mesh and how this control helps us control new deployments, route around failures, and implement powerful testing capabilities.

Things to consider:

- Adding more information about securing tha gateway with a PSP
- Running the gateway on other ports
- Running multiple /your own gateway
- Setting up with a cloud LB? *

Maybe with this PSP? apiVersion: policy/v1beta1 kind: PodSecurityPolicy metadata: name: psp-restricted spec: allowPrivilegeEscalation: false forbiddenSysctls: - '*' fsGroup: ranges: - max: 65535 min: 1 rule: MustRunAs requiredDropCapabilities: - ALL runAsUser: rule: MustRunAsNonRoot runAsGroup: rule: MustRunAs ranges: - min: 1 max: 65535 seLinux: rule:

RunAsAny supplementalGroups: ranges: - max: 65535 min: 1 rule: MustRunAs volumes: - configMap - emptyDir - projected - secret - downwardAPI - persistentVolumeClaim

5 Traffic control: fine-grained traffic routing

This chapter covers:

- Traffic routing basics
- Traffic shifting when doing a new release
- Mirroring traffic to reduce the risk of a new release
- Controlling traffic leaving a cluster

In the previous chapter, we saw how to get traffic into a cluster and what considerations we needed to account for when doing so. Once a request makes it into our cluster, how does it get routed to the appropriate service to handle the request? How do services that live within the cluster communicate with other services that live within the same cluster or sometimes that live outside the cluster? Lastly, and most importantly, when we make changes to a service and introduce new versions, how do we safely expose our clients and customers to these changes with minimal disruption and impact?

As we've seen Istio service proxies broker the communication between services within the service mesh cluster and gives us a point of control for traffic. Istio allows us to finely control traffic flowing between applications down to the individual request. In this chapter, we'll take a look at why you might want to do that, how to do it, and what benefits you should achieve when utilizing these capabilities.

5.1 Reducing the risk of deploying new code

In Chapter 1, we introduced the scenario of ACME company moving to a cloud platform and trying to adopt practices that helped reduce their risk of deploying code. One of those patterns they tried was "blue/green" deployments when they wanted to introduce changes to their applications. With a blue-green deployment, they would take v2 (green) of their service they wanted to change and deploy it in production next to v1 (blue).

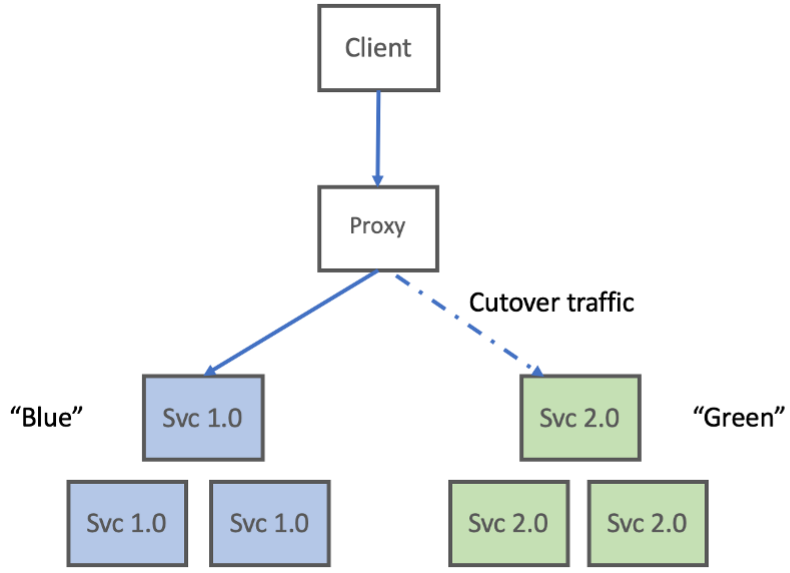


Figure 5.1 In a blue-green deployment, blue is the currently released software and when we release the new software we cutover traffic to green version

When they wanted to release to their customers, they would cut over the traffic to v2 (green). This approach helped them reduce outages when doing deployments because if there were any issues, they could always just cut back to v1 (blue) of their service. Blue-green deployments help, but when we do the cut-over from v1 to v2, we still experience a "big bang" release in which we release all the code changes at once. Let's see how we can further reduce the risk of doing deployments. First, we should clarify what we mean by "deployment" and "release".

5.1.1 Deployment vs Release

Let's take a fictitious `catalog` service to help illustrate the differences between a **deployment** and a **release**. Let's say we have v1 of the `catalog` service running in production at the moment. If we want to introduce a code change to the `catalog` service, we expect to build it using our continuous-integration system, tag it with a new version (let's say v1.1), and then deploy it and test it in pre-production environments. Once we can validate these changes in pre-production, and have necessary approvals, we can begin to bring this new version v1.1 to production.

When we do a **deployment** to production, we install the new code onto production resources (servers, containers, etc) but we do not send any traffic to it. Doing a **deployment** to production

should *not* impact users running in production because it doesn't take any user requests. At this point, we can run some tests on this new deployment running in production to verify it appears to work as we expect. We should have metric and log collection enabled at this point so we can use these signals to inform our confidence that our new deployment is behaving as expected.

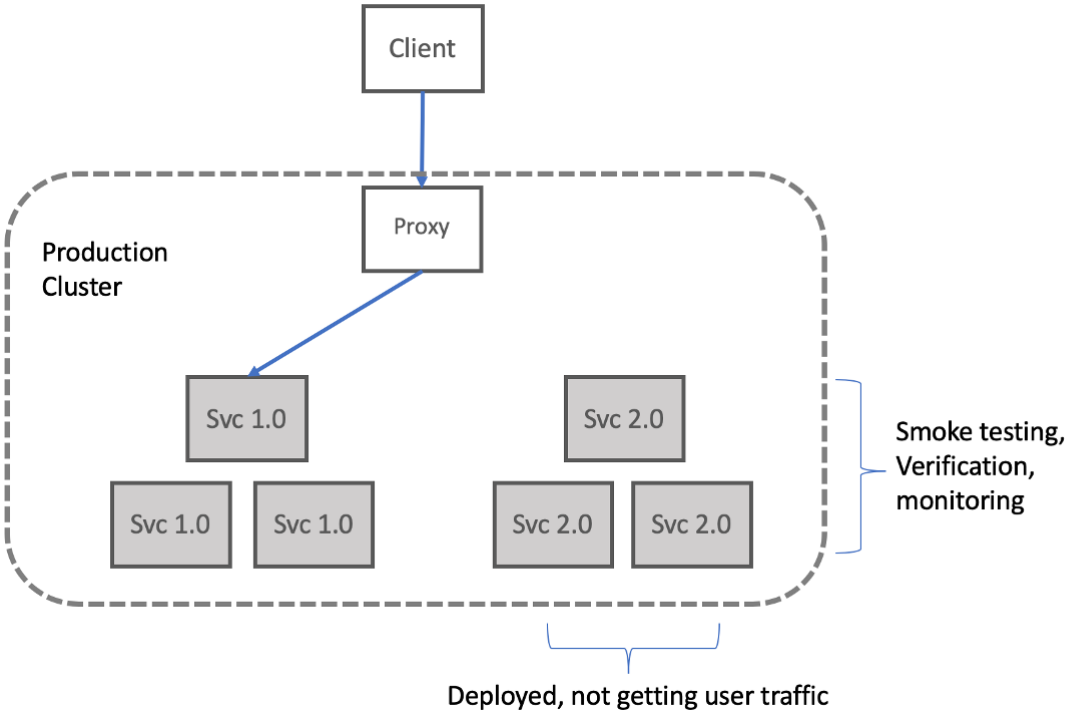


Figure 5.2 A deployment is code that is installed into production but does not take any live production traffic. While installed into production, we should smoke test and validate it

Once we have code **deployed** into production, we can make a business decision about how to **release** it to our users. **Releasing** our code means bringing live traffic over to our new deployment. This is not, however, an all-or-nothing proposition. This is where a decoupling between **deployment** and **release** becomes crucial to reduce the risk of bringing new code to production. We can decide to **release** our new software to just our internal employees. If we are an internal employee, we can control the traffic such that we'd be exposed to this new version of the software. As operators of the software we can then observe (using our logging and metrics collection) and verify our code change had the intended affect.

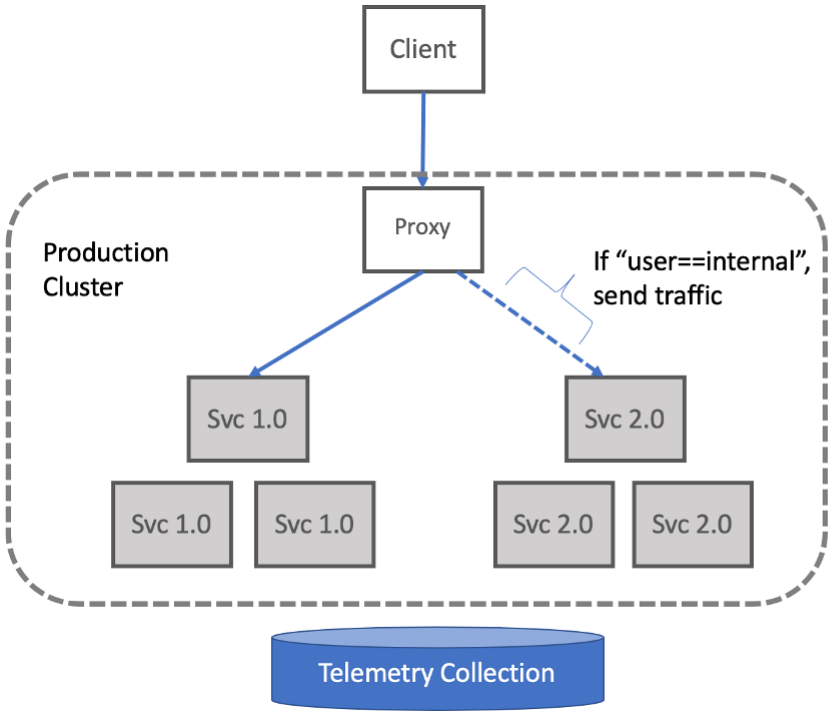


Figure 5.3 A release is when we start to bring production traffic over to the deployment ideally in an incremental way

At this point we have both the old version of our software taking the bulk of the live traffic and our newer version taking a small fraction of the traffic. This approach is known as "canarying" or a "canary release" which follows the metaphor of a "canary bird" in a coal mine. We basically choose a small group of users to expose to the new version of our code and watch how it behaves. If it has unintended behaviors, we can back out the release and redirect traffic back to the previous version of our service.

If we’re comfortable with the behavior and performance of the new code changes, we can further open up the aperture of the release; we may wish to allow our non-paying customers or silver-level (vs gold or platinum) customers to see these changes now.

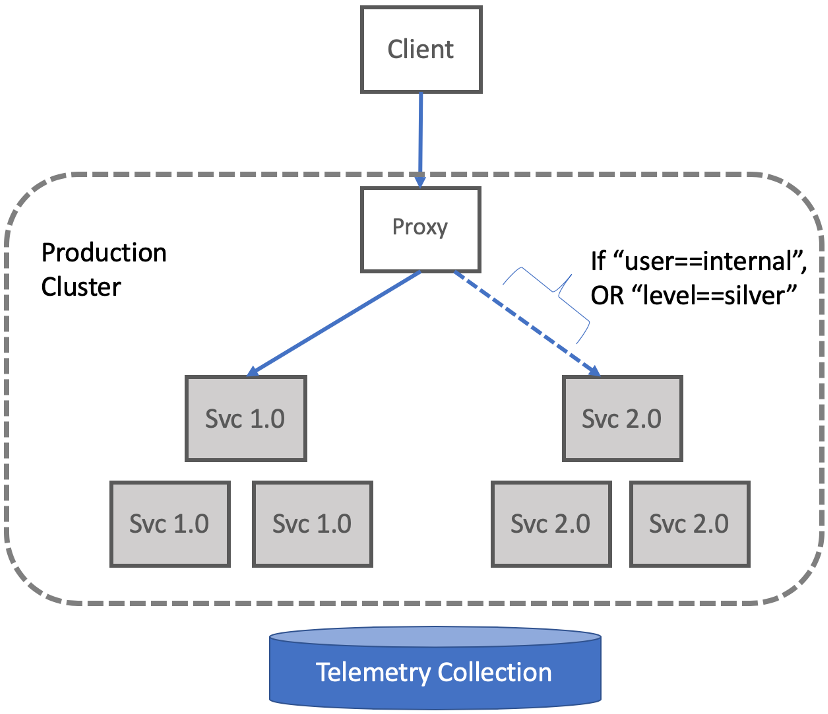


Figure 5.4 We can graduate the release to more of our user base by opening up the criteria of which users should be routed to our new deployment

We continue this iterative approach to release and observe until all of our customers are now exposed to these new code changes. At any point in this process, we may find that the new code doesn’t deliver the functionality, behavior, or performance that we expected and validated through real user interaction. We can at any point rollback this release by directing traffic back to the previous version.

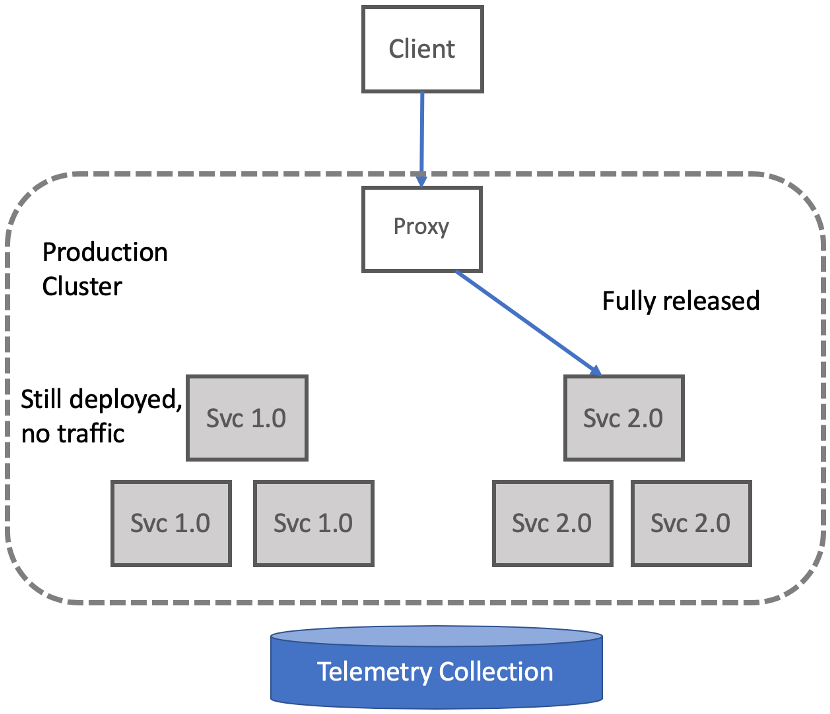


Figure 5.5 We can continue to shift traffic over to our new deployment until its fully released. A rollback would be shifting traffic back to the original deployment

In the past, ACME company would combine the two ideas of **deployment** and **release**. To bring code changes to production, they would initiate a rolling upgrade which effectively replaces old versions of a service with a new version. As soon as the new version was introduced to the cluster, it would be taking traffic. This would expose users to the new version of the code and any bugs/issues that the new code may have brought along.

Decoupling **deployment** and **release** allows us to more finely control how and which users get exposed to the new changes.This allows us to reduce the risk of bringing new code to production. Let's see how Istio can help us lower the risk of doing a release by controlling traffic based on the requests that come into the system.

5.2 Routing requests with Istio

In Chapter 2, we used Istio to control traffic to the `catalog` service. We used the `VirtualService` Istio resource to specify how to route the traffic. Let's take a closer look at how that works. We'll be controlling the route of a request based on its content (by evaluating its headers). In this way, we can make a deployment available to certain users in a technique called a *dark launch*. In a dark launch, a large percentage of users are sent to a known working version of a service, while certain classes of users are sent to a newer version. This way we can expose new functionality in a controlled way to a specific group without affecting everyone else.

5.2.1 Clean up our workspace

Before we dive in, let's clean up our environment so we can start from the same clean slate. If you're in the `istioinaction` namespace in your Kubernetes cluster, you should be able to run the following commands. If you're not, then switch to the `istioinaction` namespace like this:

```
$  kubectl create namespace istioinaction
$  kubectl config set-context $(kubectl config current-context) \
 --namespace=istioinaction
```

Now clean up any resources:

```
$  kubectl delete deployment,svc,gateway,\
virtualservice,destinationrule --all -n istioinaction
```

5.2.2 Deploy v1 of catalog service

Now, let's deploy `v1` of our `catalog` service. From the root of the book's source code, run the following command:

```
$  istioctl kube-inject -f services/catalog/kubernetes/catalog.yaml \
| kubectl apply -f -

serviceaccount/catalog created
service/catalog created
deployment.extensions/catalog created
```

Give it a few moments to start up. You can watch the progress with the following command:

```
$  kubectl get pod -w

NAME                       READY     STATUS    RESTARTS   AGE
catalog-98cfcf4cd-xnv79    2/2       Running   0          33s
```

Recall from the previous chapters that the `istioctl kube-inject` command is used to inject the Istio service proxy into the deployment. This sidecar allows the deployment to participate in the mesh. At this point, we can only reach the `catalog` service from within the cluster. Run the following command to both verify we can reach the `catalog` service and that it responds correctly:

```
$  kubectl run -i --rm --restart=Never dummy \
--image=dockerqa/curl:ubuntu-trusty --command \
-- sh -c 'curl -s catalog/items'
```

Now, let's expose the `catalog` service to clients that live outside the cluster. Recalling our knowledge from Chapter 4, we will use an Istio Gateway to do this (note the domain we're using for this is `catalog.istioinaction.io`):

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: coolstore-gateway
spec:
```

```
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "catalog.istioinaction.io"
```

Let's create this `Gateway` resource. Go in to the `chapters/chapter5` folder and run the following command:

```
$  kubectl apply -f catalog-gateway.yaml

gateway.networking.istio.io/coolstore-gateway created
```

Next, just like we saw in Chapter 4, we need to create a `VirtualService` that routes traffic to our `catalog` service. Let's create a `VirtualService` resource that looks like this:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: catalog-vs-from-gw
spec:
  hosts:
  - "catalog.istioinaction.io"
  gateways:
  - catalog-gateway
  http:
  - route:
    - destination:
        host: catalog
```

Let's create this `VirtualService`:

```
$  kubectl apply -f catalog-vs.yaml

virtualservice.networking.istio.io/catalog-vs-from-gw created
```

We should now be able to reach the catalog service from outside the cluster by calling into the Istio Gateway. We are using Docker for Desktop which publishes the Istio ingress gateway on `localhost:80` so we should be able to run the following command:

```
$  curl http://localhost/items -H "Host: catalog.istioinaction.io"
```

You should see the same output that we got when we called the service from inside the cluster. In this case, we're going through the Gateway and calling the `catalog` service from outside the cluster.

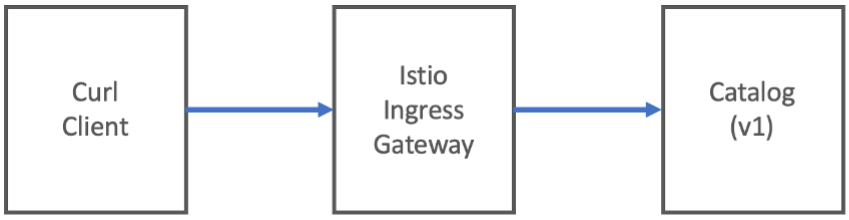


Figure 5.6 In this initial example, we're calling the catalog service directly through the Gateway

5.2.3 Deploy v2 of catalog service

To see the traffic-control features of Istio, let's deploy a `v2` of the `catalog` service. This command assumes you're at the root of the source code directory:

```
$  istioctl kube-inject \
-f services/catalog/kubernetes/catalog-deployment-v2.yaml \
| kubectl apply -f -

deployment.extensions/catalog-v2 created
```

If you list the pods in our cluster, you should see this:

```
$  kubectl get pod

NAME                          READY     STATUS    RESTARTS   AGE
catalog-98cfcf4cd-xnv79       2/2       Running   0          14m
catalog-v2-598b8cfbb5-6vw84   2/2       Running   0          36s
```

If you call the `catalog` service multiple times now, you should see some responses that have an additional field in the response. `v2` responses will have a field called `imageUrl` while `v1` responses will not:

```
$  for in in {1..10}; do curl http://localhost/items \
-H "Host: catalog.istioinaction.io"; printf "\n\n"; done

[
  {
    "id": 0,
    "color": "teal",
    "department": "Clothing",
    "name": "Small Metal Shoes",
    "price": "232.00",
    "imageUrl": "http://lorempixel.com/640/480"
  }
]
[
  {
    "id": 0,
    "color": "teal",
    "department": "Clothing",
    "name": "Small Metal Shoes",
    "price": "232.00"
  }
]
```

5.2.4 Route all traffic to v1 of catalog

Like we did in Chapter 2, let's route all traffic to `v1` of the `catalog` service. When doing a dark launch, this is usually the traffic pattern before beginning the dark launch. To do that, we need to give Istio a hint about how to identify which workloads are `v1` and which are `v2`. In our Kubernetes `Deployment` resource for `v1` of the `catalog` service, we used labels `app: catalog` and `version: v1`. For the `Deployment` that specified `v2` of our `catalog` service, we used the labels `app: catalog` and `version: v2`. For Istio, we'll create a `DestinationRule` that specifies these different versions as `subsets`:

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

Let's create this `DestinationRule`. Navigate to the `chapters/chapter5` folder and run the following:

```
$  kubectl apply -f catalog-dest-rule.yaml

destinationrule.networking.istio.io/catalog created
```

Now that we've specified to Istio how to break up the different versions of our `catalog` service, let's update our `VirtualService` to route all traffic to `v1` of our `catalog` service:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: catalog-vs-from-gw
spec:
  hosts:
  - "catalog.istioinaction.io"
  gateways:
  - catalog-gateway
  http:
  - route:
    - destination:
        host: catalog
        subset: version-v1   ❶
```

❶ specify subset

Let's replace this `VirtualService`. Again, from the `chapters/chapter5` folder:

```
$  kubectl apply -f catalog-vs-v1.yaml

virtualservice.networking.istio.io/catalog-vs-from-gw replaced
```

Now if we call our `catalog` service, we should only see `v1` responses:

```
$  for i in {1..10}; do curl http://localhost/items \
-H "Host: catalog.istioinaction.io"; printf "\n\n"; done
```

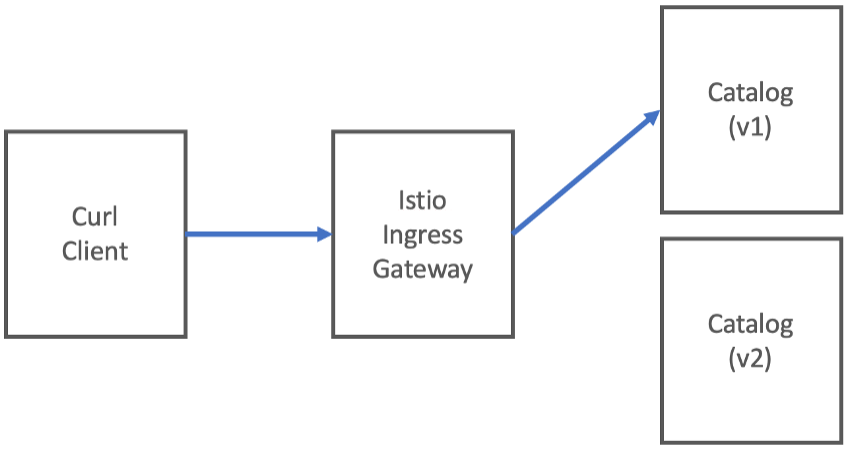


Figure 5.7 Route all traffic to v1 of catalog service

5.2.5 Route specific requests to v2

At this point, all traffic and all requests are routed to the `v1` of `catalog` service. Let's say for specific requests we want to route to `v2` of the `catalog` service. Maybe we wish to route any traffic that includes an HTTP header of `x-istio-cohort: internal` to `v2` of the `catalog` service. We can specify this request routing in the Istio `VirtualService` resource like this:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: catalog-vs-from-gw
spec:
  hosts:
  - "catalog.istioinaction.io"
  gateways:
  - catalog-gateway
  http:
  - match:
      - headers:
          x-istio-cohort:
            exact: "internal"
      route:
      - destination:
          host: catalog
          subset: version-v2
  - route:
    - destination:
        host: catalog
        subset: version-v1
```

Let's replace this `VirtualService`:

```
$  kubectl apply -f catalog-vs-v2-request.yaml

virtualservice.networking.istio.io/catalog-vs-from-gw replaced
```

At this point, if we call our service, we should still see `v1` responses. However, if we send a request with the `x-istio-cohort` header equal to `internal` we should be routed to `v2` of the `catalog` service:

```
$  curl  http://localhost/items \
-H "Host: catalog.istioinaction.io" -H "x-istio-cohort: internal"
```

Now you should see the expected response from the `v2` `catalog` service.

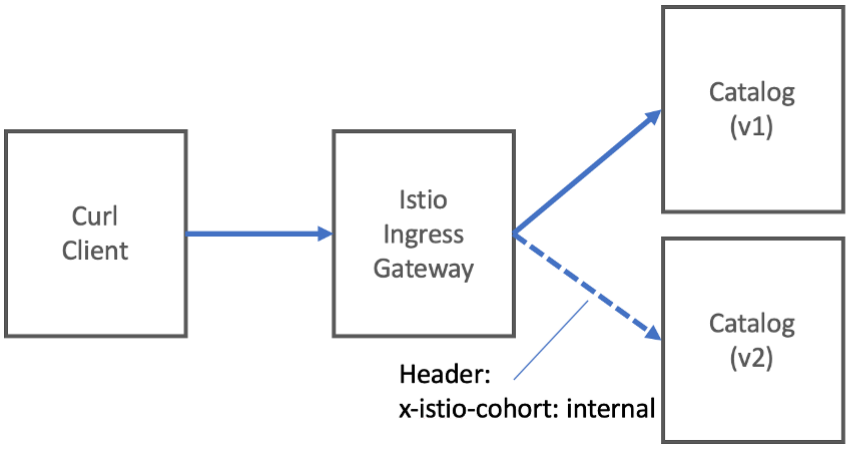


Figure 5.8 Fine-grained request routing for requests with certain content

Up until this point, we've seen how we can use Istio to do request routing, but we've been doing the routing from the edge/Gateway. These traffic rules can also be applied deep within a call graph. We actually did this in Chapter 2, so let's recreate that and verify that works as expected.

NOTE **Istio's routing is built on Envoy Proxy**

Istio's routing capabilities derive from Envoy's capabilities. For request-specific routing, teams may opt to use application injected headers (like we see here in this example of using `x-istio-cohort` or rely on known headers like `Agent` or a value from a cookie. In practice, you can also use decision engines to decide what headers to inject and subsequently make routing decisions on.

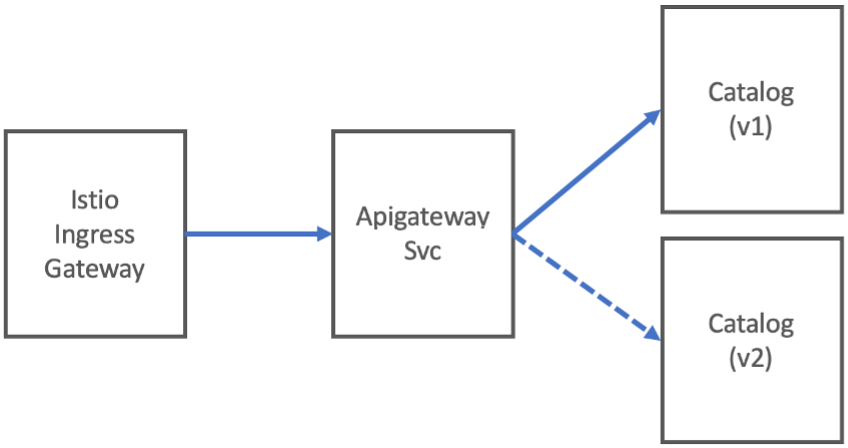


Figure 5.9 Fine-grained request routing for requests with certain content deep within a call graph

Let's remove all the Istio resources in our `istioinaction` namespace:

```
$  kubectl delete gateway,virtualservice,destinationrule --all
```

Let's restore the architecture we had in chapter 2 with the `apigateway` and `catalog` services (and an Istio Gateway directing traffic to the `apigateway` service:

```
$  istioctl kube-inject -f \
services/apigateway/kubernetes/apigateway.yaml \
| kubectl apply -f -

serviceaccount/apigateway created
service/apigateway created
deployment.extensions/apigateway created
```

Now from the `chapters/chapter5` folder, let's set up the Istio ingress gateway to route to the `apigateway` service:

```
$  kubectl apply -f apigateway-catalog-gw-vs.yaml

gateway.networking.istio.io/coolstore-gateway created
virtualservice.networking.istio.io/apigateway-virtualservice created
```

Wait until the pods come up correctly:

```
$  kubectl get pod -w

NAME                           READY     STATUS    RESTARTS   AGE
apigateway-86b9cf46d6-5vzrg    2/2       Running   0          13m
catalog-98cfcf4cd-tllnl        2/2       Running   0          13m
catalog-v2-598b8cfbb5-5m65c    2/2       Running   0          28s
```

If you issue calls again to the `apigateway` service, you should see alternating responses to the `v1` and `v2` `catalog` services like we saw earlier when accessing `catalog` service directly.

```
curl -H "Host: apiserver.istioinaction.io" http://localhost/api/catalog
```

Let's create a `VirtualService` and `DestinationRule` that routes all traffic to `v1` of the `catalog` service.

```
$  kubectl apply -f catalog-dest-rule.yaml

destinationrule.networking.istio.io/catalog created
```

```
$  kubectl apply -f catalog-vs-v1-mesh.yaml

virtualservice.networking.istio.io/catalog created
```

Now if you hit the `apigateway` service endpoint again, you should only see `v1` `catalog` service responses:

```
$  curl  http://localhost/api/catalog -H "Host: apiserver.istioinaction.io"
```

Lastly, let's add the request-based routing specifying that routing depends on whether the `x-istio-cohort` header is present and equals `internal`:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: catalog
spec:
  hosts:
  - catalog
  gateways:
    - mesh
  http:
  - match:
    - headers:
        x-istio-cohort:
          exact: "internal"
    route:
    - destination:
        host: catalog
        subset: version-v2
  - route:
    - destination:
        host: catalog
        subset: version-v1
```

Let's replace the `VirtualService`:

```
$  kubectl apply -f catalog-vs-v2-request-mesh.yaml
```

Now if you pass in the `x-istio-cohort` header, you should see the traffic routed to `v2` of the `catalog` service within the call graph:

```
$  curl  http://localhost/api/catalog -H "Host: apiserver.istioinaction.io" \
-H "x-istio-cohort: internal"
```

5.3 Traffic shifting

In this section, we'll take a look at another way to "canary" or incrementally release a deployment. In the previous section we showed routing based on header matching to achieve a "dark launch" for certain user groups. In this section we'll distribute all live traffic to a set of versions for a particular service based on weights. For example, if we've dark launched `v2` of our `catalog` service to internal employees and we'd like to slowly release this to everyone, we can specify a routing weight of 10% to `v2` in which 10% of all traffic destined for the `catalog` service will go to `v2` while 90% of the traffic will still go to `v1`. Doing this, we can further reduce the risk of doing a release by controlling how much of the total traffic would be affected by any negative impacts of the `v2` code.

Just like with the dark launch, we'll still want to monitor and observe our service for any errors and to rollback the release if there are any issues. In this case, rolling back is as simple as changing the routing weights (all the way back to 0 for `v2` if needed) so that `v2` of the `catalog` service gets a reduced percentage of total traffic. Let's take a look at using Istio to perform weighted traffic shifting.

From the previous section, we should have the following services running (including `v1` and `v2` of the `catalog` service):

```
$  kubectl get pod

NAME                            READY     STATUS    RESTARTS   AGE
apigateway-86b9cf46d6-5vzrg   2/2       Running   58         12h
catalog-98cfcf4cd-tllnl       1/2       Running   60         12h
catalog-v2-598b8cfbb5-5m65c   1/2       Running   58         11h
```

Let's reset all traffic to `v1` of the `catalog` service. From the `chapters/chapter5` folder, run the following:

```
$  kubectl apply -f catalog-vs-v1-mesh.yaml

virtualservice.networking.istio.io/catalog replaced
```

If we call our service, we should see only responses from the `v1` service as expected:

```
$  for i in {1..10}; do curl http://localhost/api/catalog \
-H "Host: apiserver.istioinaction.io"; done
```

Let's route 10% of the traffic to the `v2` of `catalog` service:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: catalog
spec:
  hosts:
  - catalog
  gateways:
```

```
    - mesh
  http:
  - route:
    - destination:
        host: catalog
        subset: version-v1
      weight: 90   ❶
    - destination:
        host: catalog
        subset: version-v2
      weight: 10   ❷
```

❶ most traffic to v1

❷ some traffic to v2

Let's update the routing for the `catalog` service:

```
$  kubectl apply -f catalog-vs-v2-10-90-mesh.yaml

virtualservice.networking.istio.io/catalog replaced
```

Now if we called our service, we should see approximately 1 out of 10 calls have the `v2` response:

```
$  for i in {1..100}; do curl -s http://localhost/api/catalog -H "Host: apiserver.istioinaction.io"  \
| grep -i imageUrl; done | wc -l
```

In this command, we call the `/api/catalog` endpoint 100 times. You should see at least 10 as the result (10% of 100 items) printed to the screen when the command returns.

If we wanted to get a complete 50/50 split between traffic, we just need to update the weights on the routing:

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

```
$  kubectl apply -f catalog-vs-v2-50-50-mesh.yaml

virtualservice.networking.istio.io/catalog replaced
```

Try calling our service again:

```
$  for i in {1..100}; do curl -s http://localhost/api/catalog -H "Host: apiserver.istioinaction.io"  \
| grep -i imageUrl; done | wc -l
```

You should see approximately `50` returned from that command which means half the calls returned with a response from `v2` of our backend `catalog` service.

You can shift the traffic between 1-100 for each of the versions of your service but the sum of all of the weights must equal 100. If it does not equal 100, unpredictable traffic weights can occur. Also note, if you have other versions that just `v1` and `v2` like we do in this example, they must be declared as `subsets` in the `DestinationRule`. See `chapters/chapter5/catalog-dest-rule.yaml` for an example.

For the steps in this chapter, we've manually shifted the traffic between different versions. Ideally, you would automate this traffic shifting behind some tooling or a deployment pipeline in your CI/CD pipeline.

WARNING **Monitoring is crucial**

As you slowly release your new version of software, you'll want to monitor and observe both new and old versions to verify things like stability, performance, and correctness. If you spot any signs of impact, you can easily rollback to the older version of the service by shifting the weights back. You'll also want to keep in mind your services will need to be built to support multiple versions running concurrently when doing this. The more stateful a service is (even if it depends on external state) the more difficult this can be. Please take a look at these blogs for some more thoughts on this topic: bit.ly/2NSE2gf and bit.ly/2oJ86jc

5.4 Lowering risk even further: Traffic mirroring

Using the previous two techniques of request-level routing and traffic shifting, we can lower our risk of doing releases. Both techniques use live traffic/requests and can impact users even though you can control how widespread a potentially negative effect can be. Another approach is to "mirror" production traffic to a new deployment which makes a copy of the production traffic and sends it to the new deployment out of band of any customer traffic. Using the mirroring approach, we can direct real production traffic to our deployment and get real feedback about how our new code will behave without actually impacting users. Istio supports mirroring traffic which can bring down the risk of doing a deployment and release even more than the previous two approaches. Let's take a look.

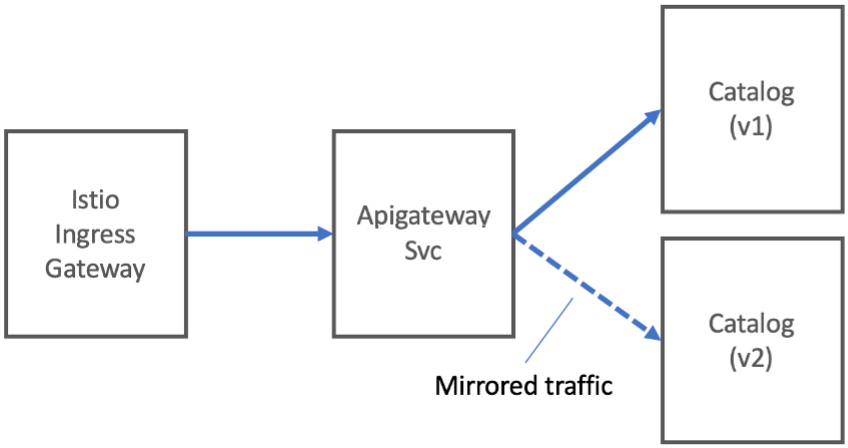


Figure 5.10 Traffic mirrored to catalog v2 service out of band from request path

To mirror our traffic to `v2` of our `catalog` service, let's first reset all traffic to `v1`. From the `chapters/chapter5` folder, run the following:

```
$  kubectl apply -f catalog-vs-v1-mesh.yaml
```

Now, let's look at the `VirtualService` that we'll need to do the mirroring:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: catalog
spec:
  hosts:
  - catalog
  gateways:
    - mesh
  http:
  - route:
    - destination:
        host: catalog
        subset: version-v1
      weight: 100
    mirror:    ❶
      host: catalog
      subset: version-v2    ❷
```

❶ Mirroring clause

❷ Subset of catalog service

With the above `VirtualService` definition, we're routing 100% of live traffic to `v1` of the `catalog` service, but we'll also mirror the traffic to `v2`. As mentioned above, mirroring is done in a fire and forget manner in which a copy of the request is create and sent to the mirrored cluster (in this case `v2` of `catalog`). This mirrored request cannot affect the real request because the Istio proxy that does the mirroring will ignore any responses (success/failure) from the mirrored cluster. Let's create this `VirtualService` resource:

```
$   kubectl apply -f catalog-vs-v2-mirror.yaml

virtualservice.networking.istio.io/catalog replaced
```

Now if we send traffic to our service, we should see the response from only the `v1` `catalog` service:

```
$  curl http://localhost/api/catalog -H "Host: apiserver.istioinaction.io"
```

We can see the logs of the `v1` service to verify we're getting traffic:

```
$  CATALOG_V1=$(kubectl get pod -l app=catalog -l version=v1 \
-o jsonpath={.items..metadata.name})
$  kubectl logs $CATALOG_V1 -c catalog
```

We should see log entries like this:

Listing 5.1 Output from the logs from catalog-v1

```
GET /items 200 2.848 ms - -
request path: /items
blowups: {}
number of blowups: 0
GET /items 200 2.846 ms - -
request path: /items
blowups: {}
number of blowups: 0
GET /items 200 3.109 ms - -
```

Now if we get the log of the `catalog` `v2` service, we should also see logging entries:

Listing 5.2 Output from the logs from catalog-v2

```
$  CATALOG_V2=$(kubectl get pod -l app=catalog -l version=v2 \
-o jsonpath={.items..metadata.name})
$  kubectl logs $CATALOG_V2 -c catalog

request path: /items
blowups: {}
number of blowups: 0
GET 10.12.1.178-shadow:80 /items 200 - - 11.045 ms
GET /items 200 11.045 ms - -
```

As we can see, for each request we send in to our service, a request ends up going to `v1` of `catalog` as well as `v2`. The request that makes it to the `v1` service is the live request and that's what response we'll end up seeing. The request that makes it to `v2` is mirrored and is sent as fire-and-forget.

Another thing to note from listing XX is that when the mirrored traffic makes it to `catalog` `v2` service, the `Host` header has been modified to indicate it is mirrored/shadowed traffic. Instead of `Host: 10.12.1.178:8080` for the `Host` header, we see `Host: 10.12.1.178-shadow:8080`. A service that receives a request with the `-shadow` postfix can identify that request as a mirrored request and take that into consideration when processing it (for example, that the response will be discarded, so either roll back a transaction or don't make any calls that are resource intensive).

Mirroring traffic is one part of the story to lower the risk of doing releases, but just like with

request routing and traffic shifting, our applications should be aware of this context and either be able to run in both live/mirrored mode, or able to run as multiple versions, or both. Please see these blog posts: bit.ly/2NSE2gf and bit.ly/2oJ86jc

5.5 Routing to services outside your cluster by using Istio's service discovery

By default, Istio allows any traffic out of the service mesh. For example, if an application tries to talk with external websites or services not managed by the service mesh, Istio will allow this traffic out. Since all traffic first passes through the service-mesh sidecar proxy (Istio proxy), and we can control routing of the traffic, we can change Istio's default policy and deny all traffic trying to leave the mesh.

Blocking all traffic leaving the mesh is a basic defense-in-depth posture to prevent bad actors from "phoning home" if a service or application within the mesh becomes compromised. Blocking external traffic using Istio is not sufficient however. A compromised pod could just bypass the proxy. Therefore you need a defense-in-depth approach with additional traffic blocking mechanisms such as firewalls.

For example, if a vulnerability allows an attacker to take control of a particular service, they can try to inject code or otherwise manipulate the service to reach out to servers that they control. If they're able to do this and further control the compromised service, they can begin to exfiltrate company-sensitive data and intellectual property.

Let's configure Istio to block external traffic leaving the mesh providing a simple layer of protection.

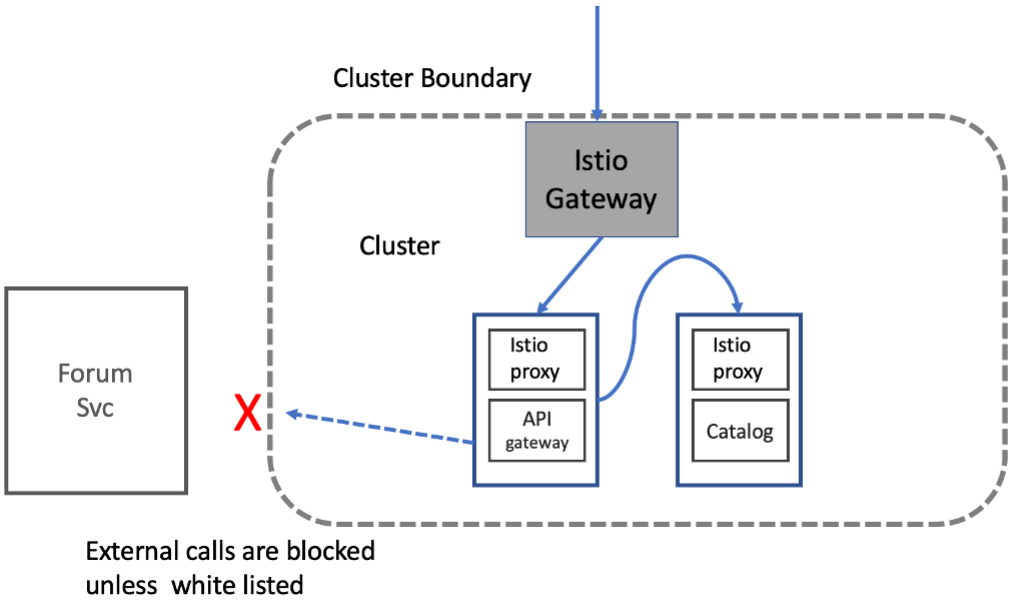


Figure 5.11 Let's block any traffic trying to leave the service by default

Run the following command to change Istio's default from "ALLOW_ANY" to "REGISTRY_ONLY". This means, we'll only allow traffic to leave the mesh if it's explicitly whitelisted in the service-mesh registry.

Listing 5.3 Reinstall Istio with correct mode for outbound traffic

```
$  istioctl install --set profile=demo \
--set meshConfig.outboundTrafficPolicy.mode=REGISTRY_ONLY
```

NOTE **Alternative way to update the outboundTrafficPolicy**

In the previous step, we updated the installation of Istio to set the `outboundTrafficPolicy` setting to `REGISTRY_ONLY`. For this book, and for experimentation purposes, that is fine. However, in a real deployment you'd likely do this with the IstioOperator or update the `istio` configmap in `istio-system` directly.

Since not all services will live in the service mesh, we need a way for services inside the mesh to communicate with those outside of the mesh. Those could be existing HTTP services or more likely infrastructure services like databases or caches that live outside of the service mesh. We can still implement sophisticated routing for services that reside outside of Istio, but first we'll have to introduce you to the concept of the `ServiceEntry`.

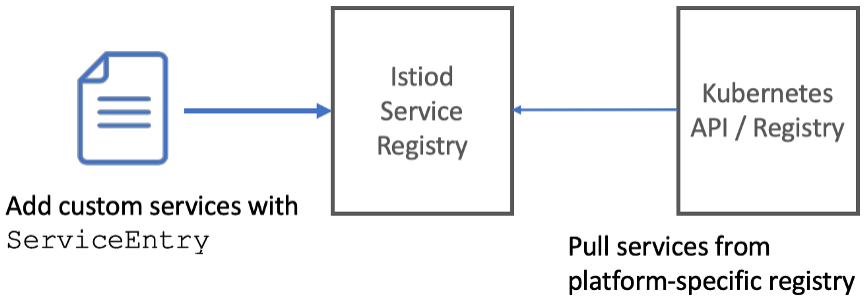


Figure 5.12 We can specify service entry resources which augment and insert external services into the Istio service registry

Istio builds up an internal service registry of all the services that are known by the mesh and can be accessed within the mesh. You can think of this registry as the canonical representation of a service-discovery registry that services within the mesh can use to find other services. Istio builds up this internal registry by making some assumptions about the platform on which the control plane is deployed. For example, in this book we're deploying the control plane onto Kubernetes. Istio will use the default Kubernetes API to build its catalog of services (based on Kubernetes Service objects, see https://kubernetes.io/docs/concepts/services-networking/service/ for more). For our services within the mesh to communicate with those outside of the mesh, we need to let Istio's service-discovery registry know about this external service.

In our fictitious conference outfitter store, we want to provide the best possible customer service and allow customers to give feedback or share thoughts directly with each other. To do that, we'll connect our users with an online forum that is built and deployed outside of our service-mesh cluster. In this case, our forum lives at the `jsonplaceholder.typicode.com` url.

The Istio `ServiceEntry` encapsulates registry metadata that we can use to insert an entry into Istio's service registry. Let's take a look at the following example:

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: jsonplaceholder
spec:
  hosts:
  - jsonplaceholder.typicode.com
  ports:
  - number: 80
    name: http
    protocol: HTTP
  resolution: DNS
  location: MESH_EXTERNAL
```

This `ServiceEntry` resource inserts an entry into Istio's service registry which makes explicit that clients within the mesh are allowed to call a host `jsonplaceholder.typicode.com`. The `jsonplaceholder.typicode.com` service exposes a sample REST API that we can use to simulate talking with services that live outside of our cluster. Before we create this service entry, let's install a service that talks to the `jsonplaceholder.typicode.com` REST API and observe that Istio indeed blocks any traffic going outbound.

Let's install a sample `forum` application that leverages `jsonplaceholder.typicode.com`. From the root of the source code, run:

```
$ istioctl kube-inject -f \
services/forum/kubernetes/forum-all.yaml \
| kubectl apply -f -
```

Give that a few moments to come up at which point you should see output similar to this:

```
$  kubectl get pod -w

NAME                           READY     STATUS    RESTARTS   AGE
apigateway-86b9cf46d6-5vzrg    2/2       Running   59         19h
catalog-98cfcf4cd-qlxrl        2/2       Running   0          6h
forum-7d7889986d-ndgbk         2/2       Running   0          10m
sleep-5f4c9b8b6b-fhntw         2/2       Running   0          3h
```

Let's try calling our new `forum` service from within the mesh:

```
$  kubectl run -i --rm --restart=Never dummy \
--image=dockerqa/curl:ubuntu-trusty --command \
-- sh -c 'curl -s forum/api/users'

error calling Users service
```

To allow this call to go through, let's create an Istio ServiceEntry resource to the `jsonplaceholder.typicode.com` host. This will insert an entry into Istio's service registry and make it known to the service mesh. From the `chapters/chapter5` folder, run:

```
$  kubectl apply -f forum-serviceentry.yaml

serviceentry.networking.istio.io/jsonplaceholder created
```

Now let's try calling the `forum` service again:

```
$  kubectl run -i --rm --restart=Never dummy \
--image=dockerqa/curl:ubuntu-trusty --command \
-- sh -c 'curl -s forum/api/users'

...

  {
    "id": 10,
    "name": "Clementina DuBuque",
    "username": "Moriah.Stanton",
    "email": "Rey.Padberg@karina.biz",
    "address": {
      "street": "Kattie Turnpike",
      "suite": "Suite 198",
      "city": "Lebsackbury",
      "zipcode": "31428-2261",
      "geo": {
        "lat": "-38.2386",
        "lng": "57.2232"
      }
    },
    "phone": "024-648-3804",
    "website": "ambrose.net",
    "company": {
      "name": "Hoeger LLC",
      "catchPhrase": "Centralized empowering task-force",
      "bs": "target end-to-end models"
    }
  }
```

We should see the call go through and return a list of `users`.

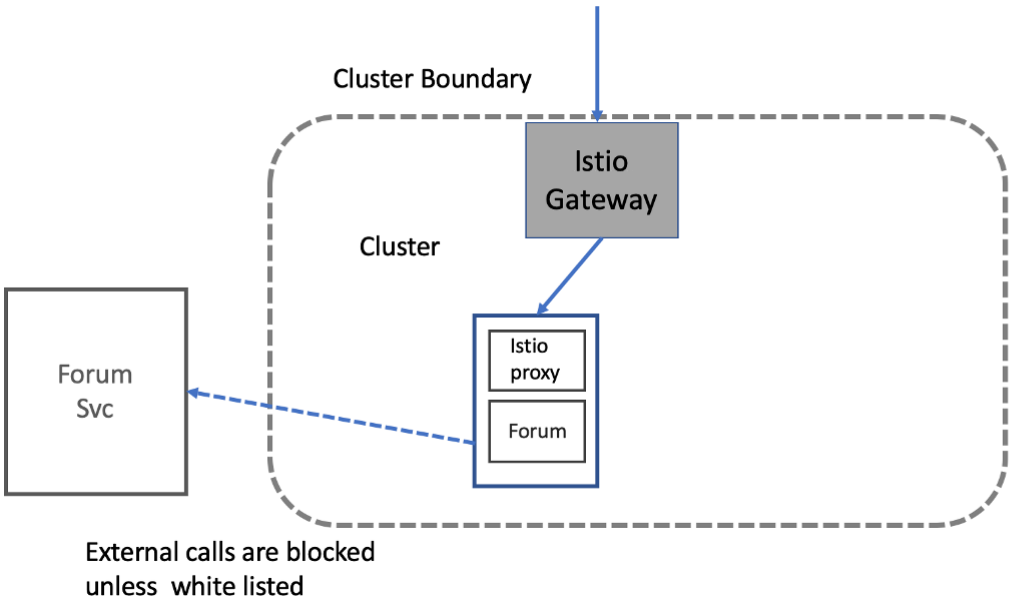


Figure 5.13 We should be able to call external services from within the service mesh once we've explicitly added the ServiceEntry

In this chapter we explored how we can reduce the risk of deploying new code by using traffic mirroring, traffic shifting, and traffic routing to slowly introduce changes to our users. In the next chapter, we'll take a look at making application interactions more resilient by implementing timeouts, retries, and circuit breakers.

5.6 Summary

Come back to this…

6 Resilience: solving application-networking challenges

This chapter covers:

- The importance of resiliency
- Client-side load balancing
- Retries / Budgets / Timeouts
- Circuit breaking and bulkheads
- Advice for migration from application libraries used for resilience

Once we have traffic coming into our cluster through the Istio ingress gateway (covered in Chapter 4) we can manipulate the traffic at the request level and control exactly what versions or "subsets" of a service to which we want certain requests to go. In the previous chapter, we covered this traffic control for weighted routing, request-match based routing, and certain types of release patterns that can be enabled with that. We can also use this traffic control to route around problems in the event of application errors, network partitions, and other major issues.

The problem with distributed systems is that they often fail in unpredictable ways and we will not be able to manually take traffic-shifting actions. What we need is a way to build sensible behaviors **into** the application so they can respond on their own when they encounter problems. We can do that with Istio including adding timeouts, retries and circuit breaking, without having to alter application code. In this chapter we'll take a look at how to do this and the implications on the rest of the system.

6.1 Building resilience into the application

Microservices must be built with resiliency as a first-class concern. The world of "just build it so it won't fail" is not real, and when it strikes, we risk taking down all of our services. When we build distributed systems with services communicating over the network, we risk creating even more failure points and possibilities of catastrophic failures. Service owners should adopt a few resiliency patterns consistently across their applications and services.

Figure 6.1 Service A calling Service B can experience network issues

For example, if service A calls service B and experiences latency in requests sent to particular endpoints of service B, we want it to proactively figure this out and route to other endpoints, other availability zones, or even other regions. If service B experiences intermittent errors, we may want to retry a failed request. Similarly if we experience issues calling service B endpoints, we may wish to back off calling service B until it can recover from whatever issues it may be experiencing. If we keep putting load on service B (and in some cases, amplifying the load as we retry the request), we risk overloading service B, which in turn could overload service A and anyone else that depends on these services and cause significant, if not cascading, errors.

The solution here is to build our applications to **expect** failures and have a way for them to automatically attempt remediation or fallback to alternative paths when servicing a request. For example, when service A calls service B and it starts to experience issues, we could retry a request, we could timeout our request, or we could cancel any further outgoing request using a circuit breaking pattern. In this chapter, we'll explore how Istio can be used to solve these problems outside of application code and in the service mesh, so that applications have a correct and consistent implementation for these resilience concerns regardless of what programming language the application is written.

6.1.1 Building resilience into application libraries

Before service-mesh technology was widely available, as service developers we had to write a lot of these basic resiliency patterns into our application code. Some frameworks emerged in the open source community that helped solve these problems. Twitter opensourced their resilience framework Finagle in 2011. Twitter Finagle is a Scala/Java/JVM application library that can be used to implement various RPC resilience patterns such as timeouts, retries, and circuit breaking. Shortly afterward, Netflix opensourced components of their resilience framework including Netflix Hystrix and Netflix Riboon which provided circuit-breaking and client-side loadbalancing respectively. Both of these libraries were very popular in the Java community including the Spring Framework adopting the Netflix OSS stack in their Spring Cloud framework.

The problem with these frameworks is that across different permutations of languages, frameworks, and infrastructure you will have varying implementations. Twitter Finagle and Netflix OSS were great for Java developers, but NodeJs, Go, and Python developers had to find or implement their own variants of these patterns. In some cases these libraries were also quite invasive to the application code so you would have networking code sprinkled around and obscuring the actual business logic. Lastly, maintaining these libraries across multiple languages and frameworks strains operational aspects of running microservices: you have to try and patch and maintain functionality parity with all of the combinations at the same time.

Istio can handle implementing these basic resiliency patterns consistently for all languages and frameworks. This does not obviate the need for the need for certain specialized application libraries or for the application to pay close attention to the details of both configuring a service mesh like Istio for resilience as well as what it takes to write a safe and correct application. For example, as we'll see in the next chapter, helping to automatically propagate specific types of request/application context can be done with application libraries. When things fail and the service mesh tries its best to recover with retries or circuit breakers, application specific code can decide how best to fall back to alternative implementations and responses to a client. Istio should be used to implement the bulk of the common resilience patterns to get consistent and understandable behavior, but then applications are free to drop into more refined nuance where necessary.

6.1.2 Using Istio to solve these problems

As we've seen in previous chapters, Istio's service proxy sits next to the application and handles all network traffic to and from the application. With Istio, since the service proxy understands application-level requests and messages (e.g., HTTP request), we can implement resilience features within the proxy. For example, we can configure Istio to **retry** a failed request up to 3 times when we experience an HTTP 503 on a service call. We can configure exactly what failures to retry on, the number of retries we would want, as well as the retry timeouts. Since the service proxy is deployed per service instance, we can have very fine grained retry behavior that's customized to fit the specific needs of the service. The same is true for all of Istio's resilience settings. Istio's service proxy implements these basic resiliency patterns out of the box:

- Client-side load balancing
- Locality aware load balancing
- Timeouts / Retries
- Circuit breaking

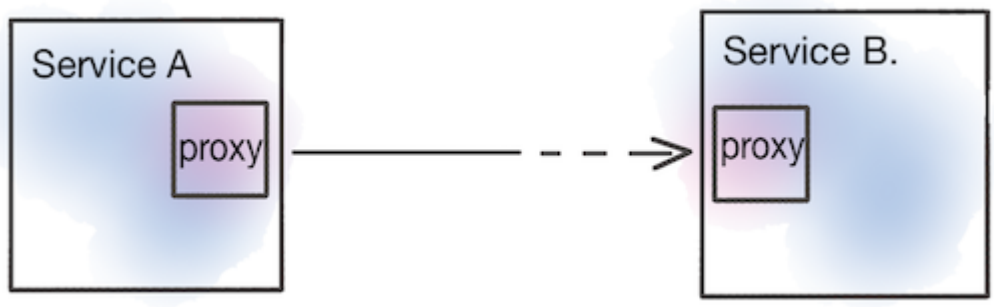


Figure 6.2 Service A calling Service B can experience network issues

6.1.3 Decentralized implementation of resilience

In cloud-native applications, we should avoid centralization of request-path processing for service-to-service communication for a few reasons. Using Istio we see our data-plane proxy through which our requests traverse are co-located with the application and there is no request-path centralization. We get the same architecture if we leverage application libraries which co-locate the handling of these resilience patterns into the code. In previous incantations of solving for some of these cross-cutting distributed systems, we've placed expensive, difficult to change, centralized hardware appliances and other software middleware into the path of the request (ie, hardware load balancers, messaging systems, Enterprise Service Bus, etc.). These previous implementations, which were built for more static environments, do not scale or respond well to highly dynamic, elastic cloud architectures and infrastructure. When solving for some of these resilience patterns, we should opt for distributed implementations.

In the following sections, we'll explore each of the resilience patterns that Istio can help with.

We will use a different set of sample applications for this section to get more fine-grained control over how the services behave. In fact, we'll use a project called fake-service by Nic Jackson who created this project to illustrate how services may behave in more realistic production environments. In these following samples, we'll see a service `simple-web` call a set of backends `simple-backend`

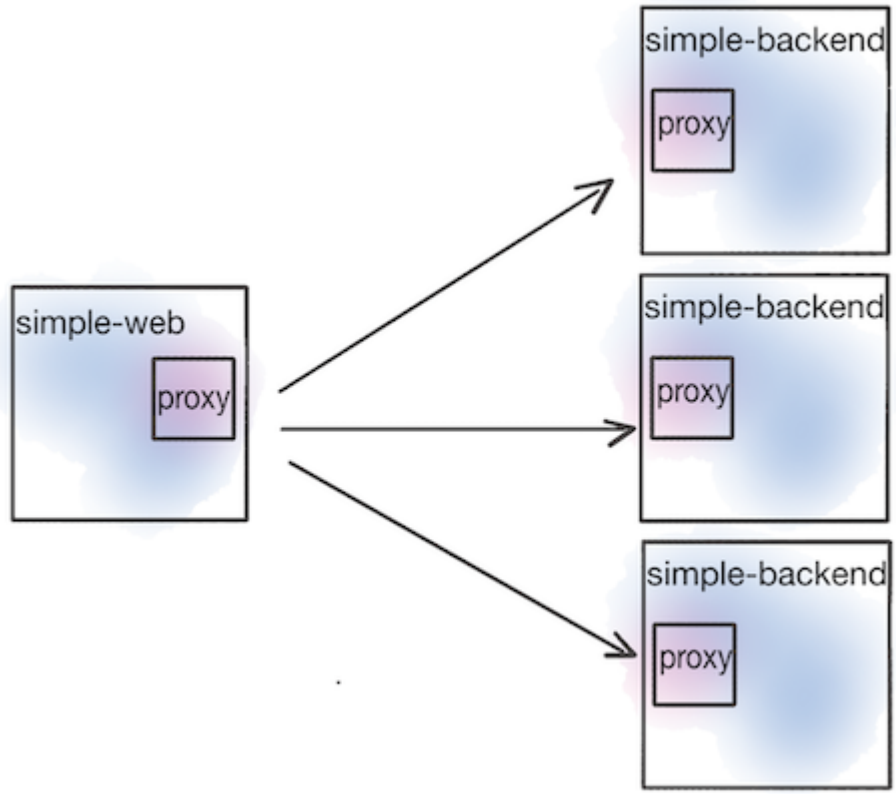


Figure 6.3 Sample services: web calls backend

6.2 Client-side load balancing

Client side load balancing is the practice of informing the client about the various endpoints available for a service and letting the client pick specific load-balancing algorithms for the best distribution of requests over the endpoints. This reduces the need to rely on centralized load-balancing, which could create bottlenecks and failure points, and allows the client to make direct, deliberate requests to specific endpoints without having to take unnecessary extra hops. This allows our clients and services to scale better and deal with a changing topology.

Istio leverages service and endpoint discovery to equip the **client side** proxy of service-to-service communication with the correct and most up-to-date information. Developers and operators of the services can then configure this client-side load balancing behavior through Istio configuration.

Service operators and developers can configure what load-balancing algorithm a client uses by defining a `DestinationRule`. Istio's service proxy is based on Envoy and supports Envoy's

load-balancing algorithms which include:

- Round robin (default)
- Random
- Weighted least request

Let's take a look at a quick example.

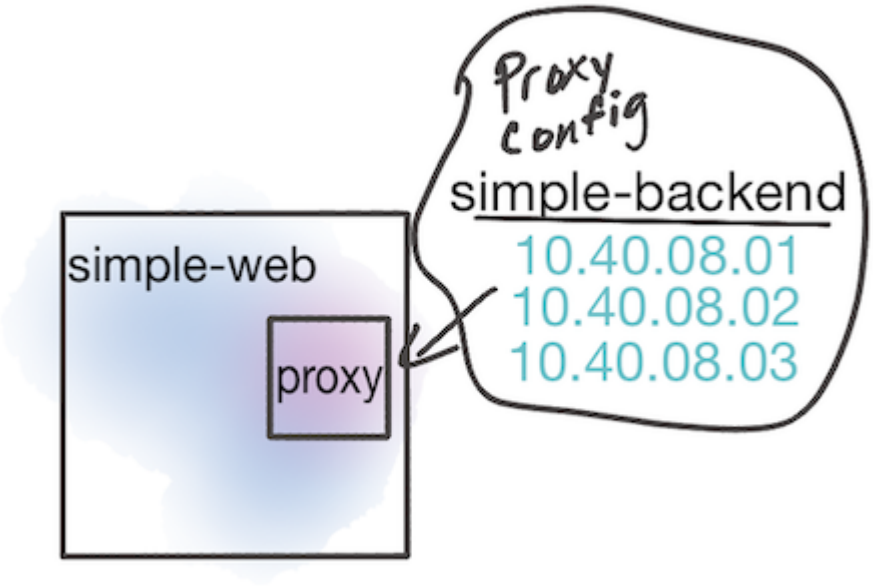


Figure 6.4 Simple web proxy knows about simple-backend endpoints

6.2.1 Getting started with client-side load balancing

Before we begin, let's cleanup our `istioinaction` namespace by deleting any resources from previous chapters. Let's make sure we're in the right namespace and then delete the appropriate resources:

```
$  kubectl config set-context $(kubectl config current-context) \
 --namespace=istioinaction
$  kubectl delete virtualservice,deployment,service,destinationrule,gateway --all
```

Let's navigate to the `chapters/chapter6` folder in the source code for the book. We will deploy two sample services with the appropriate Istio VirtualService and Gateway resources so we can call the service. For more information on Gateway and VirtualService for ingress routing, see Chapter 4.

```
$  kubectl apply -f simple-backend.yaml
$  kubectl apply -f simple-web.yaml
$  kubectl apply -f simple-web-gateway.yaml
```

Give a few moments for the pods to come up in the `istioinaction` namespace. When they're running you should see something similar to this:

```
$  kubectl get pod
NAME                                READY   STATUS    RESTARTS   AGE
simple-backend-1-54856d64fc-59dz2   2/2     Running   0          29h
simple-backend-2-64f898c7fc-bt4x4   2/2     Running   0          29h
simple-backend-2-64f898c7fc-kx88m   2/2     Running   0          29h
simple-web-56d955b6f5-7nflr         2/2     Running   0          29h
```

Let's specify the load balancing for any client calling the `simple-backend` service to be `ROUND_ROBIN` with an Istio DestinationRule. A DestinationRule specifies policies for clients in the mesh calling the specific destination. Our starting DestinationRule for `simple-backend` will look like the following:

```
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: simple-backend-dr
spec:
  host: simple-backend.istioinaction.svc.cluster.local
  trafficPolicy:
    loadBalancer:
      simple: ROUND_ROBIN
```

Let's apply this DestinationRule:

```
$  kubectl apply -f simple-backend-dr-rr.yaml
destinationrule.networking.istio.io/simple-backend-dr configured
```

We have `simple-web` which calls `simple-backend` but there are multiple replicas of the `simple-backend` service. This is intentional, as we'll modify some of the endpoints at runtime.

If all is successful, you should be able to call our sample service. We've been using Docker Desktop in our examples thus far, and for Docker Desktop it would look something similar to the following:

```
$  curl -s -H "Host: simple-web.istioinaction.io" http://localhost/

{
  "name": "simple-web",
  "uri": "/",
  "type": "HTTP",
  "ip_addresses": [
    "10.1.0.45"
  ],
  "start_time": "2020-09-15T20:39:29.270499",
  "end_time": "2020-09-15T20:39:29.434684",
  "duration": "164.184432ms",
  "body": "Hello from simple-web!!!",
  "upstream_calls": [
    {
      "name": "simple-backend",
      "uri": "http://simple-backend:80/",
      "type": "HTTP",
      "ip_addresses": [
        "10.1.0.64"
      ],
      "start_time": "2020-09-15T20:39:29.282673",
      "end_time": "2020-09-15T20:39:29.433141",
      "duration": "150.468571ms",
      "headers": {
```

```
        "Content-Length": "280",
        "Content-Type": "text/plain; charset=utf-8",
        "Date": "Tue, 15 Sep 2020 20:39:29 GMT",
        "Server": "envoy",
        "X-Envoy-Upstream-Service-Time": "155"
      },
      "body": "Hello from simple-backend-1",
      "code": 200
    }
  ],
  "code": 200
}
```

In this set of sample services, we can see we get a JSON response that shows a chain of calls. The `simple-web` service calls the `simple-backend` service and we ultimately see the response message `Hello from simple-backend-1`. If we repeat this call a few more times, we see we get responses from `simple-backend-1` and `simple-backend-2`.

```
$  for in in {1..10}; do \
curl -s -H "Host: simple-web.istioinaction.io" localhost \
| jq .upstream_calls[0].body; printf "\n"; done

"Hello from simple-backend-1"
"Hello from simple-backend-1"
"Hello from simple-backend-2"
"Hello from simple-backend-2"
"Hello from simple-backend-2"
"Hello from simple-backend-1"
"Hello from simple-backend-2"
"Hello from simple-backend-1"
"Hello from simple-backend-1"
"Hello from simple-backend-2"
```

One of the first things we can see is that the calls between `simple-web` and `simple-backend` are effectively load balanced to the different `simple-backend` endpoints. We are seeing client-side load balancing here between the `simple-web` and `simple-backend` services because the service proxy deployed with `simple-web` knows about all of the `simple-backend` endpoints and is using the default algorithm to determine which endpoints get requests. We configured our `DestinationRule` to use `ROUND_ROBIN` load balancing, but by default Istio service proxy uses a `ROUND_ROBIN` load-balancing strategy anyway. How can client-side load balancing contribute to a service's resilience?

Let's take a look at a somewhat realistic scenario using a load generator and changing the latency of the `simple-backend` service. Then we can use Istio's load-balancing strategies to help pick an appropriate configuration.

6.2.2 Setting up our scenario

In a realistic setting, services take time to process requests and the amount of time can vary for a few different reasons:

- request size
- processing complexity

- database usage
- calling other services that take time

There are also reasons outside the service itself that may contribute to varied response times:

- unexpected, stop-the-world garbage collections
- resource contention (CPU, network, etc)
- network congestion

To mimic this for our sample service, we're going to introduce delays and variance into our response times. Let's call our service again and observe the differences in overall service response times:

```
$  time curl -s -o /dev/null -H "Host: simple-web.istioinaction.io" localhost
real    0m0.261s
user    0m0.005s
sys     0m0.010s

$  time curl -s -o /dev/null -H "Host: simple-web.istioinaction.io" localhost
real    0m0.855s
user    0m0.006s
sys     0m0.009s

$  time curl -s -o /dev/null -H "Host: simple-web.istioinaction.io" localhost
real    0m0.406s
user    0m0.005s
sys     0m0.006s
```

As you can see, each time we call the service we see different response times. Load balancing can be an effective strategy to reduce the effect of endpoints experiencing periodic or unexpected latency spikes. We are going to use a tool called Fortio to exercise our services and observe differences in client-side load balancing.

Download Fortio for your platform from Fortio Releases page.

NOTE **Getting Fortio for your platform**

If you cannot find a distribution of Fortio for your platform, you can follow the instructions on https://github.com/fortio/fortio#installation to install. If that doesn't work, you can still use Fortio to follow these next steps by running Fortio within Kubernetes itself. You may not have the same experience as outlined here, but would still work the same. For example, you can call Fortio by running it within Kubernetes with the following command:

```
kubectl -n default run fortio --image=fortio/fortio:1.6.8 \
--restart='Never' -- load -H "Host: simple-web.istioinaction.io" \
-jitter -t 60s -c 10 -qps 1000 \
http://istio-ingressgateway.istio-system/
```

Let's make sure Fortio can call our service:

```
$  fortio curl -H "Host: simple-web.istioinaction.io"  http://localhost/
```

You should see a similar response to what we saw when we called with `curl` directly.

6.2.3 Testing various client-side load balancing strategies

Now that we have our Fortio load-testing client ready to go, let's explore the use case. We will use Fortio to send 1000 requests per second through 10 connections for 60 seconds. Fortio will track the latency numbers for each call and plot them on a histogram with latency percentile break down. Before our test, we'll introduce a version of the `simple-backend-1` service that increases latency for to 1 second. This will simulate one of the endpoints experiencing a long garbage collection event or other application latency. We will vary our load balancing strategy between round robin, random, and least connection and observe the differences

Let's deploy the delayed `simple-backend-1` service:

```
$  kubectl apply -f simple-backend-delayed.yaml
```

If you run Fortio in `server` mode, we can get to a web dashboard where we can input the parameters of our test, execute the test, and visualize the results.

```
$  fortio server
```

Now open your browser to the Fortio UI: http://localhost:8080/fortio

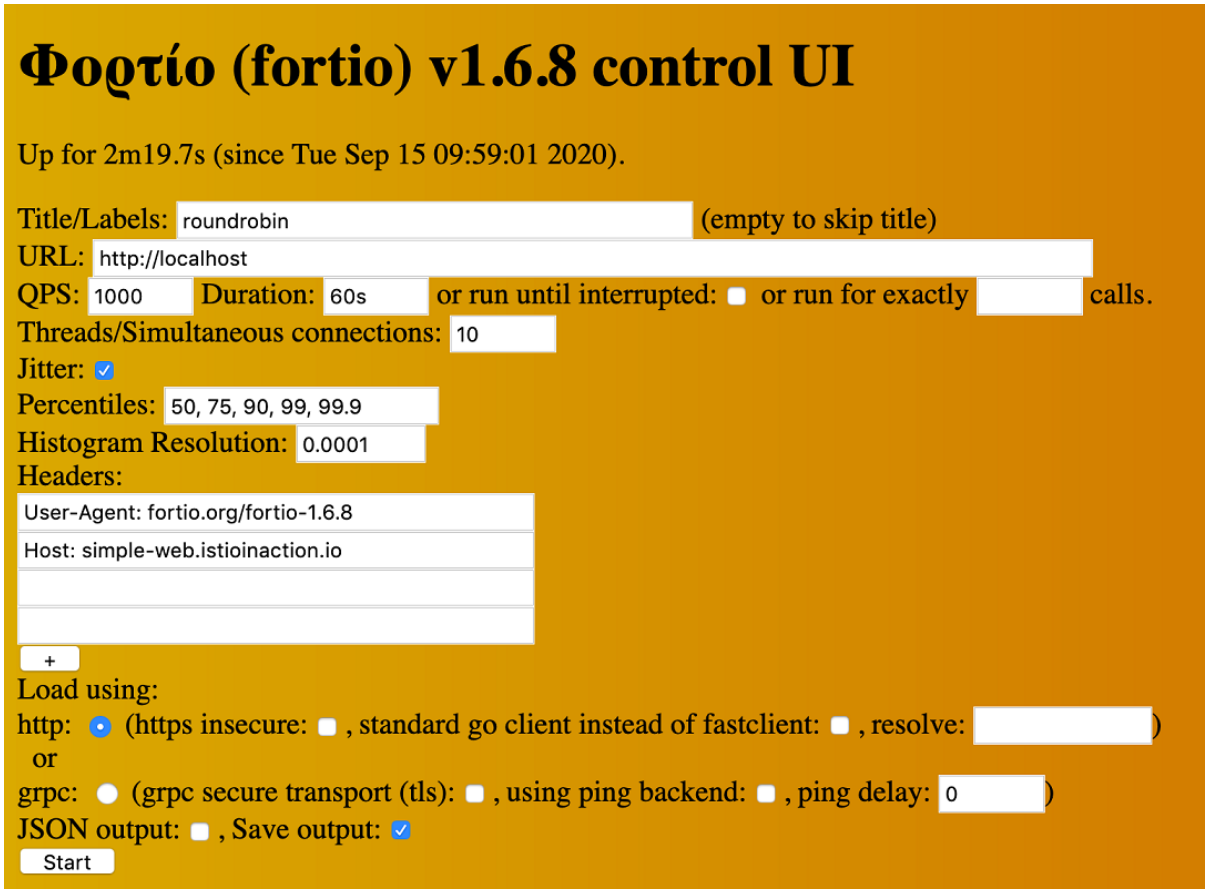


Figure 6.5 Fortio Server UI for setting up our load test

Fill in the following parameters:

- Title: roudrobin
- URL: http://localhost
- QPS: 1000
- Duration: 60s
- Threads: 10
- Jitter: checked
- Headers: "Host: simple-web.istioinaction.io"

Your entries should look like the image in Figure XX.

At this point, we can start running the test by clicking the "Start" button about half way down the Fortio web page.

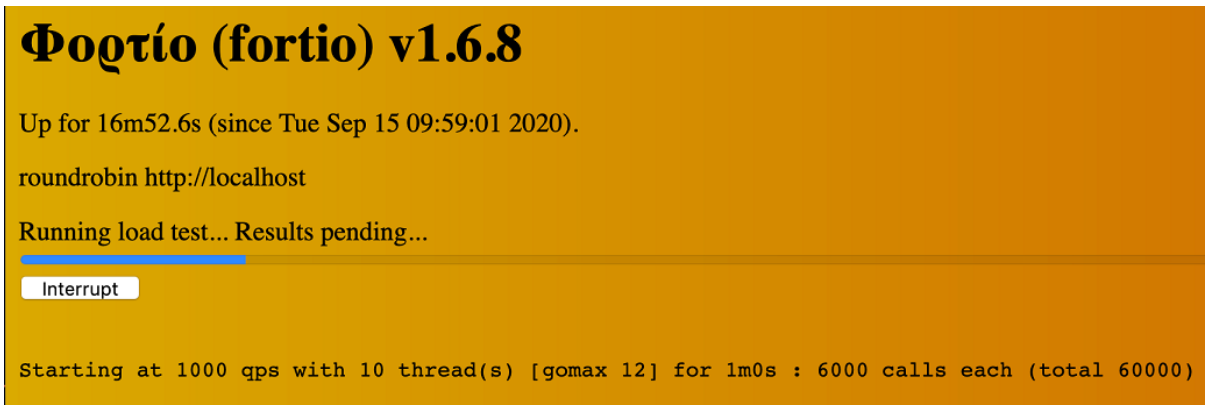


Figure 6.6 Fortio load test is in progress for 60s

Now we can wait for the test to complete. When it does, it will save a results file to your file system looking similar to `2020-09-15-101555_roundrobin.json`. You will also see a results graph:

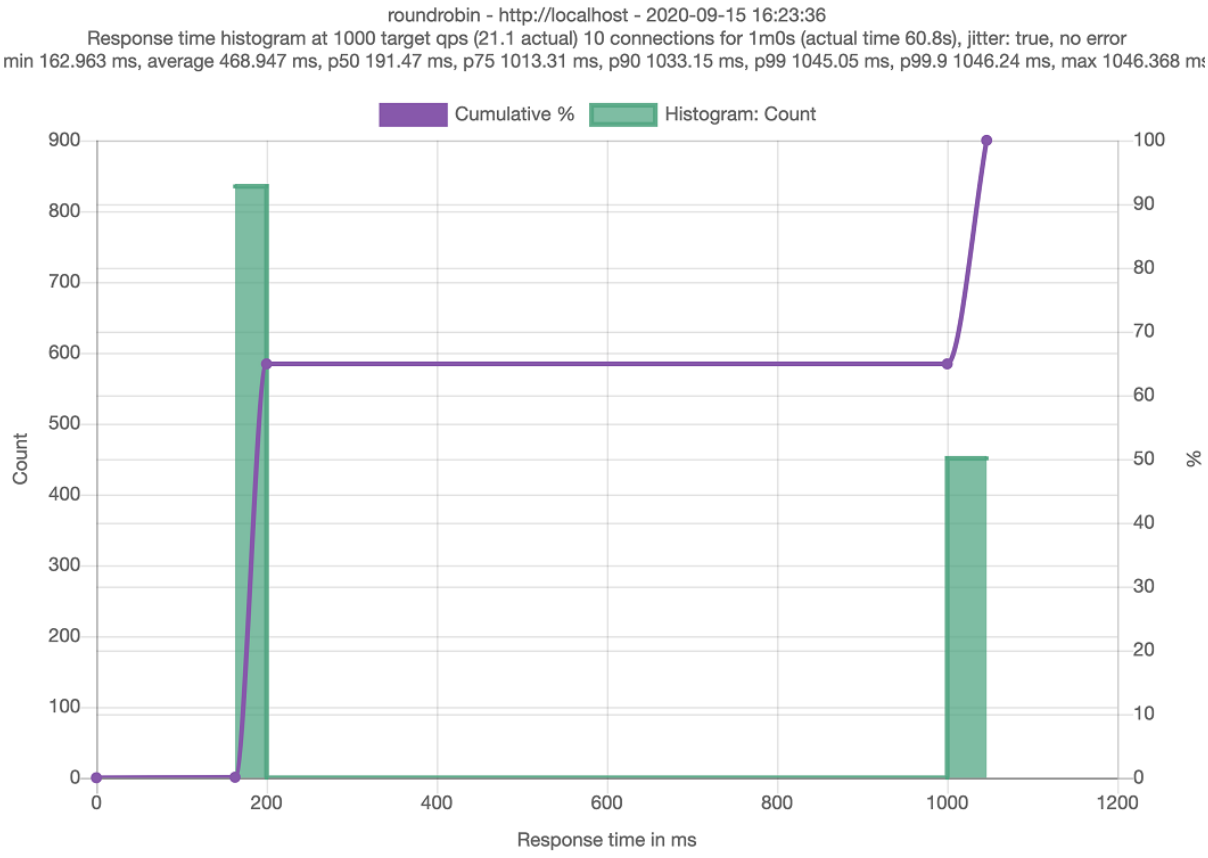


Figure 6.7 Results for load testing round robin client-side load balancing

For this `ROUND_ROBIN` load balancing strategy, we see resulting latencies as the following:

- 50% - 191.47ms
- 75% - 1013.31ms
- 90% - 1033.15ms
- 99% - 1045.05ms

- 99.9% - 1046.24ms

Now, let’s change the load-balancing algorithm to `RANDOM` and try the same load test again:

```
$  kubectl apply -f simple-backend-dr-random.yaml
destinationrule.networking.istio.io/simple-backend-dr configured
```

Now go back to the Fortio load-testing page (either click Back button, or the "Top" link). Fill in the information as you did before, but change the title to `random`

- Title: random
- URL: http://localhost
- QPS: 1000
- Duration: 60s
- Threads: 10
- Jitter: checked
- Headers: "Host: simple-web.istioinaction.io"

Click the "Start" button and wait for the results.

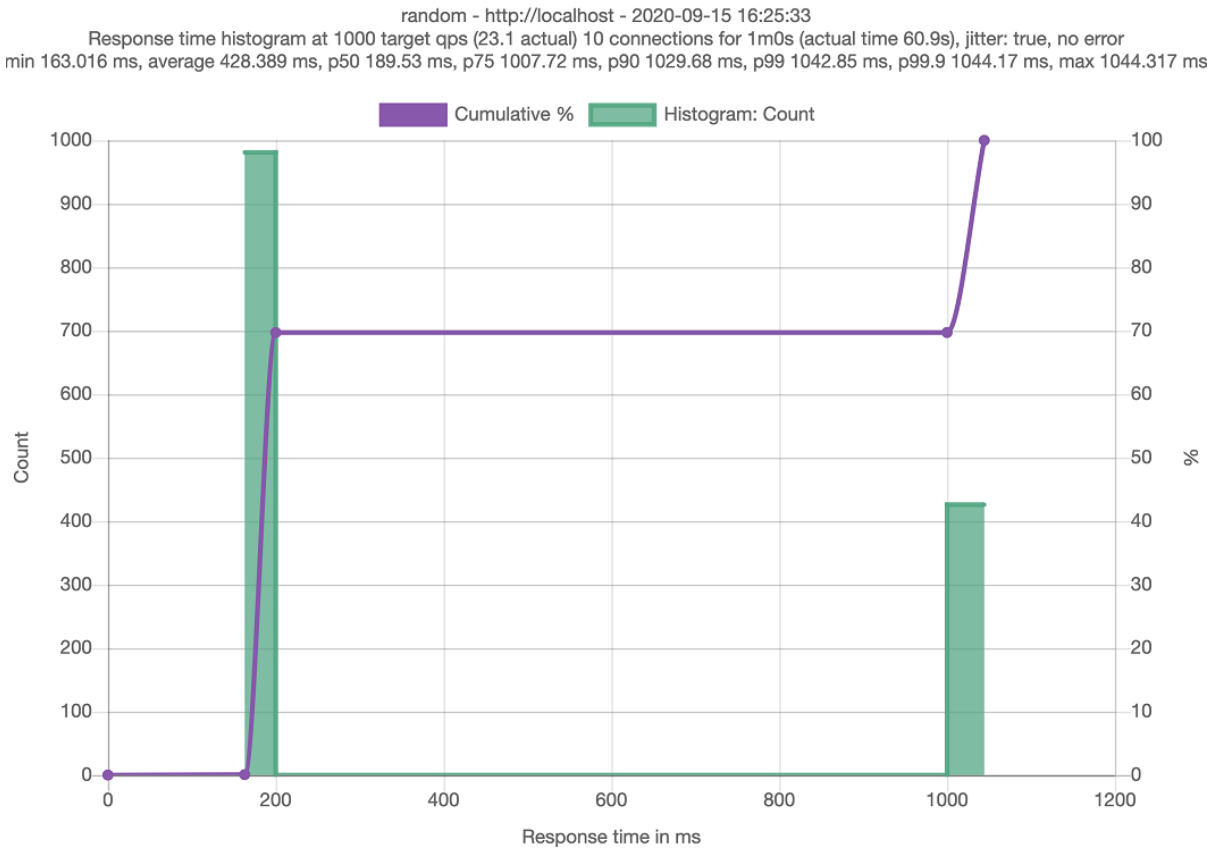


Figure 6.8 Results for load testing random client-side load balancing

For this `RANDOM` load balancing strategy, we see resulting latencies as the following:

- 50% - 189.53ms

- 75% - 1007.72ms
- 90% - 1029.68ms
- 99% - 1042.85ms
- 99.9% - 1044.17ms

Lastly, do the same for least-connection load balancing:

```
$  kubectl apply -f simple-backend-dr-least-conn.yaml
destinationrule.networking.istio.io/simple-backend-dr configured
```

- Title: leastconn
- URL: http://localhost
- QPS: 1000
- Duration: 60s
- Threads: 10
- Jitter: checked
- Headers: "Host: simple-web.istioinaction.io"

Click the "Start" button.

When it’s done running, you should see results:

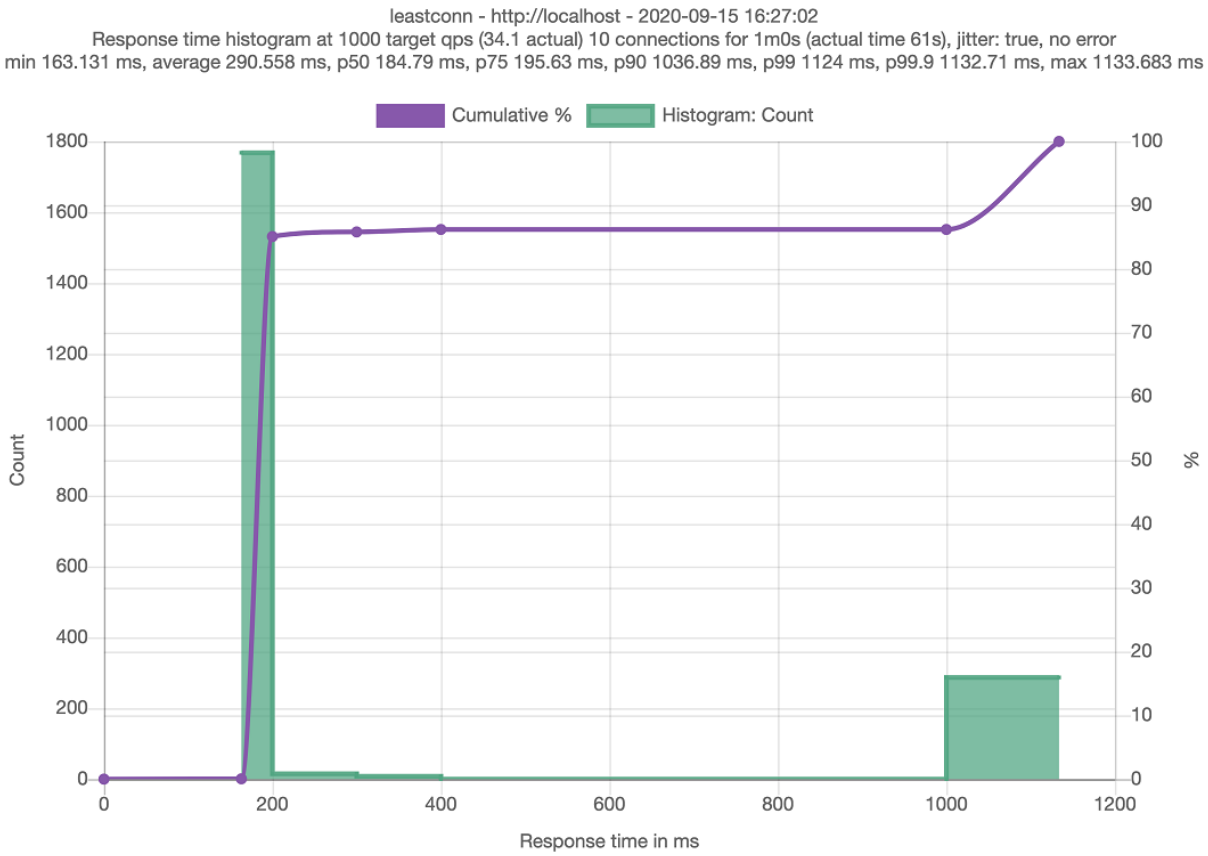


Figure 6.9 Results for load testing least-connection client-side load balancing

For this `LEAST_CONN` load balancing strategy, we see resulting latencies as the following:

- 50% - 184.79ms
- 75% - 195.63ms
- 90% - 1036.89ms
- 99% - 1124.00ms
- 99.9% - 1132.71ms

6.2.4 Understanding the different load balancing algorithms

We can see in the previous diagram a few things. First, the different load balancers indeed produce different results under realistic service latency behavior. Second, their results differ in both the histogram as well as their percentiles. Lastly, least connection performs better than both random or round robin. Let's see why.

Round robin and random are both simple load balancing algorithms. They're simple to implement and simple to understand. Round robin (or "next-in-loop") will deliver requests to endpoints in a successive loop. Random will uniformly pick an endpoint at random. With both you would expect a similar distribution. The challenge with either of these strategies is that the endpoints in the load balancer pool are not typically uniform even if they are indeed backed by the same service and resources. As we simulated in our tests previously, any of these endpoints could experience garbage collection or resource contention that introduces high latency and we see that both round robin and random do not take any runtime behavior into account.

The `LEAST_CONN` load balancer (actually, in Envoy it's implemented as **least request**) **does** take into account the latencies of the specific endpoints. When it sends requests out to endpoints it monitors the queue depths tracking active requests and will pick the endpoints with the fewest active requests in flight. We can see that using this type of algorithm we can avoid sending requests to endpoints that behave poorly and favor those that are responding quicker.

NOTE **Envoy least request load balancing**

Even though the Istio configuration refers to the least-request load balancing as `LEAST_CONN`, Envoy is in fact tracking request depths for endpoints, not connections. The load balancer will pick two random endpoints and check which has the fewest active requests. The load balancer will pick the one with the fewest active requests and do the same thing for successive load-balancing tries. This is known as the "power of two choices" and has been shown to be a good tradeoff (vs a full scan) when implementing a load balancer like this and achieving good results. See the Envoy documentation for more on this load balancer.

6.3 Locality aware load balancing

One of the roles of a control plane like Istio's is to understand the topology of services and how that topology may evolve. One advantage of understanding the overall topology of services in a service mesh is automatically making routing and load balancing decisions based on some heuristics like the location of services and peer services.

Istio supports a type of load balancing that gives weights and makes decisions to route based on where a particular workload is. For example, Istio can identify what region and availability zone into which a particular service is deployed and give priority services that are closer. If `simple-backend` service is deployed across multiple regions (us-west, us-east, europe-west) there will be multiple options to call `simple-backend`. If `simple-web` is deployed in `us-west` region, we want to calls from `simple-web` to `simple-backend` to stay local to `us-west`. If we treat all endpoints equally we will likely incur high latency as well as cost when we cross zones or regions.

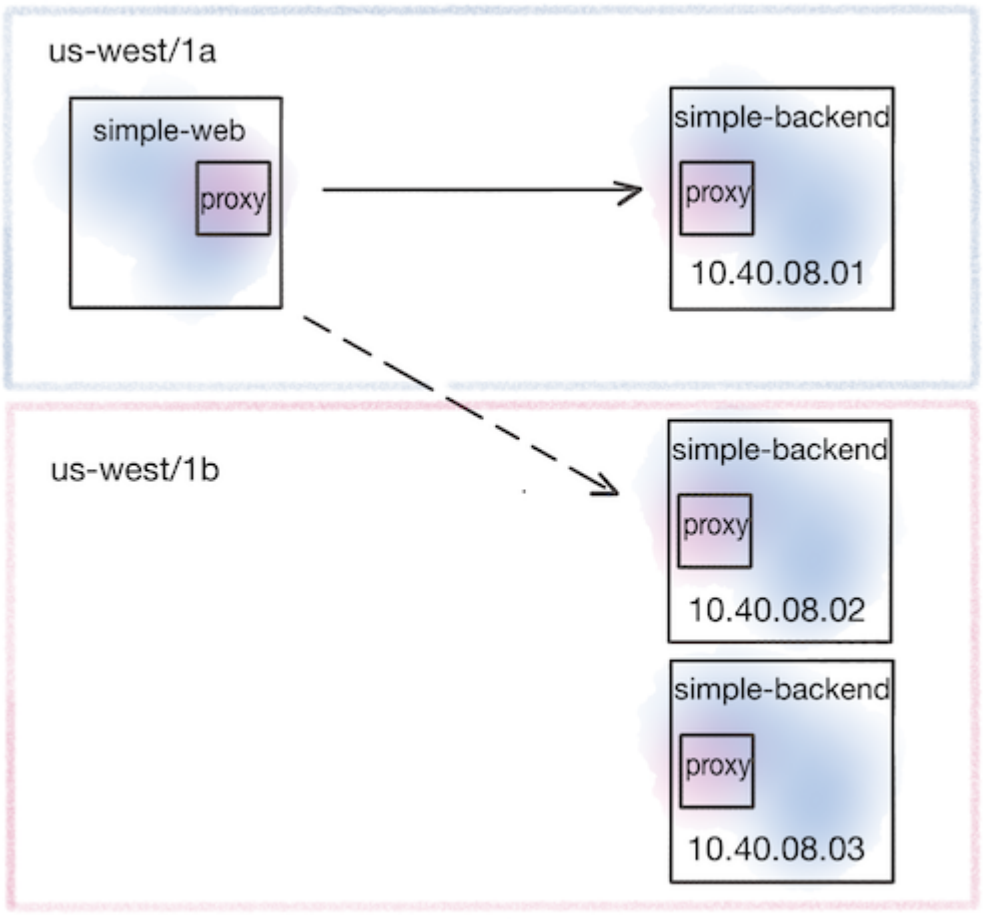


Figure 6.10 Prefer calling services in the same locality

6.3.1 Hands on with locality load balancing

Let's see locality load balancing in action. When deploying in Kubernetes, region and zone information can be added to labels on the Kubernetes nodes. For example labels `failure-domain.beta.kubernetes.io/region` and `failure-domain.beta.kubernetes.io/zone` allow to specify region and zone respectively. Often these labels are automatically added by cloud providers like Google Cloud or AWS. Istio will pick up these node labels and enrich the Envoy load-balancing endpoints with this locality information.

> **NOTE** **Kubernetes failure domain labels**
>
> In previous versions of Kubernetes' API, `failure-domain.beta.kubernetes.io/region` and `failure-domain.beta.kubernetes.io/zone` where the labels used to identify region and zone. In GA versions of the Kubernetes API, those labels have been replaced with `topology.kubernetes.io/region` and `topology.kubernetes.io/zone`. Just be aware, cloud vendors still use the older `failure-domain` labels. Istio looks for both.

Since we're using Docker for Desktop for this book, it's a little bit more difficult to demonstrate locality-aware routing using the out of the box locality information that Istio pulls. We could set up multiple nodes and label them with a desktop deployment of Kubernetes (using Kind or k3s, etc) but luckily for us, Istio gives an approach to explicitly set the locality of our workloads. We can label our pods with `istio-locality` and give it an explicit region/zone. For example, our `simple-web` deployment could look like this:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: simple-web
  name: simple-web
spec:
  replicas: 1
  selector:
    matchLabels:
      app: simple-web
  template:
    metadata:
      labels:
        app: simple-web
        istio-locality: us-west1.us-west1-a   (1)
    spec:
      serviceAccountName: simple-web
      containers:
      - image: nicholasjackson/fake-service:v0.14.1
        imagePullPolicy: IfNotPresent
        name: simple-web
        ports:
        - containerPort: 8080
          name: http
          protocol: TCP
```

```
        securityContext:
          privileged: false
```

❶ Locality label

When we deploy the `simple-backend` service, we'll annotate it with a couple different localities. We will deploy `simple-backend-1` in the same locality as `simple-web`, specifically in `us-west1-a`. We will deploy `simple-backend-2` in `us-west1-b`. In this case, the localities are in the same region but in different zones. Istio's ability to load balance across locality includes region, zone and even a more fine-grained "subzone".

Let's deploy these services:

```
$  kubectl apply -f simple-service-locality.yaml

deployment.apps/simple-web configured
deployment.apps/simple-backend-1 configured
deployment.apps/simple-backend-2 configured
```

We have now deployed our services with locality information. Istio's locality-aware load balancing is enabled by default. If you wish to disable it, you can configure the `meshConfig.localityLbSetting.enabled` setting to be false. With the locality information in place, we expect calls from `simple-web` in `us-west1-a` to go to `simple-backend` services that are deployed in the same zone, i.e., `us-west1-a`. In our example, we would expect all traffic to from `simple-web` to `simple-backend-1` which is in `us-west1-a`. Deployment `simple-backend-2` service is in `us-west1-b` which is not in the same zone as `simple-web` so we would only expect traffic to go to that endpoint if the services in `us-west1-a` start to fail.

If we call our Istio ingress gateway (which was configured from the previous section to accept traffic and route to `simple-web`):

```
$  for in in {1..10}; do \
curl -s -H "Host: simple-web.istioinaction.io" localhost \
| jq .upstream_calls[0].body; printf "\n"; done

"Hello from simple-backend-1"
"Hello from simple-backend-1"
"Hello from simple-backend-2"
"Hello from simple-backend-2"
"Hello from simple-backend-2"
"Hello from simple-backend-1"
"Hello from simple-backend-2"
"Hello from simple-backend-1"
"Hello from simple-backend-1"
"Hello from simple-backend-2"
```

What happened here? We see that the traffic has been load balanced across all of the available endpoints that make up `simple-backend` service. It appears the locality information was not taken into account.

For locality-aware load balancing to work in Istio, we need to configure one last piece of the

puzzle: health checking. Without health checking, Istio does not know which endpoints in the load balancing pool are unhealthy and what heuristics to use to spill over into the next locality. Let's add a passive health checking configuration by configuring outlier detection for the `simple-backend` service. Outlier detection passively watches the behavior of endpoints and whether they appear healthy or not. It does this by tracking errors an endpoint may return and marking them unhealthy. We will cover outlier detection in more detail in the next sections.

```
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: simple-backend-dr
spec:
  host: simple-backend.istioinaction.svc.cluster.local
  trafficPolicy:
    connectionPool:
      http:
        http2MaxRequests: 10
        maxRequestsPerConnection: 10
    outlierDetection:
      consecutiveErrors: 1
      interval: 1m
      baseEjectionTime: 30s
```

Let's apply this DestinationRule. From the `chapters/chapter6` folder in the source code for this book, you can run:

```
$  kubectl apply -f simple-backend-dr-outlier.yaml
destinationrule.networking.istio.io/simple-backend-dr created
```

Now let's try calling the `simple-web` service through the Istio ingress gateway:

```
$  for in in {1..10}; do \
curl -s -H "Host: simple-web.istioinaction.io" localhost \
| jq .upstream_calls[0].body; printf "\n"; done

"Hello from simple-backend-1"
"Hello from simple-backend-1"
"Hello from simple-backend-1"
"Hello from simple-backend-1"
"Hello from simple-backend-1"
"Hello from simple-backend-1"
"Hello from simple-backend-1"
"Hello from simple-backend-1"
"Hello from simple-backend-1"
"Hello from simple-backend-1"
```

We see that all traffic went to the `simple-backend` service which is in the same zone as `simple-web`. If we want to see traffic spill over to another availability zone, let's put the `simple-backend-1` service into a state that it misbehaves. Whenever `simple-web` calls `simple-backend-1`, it will get a HTTP 500 error 100% of the time.

```
$  kubectl apply -f simple-service-locality-failure.yaml
deployment.apps/simple-backend-1 configured
```

Now if we call our service through the Istio ingress gateway, we should see that all traffic will go

to the `simple-backend-2` service. This happens because `simple-backend-1` which is the same locality as `simple-web` returns with an HTTP 500 error and gets marked as unhealthy. When enough of the endpoints in the same locality as the `simple-web` service are unhealthy, load balancing will automatically spill over to the next-closest locality. In this case, it would be the endpoints in the `simple-backend-2` deployment.

```
$  for in in {1..10}; do \
curl -s -H "Host: simple-web.istioinaction.io" localhost \
| jq .upstream_calls[0].body; printf "\n"; done

"Hello from simple-backend-2"
"Hello from simple-backend-2"
"Hello from simple-backend-2"
"Hello from simple-backend-2"
"Hello from simple-backend-2"
"Hello from simple-backend-2"
"Hello from simple-backend-2"
"Hello from simple-backend-2"
"Hello from simple-backend-2"
"Hello from simple-backend-2"
```

Now we get the locality-aware load balancing outcome we would expect when services in a particular locality do not behave well. Note, this locality aware load balancing we are seeing here is within a single cluster. We will explore locality-aware load balancing behavior across multiple clusters in Chapter XX.

6.3.2 More control over locality load balancing

In the previous section we saw locality-aware load balancing in action. The last part to know about locality-aware load balancing is that you can control some of the behavior of how it works. By default, Istio's service-proxy will send all traffic to services in the same locality as we saw and only spill over when there are failures/unhealthy endpoints. We can influence this behavior in scenarios where we may wish to load balance some of the traffic across multiple locality zones. We may wish to do this when we expect services in a particular locality may become overloaded because of some peak or seasonal traffic.

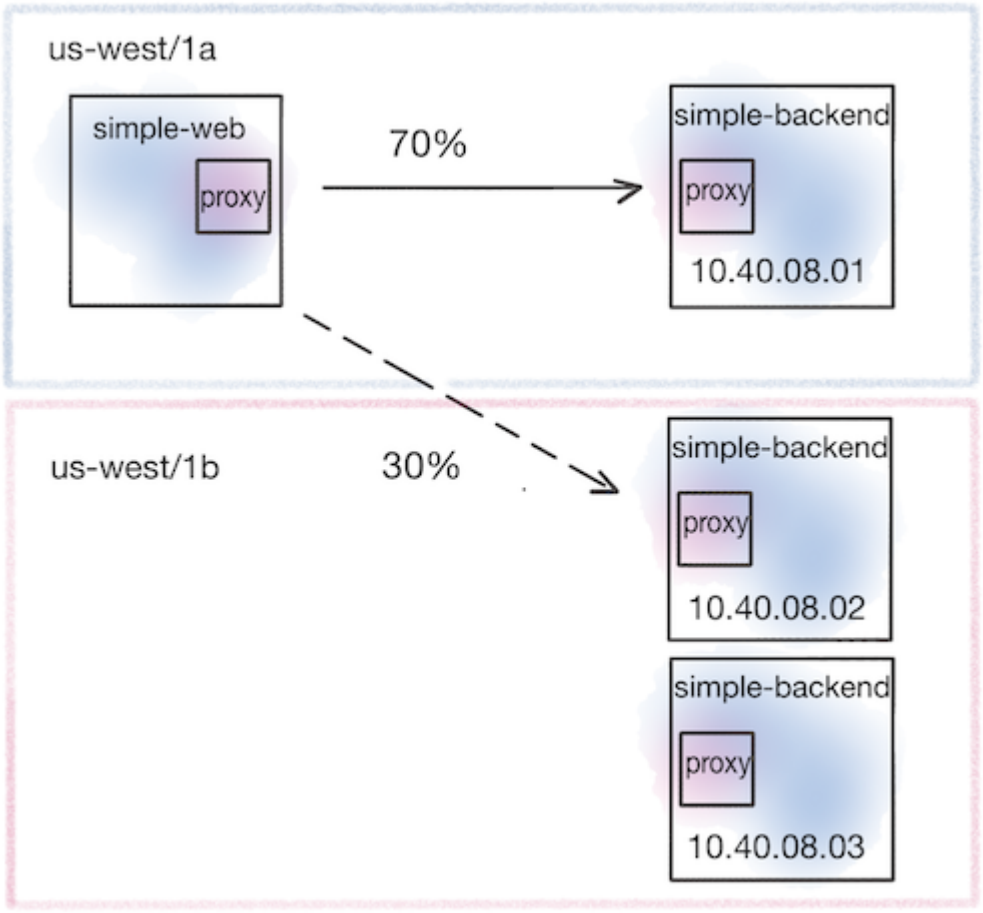


Figure 6.11 Define the locality weights more explicitly

In the previous examples we introduced a misbehaving service. Let's restore our services so they all behave correctly and return HTTP 200 responses:

```
$  kubectl apply -f simple-service-locality.yaml
deployment.apps/simple-web unchaged
deployment.apps/simple-backend-1 configured
deployment.apps/simple-backend-2 unchanged
```

Let's say we have some expected load coming that we don't expect the services within a certain zone or region to be able to handle. We will want to spill over to a neighboring locality so that 70% of traffic goes to the locality closest and 30% goes to to the neighboring locality. Following our example from above, we will send 70% of the traffic destined for `simple-backend` service to `us-west1-a` and 30% of traffic to `us-west1-b`. This roughly translates to 70% of traffic to `simple-backend-1` service and 30% of traffic to `simple-backend-2`.

To accomplish this configuration, we'll specify locality load balancing preferences in a DestinationRule resource:

```
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: simple-backend-dr
```

```
spec:
  host: simple-backend.istioinaction.svc.cluster.local
  trafficPolicy:
    loadBalancer:   ❶
      localityLbSetting:
        distribute:
        - from: us-west1/us-west1-a/*   ❷
          to:
            "us-west1/us-west1-a/*": 70   ❸
            "us-west1/us-west1-b/*": 30   ❹
    connectionPool:
      http:
        http2MaxRequests: 10
        maxRequestsPerConnection: 10
    outlierDetection:
      consecutive5xxErrors: 1
      interval: 5s
      baseEjectionTime: 30s
      maxEjectionPercent: 100
```

❶ Add load balancer config

❷ Origin zone

❸ Dest zone

❹ Dest zone

Let’s apply this to take effect:

```
$  kubectl apply -f simple-backend-dr-outliter-locality.yaml
destinationrule.networking.istio.io/simple-backend-dr configured
```

Let’s now call our service once again:

```
$  for in in {1..10}; do \
curl -s -H "Host: simple-web.istioinaction.io" localhost \
| jq .upstream_calls[0].body; printf "\n"; done

"Hello from simple-backend-1"
"Hello from simple-backend-1"
"Hello from simple-backend-1"
"Hello from simple-backend-1"
"Hello from simple-backend-2"
"Hello from simple-backend-1"
"Hello from simple-backend-1"
"Hello from simple-backend-1"
"Hello from simple-backend-2"
"Hello from simple-backend-1"
```

At this point we can see some of the requests were load balanced mostly to the closest locality but with some wiggle room to spill over to the next-closest locality. Note, this is not exactly the same as controlling the traffic explicitly like we did in Chapter 5. With traffic routing, we can definitely control the traffic between different subsets of our services, typically when there are different classes of service or version of service within the overall group. In this case, we’re weighting the traffic based on the deployed topology of the services, independent of subsets.

These are not mutually exclusive concepts either. They can be layered in such a way that fine-grained traffic control and routing that we saw in Chapter 5 can be applied on top of the location aware load balancing we see in this section.

6.4 Transparent timeouts and retries

One of the biggest problems to watch out for when building systems that rely on components distributed over the network is latency and failures. We saw in earlier sections how we can use Istio to mitigate these challenges using load balancing and locality. What happens if these network calls take too long? Or what happens if we experience intermittent failures either as a result of this latency or because of other network factors? How can Istio help with these issues?

Istio allows us to configure various types of timeouts and retries to overcome the inherent network unreliability.

6.4.1 Timeouts

We should discuss these different timeouts for this section, but won't provide examples until we get this sorted:

https://github.com/istio/istio/issues/6861

- connection timeout (in meshConfig) testing: https://stackoverflow.com/questions/100841/artificially-create-a-connection-timeout-error
- idle timeout (in env variables) can test with tcp services
- request timeout (in virtualservice) https://istio.io/latest/docs/reference/config/networking/virtual-service/#HTTPRoute easiest to test, default no timeout

6.4.2 Retries

When calling a service and experiencing intermittent network failures, we may want the application to retry the request. If we don't retry the request we make our services susceptible to common and expected failures that could deliver a bad user experience. On the other hand, we have to balance out the fact that unbridled retries can contribute to degraded system health including causing cascading failures. If a service is legitimately overloaded and misbehaving, retrying requests would only amplify this degraded situation. Let's take a look at the retry options provided by Istio.

Before we begin, let's set our sample services back to some sane defaults:

```
$  kubectl apply -f simple-web.yaml
$  kubectl apply -f simple-backend.yaml
$  kubectl delete destinationrule simple-backend-dr
```

Istio by default has retries enabled. Let's first understand what default behavior exists before we

start fine-tuning it. To get started with the retry section, let's disable the default retries for our sample application by configuring VirtualService resources to set max retrys to `0`.

```
$  kubectl apply -f simple-service-disable-retry.yaml
virtualservice.networking.istio.io/simple-web-vs-for-gateway configured
virtualservice.networking.istio.io/simple-backend-vs created
```

Let's deploy a version of the `simple-backend` service that has periodic (75%) failures. In this case we'll have one of the three endpoints (`simple-backend-1`) returning HTTP 503 on 75% of its calls.

```
$  kubectl apply -f simple-backend-periodic-failure-503.yaml
deployment.apps/simple-backend-1 configured
```

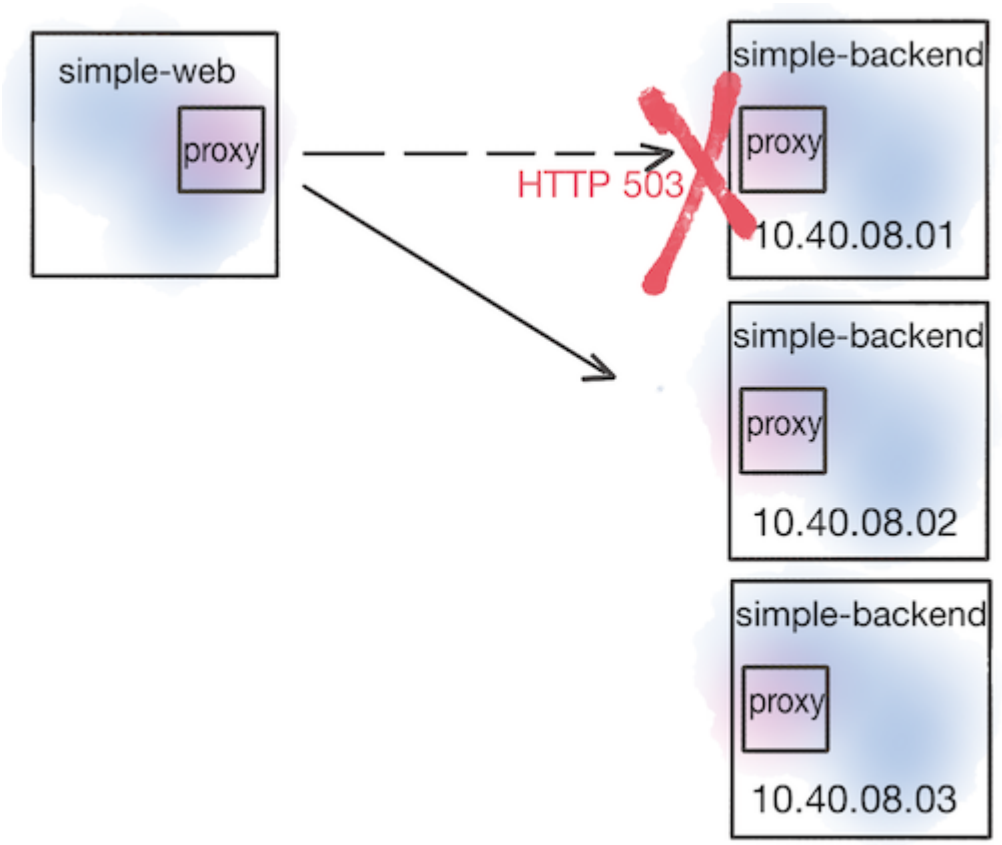


Figure 6.12 Service simple-web calling simple-backend with failures from simple-backend-1

If we call the service a number of times, we should see some failures:

```
$  for in in {1..10}; do curl -s \
-H "Host: simple-web.istioinaction.io" localhost \
| jq .code; printf "\n"; done

200
500   ❶
200
200
200
500   ❷
```

```
200
200
200
200
```

❶ Expected failure

❷ Expected failure

By default, Istio will try a call and if it fails, will try 2 more times. This default retry will only apply to certain situations. These default situations are typically safe to retry a request:

- `connect-failure`
- `refused-stream`
- `unavailable` (grpc status code 14)
- `cancelled` (grpc status code 1)
- `retriable-status-codes` (default to HTTP 503 in istio)

In the previous configurations, we disabled the default retry policy. Let's put back the default retry policy for one of the services by deleting our default value override. When the `simple-web` service calls `simple-backend`, we'll restore the default retry policy:

```
$  kubectl delete virtualservice simple-backend-vs
virtualservice.networking.istio.io "simple-backend-vs" deleted
```

If we call our service again, we should see no failures:

```
$  for in in {1..10}; do curl -s \
-H "Host: simple-web.istioinaction.io" localhost \
| jq .code; printf "\n"; done

200
200
200
200
200
200
200
200
200
200
```

Although there were failures (as we saw earlier) they are not bubbled up to the caller because we leverage Istio's default retry policy to work around those errors. By default, HTTP 503 is one of the retriable status codes. If we take a look at a VirtualService retry policy, we can see what parameters are configurable out of the box for retries:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: simple-backend-vs
spec:
  hosts:
  - simple-backend
  http:
```

```
  - route:
    - destination:
        host: simple-backend
    retries:
      attempts: 2   ❶
      retryOn: gateway-error,connect-failure,retriable-4xx   ❷
      perTryTimeout: 300ms   ❸
      retryRemoteLocalities: true   ❹
```

❶ Max number of retries for a request

❷ On what errors to retry

❸ Timeout for individual retry

❹ Whether to retry endpoints in other localities

The various settings for retries give us some control over the behavior of retry (how many, how long, which endpoints to retry) as well as on which status codes to retry. As we mentioned previously, not all requests can or should be retried.

For example, if we deploy our `simple-backend` service to return HTTP 500 codes, the default retry behavior would not catch that:

```
$  kubectl apply -f simple-backend-periodic-failure-500.yaml
deployment.apps/simple-backend-1 configured
```

If we call our service again, we should see those HTTP 500 failures bubble up:

```
$  for in in {1..10}; do curl -s \
-H "Host: simple-web.istioinaction.io" localhost \
| jq .code; printf "\n"; done

500
200
500
200
200
200
200
500
200
200
```

We see HTTP 500 is not part of the status codes that are retried. Let's use a VirtualService retry policy that retries on all HTTP 500 codes (including `connect-failure` and `refused-stream`):

```
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: simple-backend-vs
spec:
  hosts:
  - simple-backend
  http:
  - route:
    - destination:
        host: simple-backend
```

```
    retries:
      attempts: 2
      retryOn: 5xx   ❶
```

❶ retryOn HTTP 5xx

If we apply this VirtualService:

```
$  kubectl apply -f simple-backend-periodic-failure-500.yaml
deployment.apps/simple-backend-1 configured
```

We should not see any 500 errors bubble up:

```
$  for in in {1..10}; do curl -s \
-H "Host: simple-web.istioinaction.io" localhost \
| jq .code; printf "\n"; done

200
200
200
200
200
200
200
200
200
200
```

For more about the available `retryOn` configurations, see the Envoy documentation on retries.

RETRIES IN TERMS OF TIMEOUTS

Each retry will have its own `perTryTimeout`. One thing to note about this setting is that the `perTryTimeout` value multiplied by the attempts must be lower than the overall request timeout (described in previous section). For example, an overall timeout of 1s and retry policy of three attempts with per retry timeout of 500ms won't work. The overall request timeout will kick in before all of the retries get a chance. Also keep in mind there is a backoff delay between retries which also goes against the overall request tiemout. We describe more about the backoff next.

HOW IT WORKS

When a request flows through the Istio service proxy, if it fails to be delivered upstream, it will be marked failed and retried up to the max `attempts` field defined in the VirtualService. This means with an `attempts` of `2`, the request will actually be delivered up to `3` times: once for the original request and two for the retries. Between retries, Istio will "backoff" the retry with a base of 25ms. This means for each successive retry, it will backoff (wait) until (25ms x attempt #) so to stagger the retries. At the moment this retry base is fixed, but as we will see in the next section we can make changes to parts of the Envoy API that are not exposed by Istio.

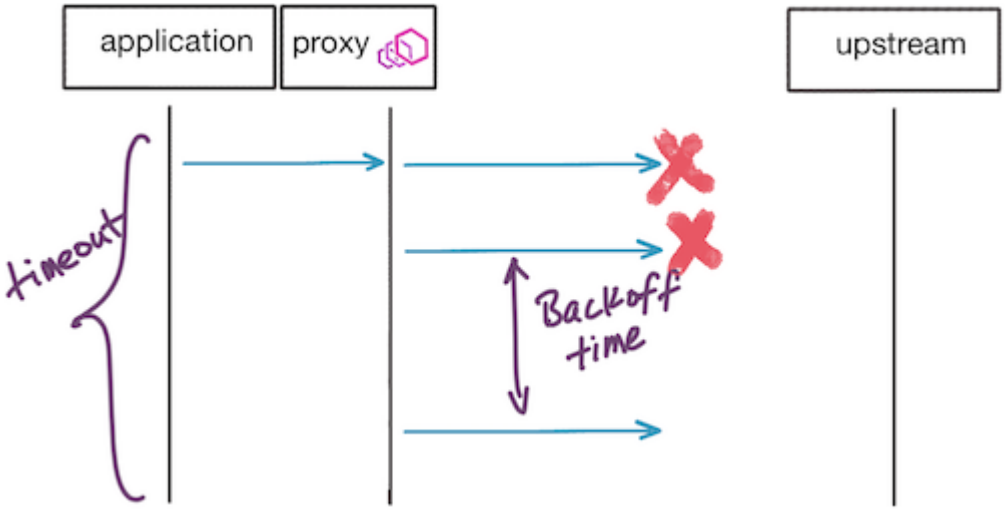


Figure 6.13 Request flow on retries when requests fail

As mentioned, Istio by default with have retry `attempts` set to `2`. You may wish to override this so that different layers of your system retry different amounts. Naive retry settings (like the default) can lead to significant retry "thundering herd" problem. For example if a service chain is 5 calls deep and each step can retry a request 2 times, we could end up with 32x requests for each incoming request. If a resource at the end of the chain is overloaded, this extra load could overwhelm it to the point that it falls over. An option to deal with this is to limit the retry attempts at the edges of your architecture to 1 or 0 and only retry deep into your call stack with the intermediate components not retrying. This may not work very well either so another approach is to put caps on the total overall retries. We can do that with retry budgets, however budgets are not yet exposed in the Istio API. We'll explore those in the next section.

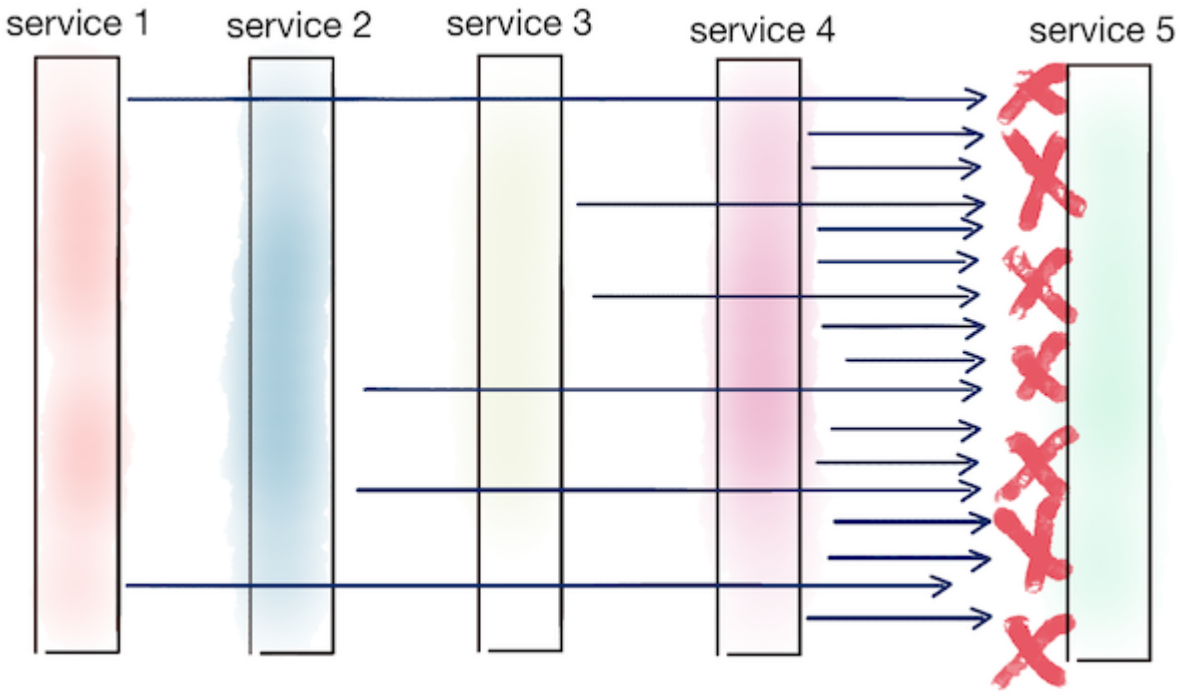


Figure 6.14 Thundering herd effect when retries compound each other

Lastly, retries will be attempted against endpoints in their own locality by default. There is a setting `retryRemoteLocalities` that does affect this behavior. If we set this to `true`, Istio will allow retries to spill over to other localities. This may come in handy before outlier detection determines that the locally-preferred endpoints are misbehaving.

6.4.3 Advanced retries

In the previous section we saw how Istio can help make our services resilient to intermittent network failures by using automatic retries. We also saw some of the parameters we can tune for retry use cases. Some of the retry capabilities take into account some defaults that aren't easy to change like the backoff retry time or the default retriable status codes. By default the backoff time is 25ms and the retriable code is limited to HTTP 503. Even though the Istio API doesn't expose these configurations at the time of writing, we can use the Istio extension API to alter these values directly in the Envoy configuration. We will use the EnvoyFilter API to do this.

> **NOTE** **EnvoyFilter API**
>
> Just note that the EnvoyFilter API is a "break glass" solution. Istio's API in general is an abstraction over the underlying dataplane. The underlying Envoy API may change at any time between releases of Istio so be sure to validate any EnvoyFilter you put into production. Do not assume any backward compatibility here.

```
apiVersion: networking.istio.io/v1alpha3
kind: EnvoyFilter
metadata:
  name: simple-backend-retry-status-codes
  namespace: istioinaction
spec:
  workloadSelector:
    labels:
      app: simple-web
  configPatches:
  - applyTo: HTTP_ROUTE
    match:
      context: SIDECAR_OUTBOUND
      routeConfiguration:
        vhost:
          name: "simple-backend.istioinaction.svc.cluster.local:80"
    patch:
      operation: MERGE
      value:
        route:
          retry_policy:   ❶
            retry_back_off:
              base_interval: 50ms   ❷
            retriable_status_codes:   ❸
            - 402
            - 403
```

❶ Envoy configuration directly

❷ Increase the base interval

❸ Add retriable codes

Don’t worry about the specifics of the EnvoyFilter API; we will cover this in more detail in Chapter XX. But what we can see is that we use the Envoy API directly here to configure/override retry policy settings. Let’s apply these configurations:

```
$  kubectl apply -f simple-backend-ef-retry-status-codes.yaml
envoyfilter.networking.istio.io/simple-backend-retry-status-codes configured
```

We also want to update our `retryOn` field to include the `retriable-status-codes`:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: simple-backend-vs
spec:
  hosts:
  - simple-backend
  http:
  - route:
    - destination:
        host: simple-backend
    retries:
      attempts: 2
      retryOn: 5xx,retriable-status-codes   ❶
```

❶ include the retriable status codes

```
$  kubectl apply -f simple-backend-vs-retry-on.yaml
virtualservice.networking.istio.io/simple-backend-vs configured
```

Lastly let’s update our `sample-backend` service to return HTTP 408 (timeout) and verify that we continue to get HTTP 200:

```
$  kubectl apply -f simple-backend-periodic-failure-408.yaml
deployment.apps/simple-backend-1 configured

$  for in in {1..10}; do curl -s \
-H "Host: simple-web.istioinaction.io" localhost \
| jq .code; printf "\n"; done

200
200
200
200
200
200
200
200
200
200
```

RETRY BUDGETS

As mentioned earlier in this section about retries, we run the risk of significantly overloading our system with naive retries and without thinking about layering our retries. We should consider limiting our retries by a fixed number or by at least a percentage of overall requests. With Istio, we can do both of these. The first, limiting parallel retries by a fixed number, can be done with the Istio DestinationRule API. The second, effectively a retry budget, can be done with the EnvoyFilter API that we saw in previous sections.

To limit max number of parallel retries to a fixed number, we can configure `maxRetries` in our DestinationRule:

```
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: simple-backend-dr
spec:
  host: simple-backend.istioinaction.svc.cluster.local
  trafficPolicy:
    connectionPool:
      http:
        maxRetries: 3   ❶
        http2MaxRequests: 10
        maxRequestsPerConnection: 10
```

❶ Specify maxRetries

Note, that in this example, we limit the number of parallel retries to `3`. We've left out any other DestinationRule settings like outlier detection just to keep the configuration text small for this example. We explore outlier detection in more detail in the next section. In a production environment, you'll probably want to set the max retries to a number greater than three but capped at **some** value that gives enough confidence in retrying the network but not overloading the upstream services. By default, Istio will set `maxRetries` to a very high number (Max value of unsigned integer), so some care should be put into this configuration.

To configure a retry budget for calling `simple-service` in this example, we can use the EnvoyFilter API and specify the configuration directly on the upstream cluster:

```
apiVersion: networking.istio.io/v1alpha3
kind: EnvoyFilter
metadata:
  name: simple-backend-retry-status-codes
  namespace: istioinaction
spec:
  workloadSelector:
    labels:
      app: simple-web
  configPatches:
  - applyTo: CLUSTER
    match:
      context: SIDECAR_OUTBOUND
      cluster:
        portNumber: 80
        service: simple-backend.istioinaction.svc.cluster.local
```

```
    patch:
      operation: MERGE
      value:
        circuit_breakers:
          thresholds:
            - retry_budget:
                budget_percent: 20.0
                min_retry_concurrency: 5
```

We will need to wait for this before we complete this section? Or we take it out:

https://github.com/istio/istio/issues/27419

REQUEST HEDGING

This last treatment of retries will center around an advanced topic that is also not exposed in the Istio API directly. When a request reaches its threshold and timesout, we can optionally configure Envoy under the covers to perform what's called "request hedging". With request hedging, if a request timesout, Envoy can send another request to a different host to "race" the original, timed-out request. In this the case if the "raced" request returns successfully, it's response is sent to the original downstream caller. If the original request actually returns, but before the raced request returns, it will be returned to the downstream caller.

To set up request hedging, you can use the following EnvoyFilter:

```
apiVersion: networking.istio.io/v1alpha3
kind: EnvoyFilter
metadata:
  name: simple-backend-retry-hedge
  namespace: istioinaction
spec:
  workloadSelector:
    labels:
      app: simple-web
  configPatches:
  - applyTo: VIRTUAL_HOST
    match:
      context: SIDECAR_OUTBOUND
      routeConfiguration:
        vhost:
          name: "simple-backend.istioinaction.svc.cluster.local:80"
    patch:
      operation: MERGE
      value:
        hedge_policy:
          hedge_on_per_try_timeout: true
```

As we've seen in this section, the topic of timeouts and retries is not that simple. There are challenges with coming up with good timeouts and retry policies for services, especially considering how they can be chained together. Misconfigured timeouts and retries can amplify undesirable behaviors in a system architecture to the extent they overload the system and cause cascading failures. One last piece of the puzzle for building resilient architectures is skipping

retries altogether and instead of retrying, we fail fast. Instead of promoting more load, we can limit load for a period of time to allow upstream systems to recover. For that we can employ circuit breaking.

6.5 Circuit breaking with Istio

We use circuit-breaking functionality to help guard against partial or cascading failures. We want to reduce traffic to unhealthy systems so we don't continue to overload them and prevent them from recovering. For example, if `simple-web` service calls out to `simple-backend` service, and the `simple-backend` service returns errors for successive calls, instead of continuous retries and adding more stress to the system, we may want to halt any calls to `simple-backend`. This approach is similar in spirit to how a circuit breaker works in your electrical system for your house. If we experience shorts in the system or repeated faults, a circuit breaker is designed to open the circuit to protect the rest of the system. The circuit-breaker pattern forces our application to deal with the fact our network calls can and do fail and help safeguard the overall system from cascading failures.

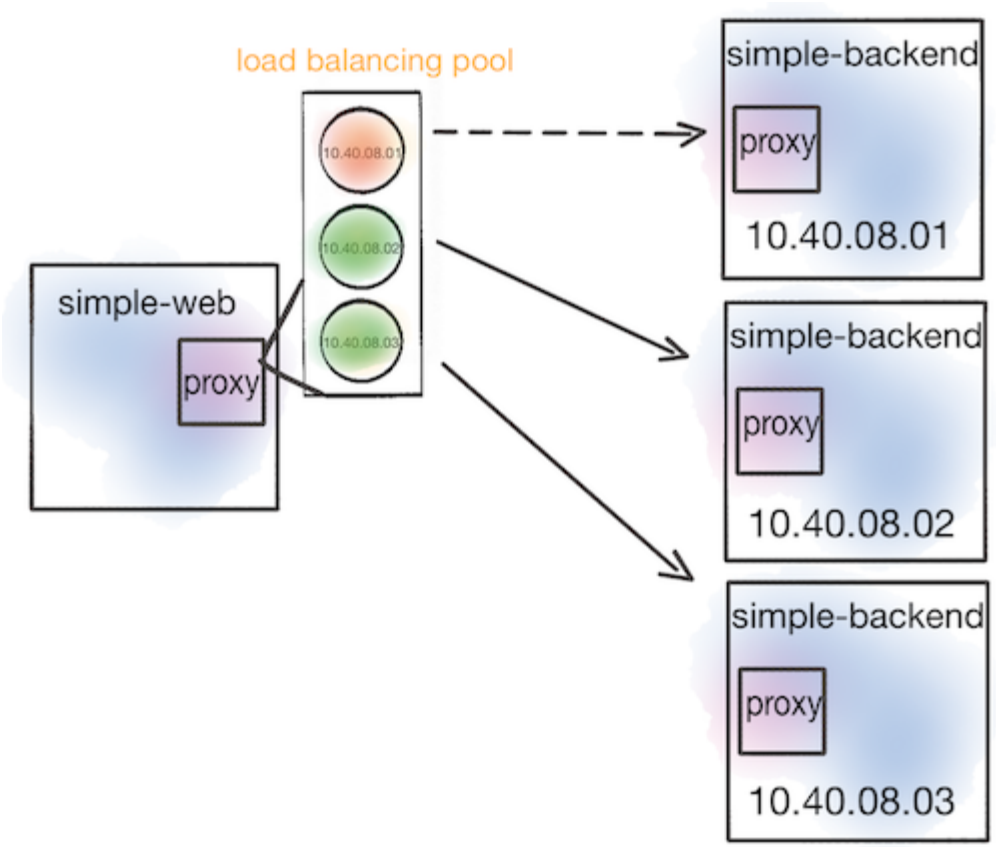


Figure 6.15 Circuit breaking endpoints that don't behave correctly

Istio doesn't have an explicit configuration called "circuit breaker", but it does provide two controls for limiting load on backend services, especially those that are experiencing issues.

The first is to control how many connections and outstanding requests are allowed to a specific

service. We use this control to guard against services that slow down and end up backing up the client. If there are 10 requests in flight to a particular service and that number keeps growing for the same amount of inbound load, it won't make sense to continue to send more requests. Sending more requests could overwhelm the upstream service. In Istio we will use the `connectionPool` settings in a DestinationRule to limit the number of connections and requests that may be piling up when calling a service. If too many requests pile up, we can short-circuit them (fail fast) and return to the client.

The second control is to observe health of endpoints in the load-balancing pool and evict misbehaving endpoints for a period of time. If certain hosts in a service pool are experiencing failures, then we can skip sending traffic to them. If we exhaust all hosts, the circuit is effectively "open" for a period of time. Let's see how to implement each of these circuit-breaking controls with Istio.

6.5.1 Guarding against slow services with connection pool control

To set up the examples here, let's first scale down the `simple-backend` service so there is only a single pod. We can take all of the `simple-backend-2` services down to replica of `0`.

```
$  kubectl scale deploy/simple-backend-2 --replicas=0
deployment.apps/simple-backend-2 scaled
```

Next let's deploy the version of `simple-backend` service which introduces a 1s delay in responses:

```
$  kubectl apply -f simple-backend-delayed.yaml
deployment.apps/simple-backend-1 configured
```

If there are any existing DestinationRules from previous sections, let's delete them:

```
$  kubectl delete destinationrule --all
```

Now we should be in position to start testing out Istio's connection-limiting circuit breaking. Let's run a very simple load test with one single connection (`-c 1`) sending one request per second (`-qps 1`). Also note since the backend returns in approximately 1s, we should have smooth traffic and 100% successful responses:

```
$  fortio load -H "Host: simple-web.istioinaction.io" \
-quiet -jitter -t 30s -c 1 -qps 1 http://localhost/

# target 50% 1.27611
# target 75% 1.41565
# target 90% 1.49938
# target 99% 1.54961
# target 99.9% 1.55464
Sockets used: 1 (for perfect keepalive, would be 1)
Jitter: true
Code 200 : 30 (100.0 %)
All done 30 calls (plus 1 warmup) 1056.564 ms avg, 0.9 qps
```

Now let's start to introduce some connection and request limits and see what happens. We will start with a very simple set of limits:

```
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: simple-backend-dr
spec:
  host: simple-backend.istioinaction.svc.cluster.local
  trafficPolicy:
    connectionPool:
      tcp:
        maxConnections: 1   ❶
      http:
        http1MaxPendingRequests: 1   ❷
        maxRequestsPerConnection: 1   ❸
        maxRetries: 1
        http2MaxRequests: 1   ❹
```

❶ total number of connections

❷ queued requests

❸ requests per connection

❹ max concurrent requests to all hosts

Let's apply this file. Make sure you're in the `chapters/chapter6` folder of the source code for the book:

```
$  kubectl apply -f simple-backend-dr-conn-limit.yaml
destinationrule.networking.istio.io/simple-backend-dr created
```

Let's run the same load test. We set `maxConnections`, `http1MaxPendingRequests` and `http2MaxRequests` to a value of `1`. We also set some other settings `maxRetries` and `maxRequestsPerConnection` but we won't dig into those here; we covered `maxRetries` in a previous section, and `maxRequestsPerConnection` will be `1` for these HTTP 1.1 examples. Let's see what these settings mean:

- `maxConnections` - the threshold at which we'll report a connection "overflow". The Istio proxy (Envoy) will use enough connections to service requests up to an upper bound which is defined in terms of this setting. In reality we can expect a max # of connections to be one per endpoint/host in the load balancing pool + the value of this setting. Any time we go over this value, Envoy will report it in its metrics.
- `http1MaxPendingRequests` - this setting is the allowable number of requests that are "pending" and don't have a connection to use
- `http2MaxRequests` - this setting is unfortunately misnamed in Istio; under the covers this controls the max number of parallel requests across all endpoints/hosts in a cluster regardless of HTTP2 or HTTP1.1. (follow https://github.com/istio/istio/issues/27473)

Let's run our test again and verify that for these settings, when we send 1 request per second over 1 connection, things work fine:

```
$  fortio load -H "Host: simple-web.istioinaction.io" \
-quiet -jitter -t 30s -c 1 -qps 1 http://localhost/

...
Sockets used: 1 (for perfect keepalive, would be 1)
Jitter: true
Code 200 : 30 (100.0 %)
All done 30 calls (plus 1 warmup) 1027.857 ms avg, 1.0 qps
```

What would happen if we increased the number of connections and request per second to 2? From our load-testing tool, if we did that, we'd basically start sending 1 request per second from two connections. On the Istio proxy side, we would be over our connection limit and we would start queueing up outgoing requests. If we bump up against either the max number of requests (1) or max number of pending requests (1) we could trip the circuit breaker. Let's try it:

```
$  fortio load -H "Host: simple-web.istioinaction.io" \
-quiet -jitter -t 30s -c 2 -qps 2 http://localhost/

...
Sockets used: 27 (for perfect keepalive, would be 2)
Jitter: true
Code 200 : 31 (55.4 %)
Code 500 : 25 (44.6 %)
All done 56 calls (plus 2 warmup) 895.900 ms avg, 1.8 qps
```

Indeed we see requests returned as failed requests (HTTP 5xx) here. How do we know for sure these were affected by circuit breaking and not from upstream failures? To know this information, we will need to enable some more stats collection in the Istio service proxy. By default Istio's service proxy (Envoy) keeps a large amount of statistics for each cluster but Istio trims these down to not overwhelm the stats collection agents (e.g., Prometheus) with a large cardinality of statistics. Let's tell Istio to enable this statistics collection for the `simple-web` service since that's what ends up calling the `simple-backend` service in our service graph.

To extend the stats exposed by Istio, especially for upstream circuit breaking stats, we use the annotation `sidecar.istio.io/statsInclusionPrefixes`. In our `simple-web` Kubernetes Deployment, we add this annotation:

```
  template:
    metadata:
      annotations:
        sidecar.istio.io/statsInclusionPrefixes: \
"cluster.outbound|80||simple-backend.istioinaction.svc.cluster.local"
      labels:
        app: simple-web
```

Here we add additional stats that follow the **cluster.<name>** format here. You can see the entire Deployment description and even deploy it by applying the `simple-web-stats-incl.yaml` file:

```
$  kubectl apply -f simple-web-stats-incl.yaml
deployment.apps/simple-web configured
```

Let's make sure we're starting from a known state with regard to stats by resetting all the stats

for the Istio proxy in the `simple-web` service:

```
$  kubectl exec -it deploy/simple-web -c istio-proxy \
-- curl -X POST localhost:15000/reset_counters

OK
```

Now if we generate load again, we should see similar results and go inspect the stats to determine whether circuit breaking kicked in:

```
$  fortio load -H "Host: simple-web.istioinaction.io" \
-quiet -jitter -t 30s -c 2 -qps 2 http://localhost/

...
Sockets used: 25 (for perfect keepalive, would be 2)
Jitter: true
Code 200 : 31 (57.4 %)
Code 500 : 23 (42.6 %)
All done 54 calls (plus 2 warmup) 1020.465 ms avg, 1.7 qps
```

Here we see that `23` calls failed. We believe these failed because of our circuit-breaking settings, but we will verify that by looking at the stats from the Istio proxy. Let's run the following query:

```
$  kubectl exec -it deploy/simple-web -c istio-proxy\
-- curl localhost:15000/stats | grep simple-backend | grep overflow

<omitted>.upstream_cx_overflow: 59
<omitted>.upstream_cx_pool_overflow: 0
<omitted>.upstream_rq_pending_overflow: 23
<omitted>.upstream_rq_retry_overflow: 0
```

We've omitted the cluster name here for readability. The stats we're most interested are the `upstream_cx_overflow` and `upstream_rq_pending_overflow` stats which indicate that there were enough connections and requests that went over our specified thresholds (either too many requests in parallel or too many queued up) and tripped the circuit breaker. We can see there were `23` of those requests which exactly matches how many we saw in our load test that did not complete successfully. Note that we don't get any errors bubble up because of the connection overflow, but it's important to know that when connections overflow, there will be more pressure put on the existing connections. This results in the pending queue growing which eventually trips the circuit breaker. The "fail-fast" behavior we see comes from those pending or parallel requests going above the circuit-breaking threasholds.

What if we increased our `http2MaxRequests` field to account for more requests happening in parallel? Let's raise that value to `2`, reset our counters and re-run our load test:

```
$  kubectl patch destinationrule simple-backend-dr --type merge \
--patch \
'{"spec": {"trafficPolicy": {"connectionPool": {"http": {"http2MaxRequests": 2}}}}}'

$  kubectl exec -it deploy/simple-web -c istio-proxy \
-- curl -X POST localhost:15000/reset_counters

$  fortio load -H "Host: simple-web.istioinaction.io" \
-quiet -jitter -t 30s -c 2 -qps 2 http://localhost/
```

```
...
Sockets used: 4 (for perfect keepalive, would be 2)
Jitter: true
Code 200 : 32 (94.1 %)
Code 500 : 2 (5.9 %)
All done 34 calls (plus 2 warmup) 1786.089 ms avg, 1.1 qps
```

We see that fewer requests got blocked by circuit breaking:

```
$  kubectl exec -it deploy/simple-web -c istio-proxy \
-- curl localhost:15000/stats | grep simple-backend | grep overflow

<omitted>.upstream_cx_overflow: 32
<omitted>.upstream_cx_pool_overflow: 0
<omitted>.upstream_rq_pending_overflow: 2
<omitted>.upstream_rq_retry_overflow: 0
```

What likely happened here is some requests tripped the pending queue circuit breaker. Let's increase the pending queue depth to `2` and re-run:

```
$  kubectl patch destinationrule simple-backend-dr --type merge \
--patch \
'{"spec": {"trafficPolicy": {"connectionPool": {"http": {"http1MaxPendingRequests": 2}}}}}'

$  kubectl exec -it deploy/simple-web -c istio-proxy \
-- curl -X POST localhost:15000/reset_counters

$  fortio load -H "Host: simple-web.istioinaction.io" \
-quiet -jitter -t 30s -c 2 -qps 2 http://localhost/

...
Sockets used: 4 (for perfect keepalive, would be 2)
Jitter: true
Code 200 : 32 (94.1 %)
Code 500 : 2 (5.9 %)
All done 34 calls (plus 2 warmup) 1786.089 ms avg, 1.1 qps
```

We see that with these limits, we successfully complete our load test.

So we can see that when circuit breaking occurs, we can use stats to determine what happened. But what about at runtime? In our example, `simple-web` calls `simple-backend` but if the request fails because of circuit breaking, how does the `simple-web` know that and discern those from application or network failures?

When a request fails for tripping a circuit breaking threshold, Istio's service proxy will add a `X-Envoy-Overloaded` header. One way to test this is to set the connection limits back to their most stringent settings (with 1 for connections, pending requests, and max requests) and try run the load test again. If you also issue a single `curl` command while the load test is running, there's a high chance it will fail because of circuit breaking. When using `curl` you can see the actual response from the simple service implementations:

```
curl -v -H "Host: simple-web.istioinaction.io"  http://localhost/

{
  "name": "simple-web",
  "uri": "/",
```

```
  "type": "HTTP",
  "ip_addresses": [
    "10.1.0.101"
  ],
  "start_time": "2020-09-22T20:01:44.949194",
  "end_time": "2020-09-22T20:01:44.951374",
  "duration": "2.179963ms",
  "body": "Hello from simple-web!!!",
  "upstream_calls": [
    {
      "uri": "http://simple-backend:80/",
      "headers": {
        "Content-Length": "81",
        "Content-Type": "text/plain",
        "Date": "Tue, 22 Sep 2020 20:01:44 GMT",
        "Server": "envoy",
        "X-Envoy-Overloaded": "true"   (1)
      },
      "code": 503,
      "error": "Error processing upstream request: http://simple-backend:80//,
          expected code 200, got 503"
    }
  ],
  "code": 500
}
```

(1) header indication

In general, you should write application code in such a way that the network can fail. If your application code watches for this header, it can make decisions like fallback strategies for its response for its calling client.

6.5.2 Guarding against unhealthy services with outlier detection

In the previous section we saw how Istio can limit requests to services that are misbehaving when they introduce unexpected latency. In this section, we'll explore Istio's approach to removing certain hosts of a service that are misbehaving. Istio leverages Envoy's outlier detection functionality for this. We saw outlier detection in the previous section on locality load balancing and we'll take a closer look here.

To get started with this, let's set everything back to a known-working state:

```
$  kubectl apply -f simple-backend.yaml
$  kubectl delete destinationrule --all
```

Note we are staying with the `simple-web` deployment that has the extended statistics about the `simple-backend` cluster. If you're not sure whether you're in that state (from the previous section on connection pooling), you can be sure by deploying that version of `simple-web`:

```
$  kubectl apply -f simple-web-stats-incl.yaml
```

To explore the behavior, we will also disable Istio's default retry mechanisms. Retry and outlier detection go well together, but we'll try to isolate the outlier detection functionality for these examples. We will add back retry at the end of these examples to see how they complement each

other.

```
$  kubectl apply -f simple-service-disable-retry.yaml
```

Lastly, before we run our tests, let's introduce failure from the `simple-backend` service. In this case, we'll fail with HTTP 500 on 75% of the calls to the `simple-backend-1` endpoint:

```
$  kubectl apply -f simple-backend-periodic-failure-500.yaml
```

Now we are set up, let's run our load test. We turned off retry and we introduced periodic failures so we do expect some of the requests from the load test to fail:

```
$  fortio load -H "Host: simple-web.istioinaction.io" \
-allow-initial-errors -quiet -jitter -t 30s -c 10 -qps 20 http://localhost/

...
Sockets used: 197 (for perfect keepalive, would be 10)
Jitter: true
Code 200 : 412 (68.7 %)
Code 500 : 188 (31.3 %)
All done 600 calls (plus 10 warmup) 189.855 ms avg, 19.9 qps
```

Here we see some percentage of calls did indeed fail. That's okay, we expected that because we made `simple-backend-1` endpoints return failures. If we are sending requests to a service that is failing regularly and the other endpoints that make up that service do not, maybe it's overloaded or somehow degraded and it's best that we stop sending traffic to it for some period of time. Let's configure outlier detection to do exactly that:

```
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: simple-backend-dr
spec:
  host: simple-backend.istioinaction.svc.cluster.local
  trafficPolicy:
    outlierDetection:
      consecutive5xxErrors: 1
      interval: 5s
      baseEjectionTime: 5s
      maxEjectionPercent: 100
```

In this DestinationRule we configure `consecutive5xxErrors` of `1` which means the outlier detection will trip after only one bad request. This might be good for our example here, but you may want to target something more realistic for your environment. The `interval` setting specifies how often the Istio service proxy will check on the hosts and make a determination of whether to eject an endpoint based on the `consecutive5xxErrors` setting. If a service endpoint is ejected, it will be ejected for **n** * `baseEjectionTime` where **n** is the number of times that particular endpoint has been ejected. After the time has elapsed, the endpoint will be added back to the load-balancing pool. Lastly, we can control how many of the hosts in the load balancing

pool are eligible for ejection. In this particular configuration, we're willing to eject 100% of the hosts. This would be analogous to a circuit being tripped open: no requests will pass through when all the hosts are misbehaving.

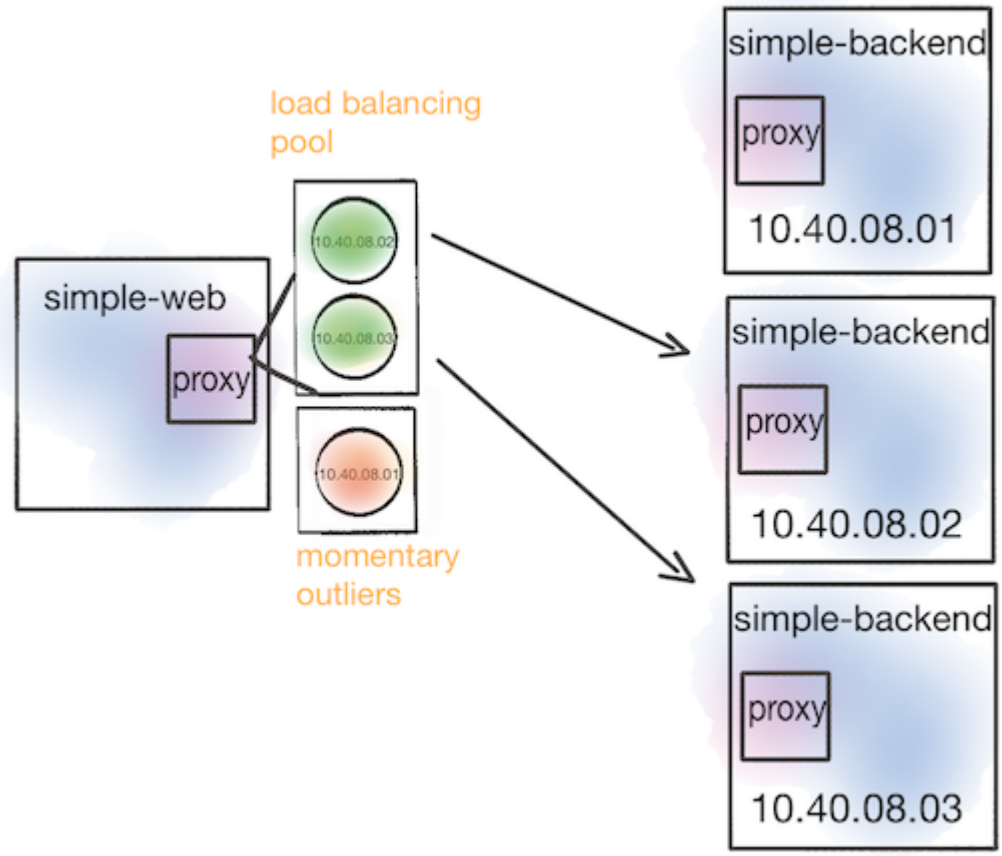


Figure 6.16 Eject endpoints that are misbehaving for a period of time

Let's enable the outlier detection and re-run our tests:

```
$  kubectl apply -f simple-backend-dr-outlier-5s.yaml
destinationrule.networking.istio.io/simple-backend-dr created

$  fortio load -H "Host: simple-web.istioinaction.io" \
-allow-initial-errors -quiet -jitter -t 30s -c 10 -qps 20 http://localhost/

...
Sockets used: 22 (for perfect keepalive, would be 10)
Jitter: true
Code 200 : 589 (98.2 %)
Code 500 : 11 (1.8 %)
All done 600 calls (plus 10 warmup) 250.173 ms avg, 19.7 qps
```

After running these tests, we see our error rate reduced dramatically. This is because the misbehaving endpoint was ejected for a period of time. However, we still have 11 failed calls. To prove these errors were caused by the misbehaving endpoints, we can check the statistics:

```
$  kubectl exec -it deploy/simple-web -c istio-proxy -- \
curl localhost:15000/stats | grep simple-backend | grep outlier

<omitted>.outlier_detection.ejections_active: 0
```

```
<omitted>.outlier_detection.ejections_consecutive_5xx: 3
<omitted>.outlier_detection.ejections_detected_consecutive_5xx: 3
<omitted>.outlier_detection.ejections_detected_consecutive_gateway_failure: 0
<omitted>.outlier_detection.ejections_detected_consecutive_local_origin_failure: 0
<omitted>.outlier_detection.ejections_detected_failure_percentage: 0
<omitted>.outlier_detection.ejections_detected_local_origin_failure_percentage: 0
<omitted>.outlier_detection.ejections_detected_local_origin_success_rate: 0
<omitted>.outlier_detection.ejections_detected_success_rate: 0
<omitted>.outlier_detection.ejections_enforced_consecutive_5xx: 3
<omitted>.outlier_detection.ejections_enforced_consecutive_gateway_failure: 0
<omitted>.outlier_detection.ejections_enforced_consecutive_local_origin_failure: 0
<omitted>.outlier_detection.ejections_enforced_failure_percentage: 0
<omitted>.outlier_detection.ejections_enforced_local_origin_failure_percentage: 0
<omitted>.outlier_detection.ejections_enforced_local_origin_success_rate: 0
<omitted>.outlier_detection.ejections_enforced_success_rate: 0
<omitted>.outlier_detection.ejections_enforced_total: 3
<omitted>.outlier_detection.ejections_overflow: 0
<omitted>.outlier_detection.ejections_success_rate: 0
<omitted>.outlier_detection.ejections_total: 3
```

We see the `simple-backend-1` host was ejected 3 times. We also see that the number of calls from the previous run that failed was 11. What happened was during the `5s interval` setting, some requests hit this misbehaving host and it wasn't until the outlier detection check happened (after 5s) some of those requests did indeed hit the misbehaving host, hence the errors. What can we do to work around those last few errors?

We can add back the default retry settings (or explicitly set them):

```
$  kubectl delete vs simple-backend-vs
virtualservice.networking.istio.io "simple-backend-vs" deleted

$  kubectl apply -f simple-web-gateway.yaml
gateway.networking.istio.io/simple-web-gateway unchanged
virtualservice.networking.istio.io/simple-web-vs-for-gateway configured
```

Now try the load test once again and you should see no errors.

6.6 Summary

Up to this chapter, we've seen how we can use Istio's functionality and APIs to change the behavior of the network from the edge using the ingress gateway to intra-cluster communication. However, as we established at the beginning of this chapter, manual intervention to react to unexpected network failures may be nearly impossible in large-scale constantly changing systems.

In this chapter we dug deeply into Istio's various client-side resilience features that allow services to transparently recover from intermittent network issues or topology changes. We saw how smart clients like the Istio service proxy can be armed with information about upstream endpoints, detect their health or behavior on the fly, and make load-balancing decisions, locality decisions, and even enforce timeouts, retry policies and circuit breaking. We also examined how a combination of these features can be applied to build highly resilient microservices applications without altering a line of application code.

In the next chapters we'll layer on to this capabilities by exploring how to observe the behaviors of the network.

7

Observability with Istio: understanding the behavior of your services

This chapter covers:

- Basic request-level metric collection
- Using prometheus to store metrics
- Adding new metrics in Istio to track in prometheus
- Creating dashboards for observing metrics visually
- Distributed tracing instrumentation with Jaeger
- Visualizing the service mesh with Kiali

You may have heard the term *observability* start to creep into the vocabulary of software engineers, operations and site-reliability teams recently. These teams have to deal with the near exponential increase in complexity when operationalizing a microservices-style architecture on cloud infrastructure. When we start to deploy our application as 10s or 100s of services (or more) per application, we increase the number of moving pieces, increase the amount of reliance on the network for things to succeed, and also increase the number of things that can go wrong. As our systems start to go down this path, and as they get bigger, there is a higher probability that at least some part of the system is always running in a degraded state. Not only must we build our applications to be more reliable and resilient (some of which can be implemented with Istio; see the previous chapter), we must also improve our tooling and instrumentations to be able to understand what's really happening when they are running. If we can confidently comprehend what's happening with our services and infrastructure at runtime, we can learn to detect failures and dive deep into debugging when we observe something unexpected. This effort will help us improve our Mean Time To Recovery (MTTR), an important measure in high-performing teams and their impact on the business. In this chapter, we take a look at what Istio can do to help us understand what's happening to our services at run time by collecting and propagating important signals.

7.1 What is observability?

Observability is a characteristic of a system that is measured by the level to which you can understand and reason about a system's internal state just by looking at its external signals and characteristics. Observability is important to be able to implement controls for a system in which we can change its runtime behavior. This definition is based on the study of control theory first introduced in a paper from "On the General Theory of Control Systems" authored by Rudolf E. Kálmán in 1960. In more practical terms, we value stability in our systems and we need to understand when things are going well in order to discern when things are going wrong and implement the right levels of automated and manual control to maintain this dynamic.

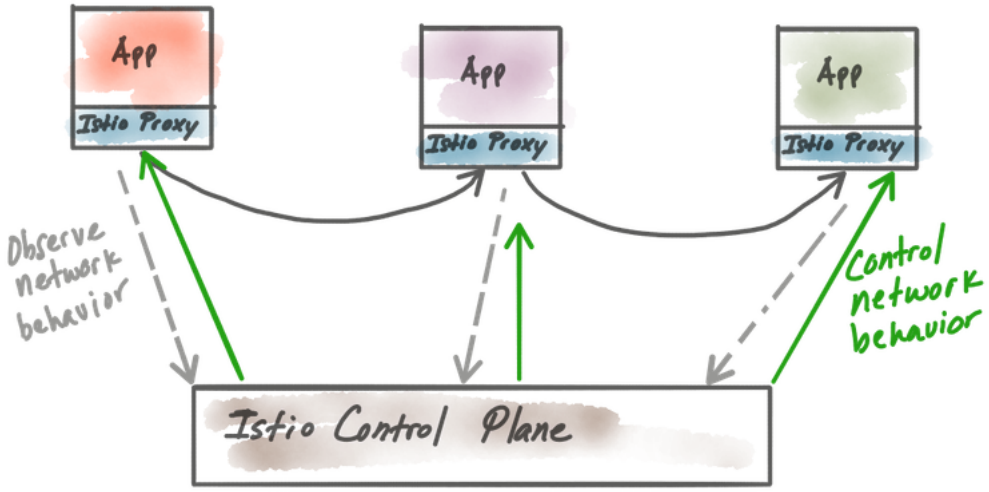


Figure 7.1 Istio is in position to implement controls and observations

In Figure 7.1, we see that Istio's data plane is in a position to affect the behavior of a request through the system. Istio can help implement controls like traffic shifting, resilience, policy enforcement, and more, but to know what controls to engage and when, we need to understand what's happening in the system. Since most of Istio's control capabilities are implemented at the network level for application requests, we shouldn't be surprised to find that Istio's ability to collect signals to inform our observations are also at this level. This does not mean that leveraging Istio to help with observability is the only thing you need to get observability into your system. Observability is a characteristic of a system involving various levels, not an off-the-shelf solution, and must incorporate a combination of application instrumentation, network instrumentation, signal collection infrastructure and databases, as well as a way to sift through the vast amount of data to piece together a picture when unpredictable things happen. Istio helps with one part of observability: application-level network instrumentation.

7.1.1 Observability vs Monitoring

The term observability has brought a level of confusion to the market in terms of a practice with which we may already be familiar: monitoring. Monitoring is the practice of collecting signals, telemetry, traces, etc and aggregating them, and matching them against some pre-defined criteria of system states we should carefully watch. When we find that one of our signals has crossed a threshold and may be heading toward a known bad state, we take action to remedy the system. For example, our operations teams can collect information about disk usage for a particular database installation. If those metrics show the disk usage approaching its capacity, we can fire an alert to trigger some kind of remediation like adding more storage to the disks.

Monitoring is a subset of observability. With monitoring we are specifically collecting and aggregating metrics to watch for known undesirable states and then alert on them. Observability on the other hand supposes up front that our systems are highly unpredictable and we cannot know all of the possible failure modes up front. We need to collect much more data, even high-cardinality data like userIDs, requestIDs, source IPs, etc where the entire set could be exponentially large, and use tools to quickly explore and ask questions about the data. For example, if a particular user — say user John Doe with userID 400000021 — goes to pay for the items in their cart and experiences a 10s delay choosing a payment option. All of the pre-defined metric thresholds (disk usage, queue depth, machine health, etc) might be at acceptable levels, but John Doe is highly irritated at this user experience. If we have designed with observability in mind, we can sift through the many signals along the request path such as service latency, infrastructure hops, which message queues or databases were involved, and even digging into the level of which database queries ran for this particular user and request.

With observability, we need fine-grained data. We should not aggregate data without the ability to dig deeper into the raw events as the source of truth. If things are going wrong, we cannot adequately debug them without the context and awareness we get with high-fidelity instrumentation data as well as the tools to dig deeper into and explore the data.

7.1.2 How Istio helps with observability

Istio is in a unique position to help build an observable system. Istio's data plane proxy, Envoy, sits in the request path and can observe important qualities of the system and report them to backed systems. Through the Envoy service proxy, Istio is able to capture important signals related to request handling and service interaction such as number of requests per second, how long requests are taking (broken out into percentiles), how many failed requests we've experienced and so on. Istio can also help you dynamically add new tracing signals to your system to capture new information you hadn't though about ahead of time.

One aspect of understanding a distributed system is tracing requests through the system to understand what services and components are involved in a request flow and how long each node

in that graph is taking to process the request. Distributed tracing, first introduced by the Google Dapper Paper, involves annotating requests with correlation IDs that represent service-to-service calls and trace IDs that represent a specific request through a graph of service-to-service calls. Istio's data plane can add (and importantly, remove them when they are unrecognized or come from external entities) these kinds of metadata from the requests as they pass through the data plane

Istio can assist with this kind of distributed tracing effort with OpenTracing support. Lastly, Istio comes with some out of the box integrations with tools like Prometheus, Grafana, and Kiali which can help you visualize and explore the state of the service mesh and the services it knows about. The rest of this chapter will walk through using Istio's observability features to collect metrics, distributed traces, and visualize them.

7.2 Collecting metrics from Istio data plane

Istio's data plane, Envoy keeps a bevy of connection, request, and runtime metrics which we can use to form a picture of a service's network and communication health. Let's first deploy a subset of our sample application and explore its components to understand where those metrics come from and how to access them. We'll explore Istio's capabilities of building an observable system by collecting telemetry around application networking and bringing those back to an area we can explore and visualize.

First, let's assume that we have Istio deployed (see Chapter XX for doing that) but that we don't have any other application components deployed. If you are continuing from previous chapters, you may have to clean up any left-behind deployments, services, gateways and VirtualServices.

To deploy the subset of our application for the purposes of this section, let's run the following command from the root of the Istio in Action source code:

```
$  make deploy-apigateway-with-catalog

service/catalog created
    deployment.extensions/catalog created
service/apigateway created
    deployment.extensions/apigateway created
    virtualservice.networking.istio.io/apigateway-vs-from-gw created
    gateway.networking.istio.io/coolstore-gateway created
```

Now we can run the following command to verify we can reach our services and they return correctly:

```
curl  -H "Host: apiserver.istioinaction.io"  \
  $(make ingress-url)/api/catalog
```

The first thing we should discover is the metrics kept by a service's sidecar Istio proxy. If we do a listing of the pods we have deployed, and that have Istio's sidecar proxy deployed alongside, we see both the `apigateway` and `catalog` services.

```
$  kubectl get pod
NAME                          READY   STATUS    RESTARTS   AGE
apigateway-67bd5dfd77-g7gcf   2/2     Running   0          20m
catalog-c89594fb9-hm47h       2/2     Running   0          20m
```

Let's execute a query to view the stats from the `apigateway` pod:

```
$  kubectl exec -it $(make apigateway-pod) \
  -c istio-proxy -- curl localhost:15000/stats
```

Wow! That's a lot of information kept by the sidecar proxy.

Let's cut down some of the output to only see stats related to incoming traffic to our `apigateway` service:

```
$  kubectl exec -it $(make apigateway-pod) -c istio-proxy \
  -- curl localhost:15000/stats | grep cluster.local | grep 8080

apigateway.istioinaction.svc.cluster.local.bind_errors: 0
apigateway.istioinaction.svc.cluster.local.internal.upstream_rq_200: 3
apigateway.istioinaction.svc.cluster.local.internal.upstream_rq_2xx: 3
apigateway.istioinaction.svc.cluster.local.internal.upstream_rq_completed: 3
apigateway.istioinaction.svc.cluster.local.lb_healthy_panic: 0
apigateway.istioinaction.svc.cluster.local.lb_local_cluster_not_ok: 0
apigateway.istioinaction.svc.cluster.local.lb_recalculate_zone_structures: 0
apigateway.istioinaction.svc.cluster.local.lb_subsets_active: 0
apigateway.istioinaction.svc.cluster.local.lb_subsets_created: 0
apigateway.istioinaction.svc.cluster.local.lb_subsets_fallback: 0
apigateway.istioinaction.svc.cluster.local.lb_subsets_removed: 0
apigateway.istioinaction.svc.cluster.local.lb_subsets_selected: 0
apigateway.istioinaction.svc.cluster.local.lb_zone_cluster_too_small: 0
apigateway.istioinaction.svc.cluster.local.lb_zone_no_capacity_left: 0
apigateway.istioinaction.svc.cluster.local.lb_zone_number_differs: 0
apigateway.istioinaction.svc.cluster.local.lb_zone_routing_all_directly: 0
apigateway.istioinaction.svc.cluster.local.lb_zone_routing_cross_zone: 0
apigateway.istioinaction.svc.cluster.local.lb_zone_routing_sampled: 0
apigateway.istioinaction.svc.cluster.local.max_host_weight: 0
apigateway.istioinaction.svc.cluster.local.membership_change: 1
apigateway.istioinaction.svc.cluster.local.membership_healthy: 1
apigateway.istioinaction.svc.cluster.local.membership_total: 1
```

Note, the output has been trimmed to make it better viewable in the listing above. A couple things to notice is that we've seen `3` requests completed to this service with all of the returning a `2xx` or `200` in this case. We can also see detailed statistics about the loadbalancer, the health of it, and any zone-aware information. The Istio proxy actually keeps this level of detail for any traffic coming into the service and for any communication leaving the service. Let's explore what stats are kept for calls going to the `catalog` service:

```
kubectl exec -it $(make catalog-pod) -c istio-proxy \
  -- curl localhost:15000/stats | grep catalog | grep 8080
```

Explore this listing to get an idea of what stats are kept for all of the outgoing calls to the `catalog` service. The proxies do a good job of collecting metrics, but we don't want to have to go to each service instance and each proxy to retrieve the metrics. The Istio service proxy can stream these metrics to a metric-collection system such as statsd. StatsD is a metric collection

system open-sourced by Etsy to format, collect, and distribute statistics like counters, guages, and timers to backend monitoring, alerting, or visualization tools. Let's take a look at how to configure the Istio service proxy to send metrics to statsd.

7.2.1 Pushing Istio metrics into statsD

Before we configure the Istio service proxy to send metrics to statsd, we need to set up a statsd server. Setting up and operationalizing a metric-collection system is beyond the scope of this book, however, we'll install a statsd daemon and configure Istio to send metrics to it.

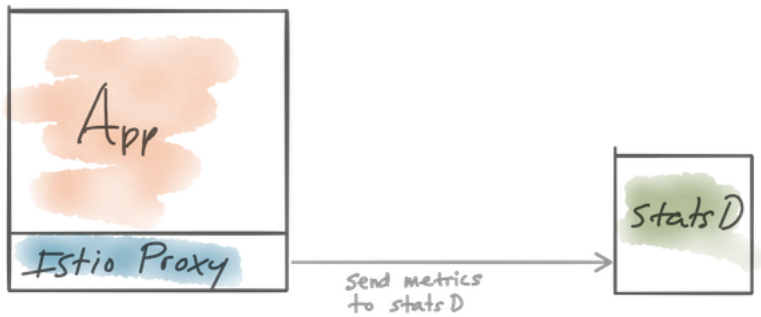


Figure 7.2 Istio service proxy sends stats to statsD

Let's start by installing a simple statsd deamon:

```
kubectl apply -f chapters/chapter7/statsd

deployment.apps/statsd created                    service/statsd created
```

We can try to query the stats of the statsd server to verify everything is up and running. First let's grab the name of the new `statsd` server:

```
$  kubectl get pod
NAME                          READY   STATUS    RESTARTS   AGE
apigateway-58fdc5bbd4-8tlpm   2/2     Running   0          8m
catalog-c89594fb9-kqd2h       2/2     Running   0          15m
statsd-fd6c7477d-fw9rl        1/1     Running   0          10m
```

From the above, we can see the pod is named `statsd-fd6c7477d-fw9rl`. Let's `exec` into it to run some comamnds that query it's Admin interface:

```
$  kubectl exec -it $(make statsd-pod) sh

/app/statsd # echo "counters" | nc 127.0.0.1 8126

{ 'statsd.bad_lines_seen': 0,
  'statsd.packets_received': 0,
  'statsd.metrics_received': 0 }
END

/app/statsd #
```

We can see some very simple counters from the statsd interface. That's great, but let's add some new ones. We'll now configure Istio to send statsd metrics to this new statsd server. Exit out of

`kubectl exec` with a `^c` and let's edit our `apigateway` deployment.

To edit the `apigateway` deployment, we will edit it in place. What we're going to edit is the configuration passed into the istio proxy (recall, this configuration was automatically injected any time we do a `istioctl kube-inject -f <filename>`). To do so, let's do a `kubectl edit` and edit the deployment:

```
$  kubectl edit deploy apigateway
```

NOTE For this to work from the CLI, you will need to have your environment variable `EDITOR` set to an appropriate editor. See kubernetes.io/docs/reference/kubectl/cheatsheet/#editing-resources for more.

When we're in our editor with the `apigateway` deployment yaml, scroll down to the section that configures the `istio-proxy`, specifically the `args` section:

```
- args:
  - proxy
  - sidecar
  - --configPath
  - /etc/istio/proxy
  - --binaryPath
  - /usr/local/bin/envoy
  - --serviceCluster
  - apigateway
  - --drainDuration
  - 45s
  - --parentShutdownDuration
  - 1m0s
  - --discoveryAddress
  - istio-pilot.istio-system:15007
  - --discoveryRefreshDelay
  - 1s
  - --zipkinAddress
  - zipkin.istio-system:9411
  - --connectTimeout
  - 10s
  - --proxyAdminPort
  - "15000"
  - --controlPlaneAuthPolicy
  - NONE
```

We need to add a flag to this configuration to let the `istio-proxy` know to send stats to a statsd server. Let's add the following flag to that list:

```
- args:
  - proxy
  - sidecar
  - --statsdUdpAddress
  - statsd:8125
```

Now save and exit your editor. You should observe a new `apigateway` pod created and deployed

```
kubectl get pod -w
NAME                           READY   STATUS            RESTARTS   AGE
apigateway-66c7cbff68-z7jp9   0/2     PodInitializing   0          8h
apigateway-67bd5dfd77-vpvv5   0/2     Terminating       15         10h
catalog-c89594fb9-kqd2h       2/2     Running           15         11h
statsd-fd6c7477d-5gh7b        1/1     Running           0          10h
apigateway-66c7cbff68-z7jp9   1/2   Running   0     8h
apigateway-67bd5dfd77-vpvv5   0/2   Terminating   15    10h
apigateway-67bd5dfd77-vpvv5   0/2   Terminating   15    10h
apigateway-66c7cbff68-z7jp9   2/2   Running   0     8h
```

Now if we `kubectl exec` back into our `statsd` pod, we should see new metrics added. These counters and guages were added by our Istio service proxy that's deployed as a sidecar with the `apigateway` service.

```
$  kubectl exec -it $(make statsd-pod) sh

/app/statsd # echo "counters" | nc 127.0.0.1 8126

{ 'statsd.bad_lines_seen': 0,
  'statsd.packets_received': 493,
  'statsd.metrics_received': 493,
  'envoy.http.0.0.0.0_15004.rds.15004.config_reload': 0,
  'envoy.http.0.0.0.0_15004.rds.15004.update_attempt': 0,
  'envoy.http.0.0.0.0_15004.rds.15004.update_success': 0,
  'envoy.cluster.outbound80tracing.istio-system.svc.cluster.local.update_empty': 0,
  'envoy.cluster.outbound80tracing.istio-system.svc.cluster.local.update_attempt': 0,
  'envoy.cluster.outbound80tracing.istio-system.svc.cluster.local.membership_change': 0,
  'envoy.cluster.outbound80tracing.istio-system.svc.cluster.local.update_success': 0,
  'envoy.cluster.outbound15031istio-ingressgateway.istio-system.svc.cluster.local.update_empty': 0,
  'envoy.cluster.outbound15031istio-ingressgateway.istio-system.svc.cluster.local.update_attempt': 0,
  'envoy.cluster.outbound15031istio-ingressgateway.istio-system.svc.cluster.local.membership_change': 0,
  'envoy.cluster.outbound15031istio-ingressgateway.istio-system.svc.cluster.local.update_success': 0,
  'envoy.cluster.outbound42422istio-telemetry.istio-system.svc.cluster.local.membership_change': 0,
  'envoy.cluster.outbound42422istio-telemetry.istio-system.svc.cluster.local.update_empty': 0,
  'envoy.cluster.outbound42422istio-telemetry.istio-system.svc.cluster.local.update_success': 0,
  'envoy.cluster.outbound42422istio-telemetry.istio-system.svc.cluster.local.update_attempt': 0,
  'envoy.cluster.outbound853istio-ingressgateway.istio-system.svc.cluster.local.update_attempt': 0,
  'envoy.cluster.outbound853istio-ingressgateway.istio-system.svc.cluster.local.membership_change': 0,
  'envoy.cluster.outbound853istio-ingressgateway.istio-system.svc.cluster.local.update_empty': 0,
  'envoy.cluster.outbound853istio-ingressgateway.istio-system.svc.cluster.local.update_success': 0,
  'envoy.cluster.outbound9093istio-telemetry.istio-system.svc.cluster.local.update_empty': 0,
  'envoy.cluster.outbound9093istio-telemetry.istio-system.svc.cluster.local.update_success': 0,
  'envoy.cluster.outbound9093istio-telemetry.istio-system.svc.cluster.local.update_attempt': 0,
  'envoy.cluster.outbound9093istio-telemetry.istio-system.svc.cluster.local.membership_change': 0,
  'envoy.cluster.outbound8060istio-ingressgateway.istio-system.svc.cluster.local.update_success': 0,
  'envoy.cluster.outbound8060istio-ingressgateway.istio-system.svc.cluster.local.update_attempt': 0,
  'envoy.cluster.outbound8060istio-ingressgateway.istio-system.svc.cluster.local.update_empty': 0,
  'envoy.cluster.outbound8060istio-ingressgateway.istio-system.svc.cluster.local.membership_change': 0,
  'envoy.http.0.0.0.0_8080.rds.8080.config_reload': 0,
  'envoy.http.0.0.0.0_8080.rds.8080.update_attempt': 0,
  'envoy.http.0.0.0.0_8080.rds.8080.update_success': 0,
...

/app/statsd #
```

Now we have our Istio service proxy for the `apigateway` service transmitting statistics to a statsd server. You can expand on this in your environment to use statsd to collect metrics from all of your sidecar service proxies. To change this configuration for any new deployments with

Istio (that use either automatic sidecar injection of the `istioctl kube-inject` approach) we need to update the `istio-sidecar-injector` ConfigMap in the `istio-system` namespace. To do that, let's use `kubectl edit` again:

```
$  kubectl edit cm/istio-sidecar-injector -n istio-system
```

This file contains the template for injecting the sidecar proxy. Edit it to add the `--statsdUdpAddress` parameter like we did earlier to the `apigateway` Kubernetes `deployment` .

To quickly recap, to add statsD support, we configure the Istio sidecar proxy to stream stats to a statsD collector. Istio's service proxy, Envoy, keeps a large table of information it collects and it has native statsD integration. In a default Istio installation, statsD is not used but it can be enabled pretty simply by following the steps above.

Some folks prefer to use, or are already using, Prometheus as their metrics collecting engine. In the next section, we'll take a look at using Prometheus to gather metrics from our service mesh and how it differs from systems like StatsD.

7.2.2 Pulling Istio Metrics into Prometheus

Prometheus is a metrics-collection engine and set of related monitoring and alerting tools that originated at SoundCloud and was loosely based on Google's internal monitoring system Borgmon (in a similar way that Kubernetes was based on Borg). Prometheus is slighly different from other telemetry or metrics collection systems in that it "pulls" metrics from its targets rather than listens for the targets to send their metrics to it. For example, in the previous section, we set up the Istio service proxy to push metrics to statsd. With Prometheus, we expect our applications or the Istio service proxy to expose an endpoint with the latest metrics from which Prometheus can the pull or scrape the metrics. In this book we won't get into the discussion about whether pull or push metric collection is better but we'll acknowledge both exist and an organization may choose one over the other or both. Please see this podcast with Brian Brazil to hear more about Pormetheus' approach to pull-based metrics and how it differs from "push" based systems.

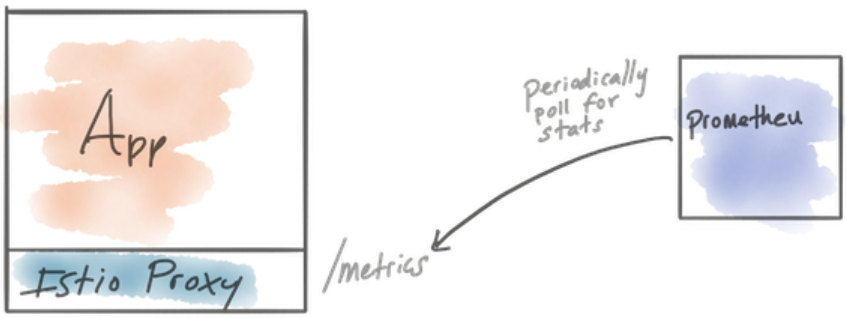


Figure 7.3 Prometheus scraping Istio service proxy for metrics

One of the benefits of using Prometheus is we can, using a simple HTTP client or web browser,

examine our metrics endpoints. Let's use a simple curl command to scrape an HTTP endpoint that exposes our Istio service proxy metrics in Prometheus format:

First, we'll list our pods and pick any of the services running. For this example, we'll just use the `apigateway` pod:

```
$  kubectl get pod
NAME                           READY   STATUS    RESTARTS   AGE
apigateway-66c7cbff68-z7jp9    2/2     Running   3          5d
catalog-c89594fb9-kqd2h        2/2     Running   17         5d
statsd-fd6c7477d-5gh7b         1/1     Running   1          5d
```

Next, we'll issue a `curl` command to the port where our Istio service proxy is exposing the Prometheus metrics, port `15090`. Note, we can look up exactly which port on which our Istio service proxy is exposing Prometheus metrics with this command:

```
$  kubectl get pod $(make apigateway-pod)  \
-o jsonpath='{.spec.containers[?(@.name=="istio-proxy")]\
.ports[?(@.name=="http-envoy-prom")].containerPort}'
```

Now let's curl the port:

```
kubectl exec -it $(make apigateway-pod) curl localhost:15090/stats/prometheus

...
envoy_cluster_version{cluster_name="outbound|9093||istio-galley.istio-system.svc.cluster.local"}
    16933522936099040193
envoy_cluster_membership_healthy{cluster_name="outbound|9093||
    istio-galley.istio-system.svc.cluster.local"} 1
envoy_cluster_max_host_weight{cluster_name="outbound|9093||
    istio-galley.istio-system.svc.cluster.local"} 1
envoy_cluster_upstream_rq_pending_active{cluster_name="outbound|9093||
    istio-galley.istio-system.svc.cluster.local"} 0
envoy_cluster_lb_subsets_active{cluster_name="outbound|9093||
    istio-galley.istio-system.svc.cluster.local"} 0
envoy_cluster_membership_total{cluster_name="outbound|9093||
    istio-galley.istio-system.svc.cluster.local"} 1
envoy_cluster_upstream_cx_active{cluster_name="outbound|9093||
    istio-galley.istio-system.svc.cluster.local"} 0
envoy_cluster_upstream_rq_active{cluster_name="outbound|9093||
    istio-galley.istio-system.svc.cluster.local"} 0
envoy_cluster_upstream_cx_rx_bytes_buffered{cluster_name="outbound|9093||
    istio-galley.istio-system.svc.cluster.local"} 0
envoy_cluster_upstream_cx_tx_bytes_buffered{cluster_name="outbound|9093||
    istio-galley.istio-system.svc.cluster.local"} 0
envoy_listener_downstream_cx_active{listener_address="10.109.82.141_443"} 0
...
```

We should see a list of metrics formatted the way Prometheus expects. All of our applications that have the Istio service project injected will automatically expose these Prometheus metrics. All we have to do is set up a Prometheus server to scrape them.

Prometheus is nice in that we can quickly spin up a Prometheus server and begin scraping metrics even if we have other Prometheus servers already scraping from the metrics endpoints on individual targets (pods in this case). In fact, this is how Prometheus can be configured to be highly available: we can just run multiple Prometheus servers scraping the same targets. In this

case, we'll first configure Prometheus to scrape all of the pods in the `istioinaction` namespace that have the Istio service proxy injected.

Let's create a Prometheus configuration similar to the following:

```
global:
  scrape_interval: 15s
scrape_configs:

# Scrape config for envoy stats
- job_name: 'envoy-stats'
  metrics_path: /stats/prometheus ❶
  kubernetes_sd_configs: ❷
  - role: pod

  relabel_configs:
  - source_labels: [__meta_kubernetes_pod_container_port_name]
    action: keep
    regex: '.*-envoy-prom' ❸
  - source_labels: [__address__, __meta_kubernetes_pod_annotation_prometheus_io_port]
    action: replace
    regex: ([^:]+)(?::\d+)?;(\d+)
    replacement: $1:15090
    target_label: __address__
  - action: labelmap
    regex: __meta_kubernetes_pod_label_(.+)
  - source_labels: [__meta_kubernetes_namespace]
    action: replace
    target_label: namespace
  - source_labels: [__meta_kubernetes_pod_name]
    action: replace
    target_label: pod_name
```

❶ Path to scrape

❷ Kubernetes API

❸ Find Prometheus ports

In this `prometheus.yaml` configuration, we're setting up Prometheus to automatically discover the pods in our Kubernetes cluster and scrape the metrics exposed at the `/stats/prometheus` context path. Prometheus's `kubernetes_sd_configs` configuration tells Prometheus to query the Kubernetes API for certain objects that allow Prometheus to automatically discover certain endpoints dynamically at runtime. We use the `relabel_configs` stanza to figure out the right port number and relabel some of the target's information. Please see Prometheus's documentation on `relabel_configs` for more information and specifics for tuning your specific Prometheus configuration.

When running in Kubernetes, we will want to turn this Prometheus configuration file into a `ConfigMap` so we can mount it into the Prometheus Kubernetes Pod. The `ConfigMap` should look something similar to this:

```
apiVersion: v1
kind: ConfigMap
metadata:
  labels:
```

```
    source: istioinaction
  name: istioinaction-prom
data:
  prometheus.yml: |-
    global:
      scrape_interval: 15s
    scrape_configs:

    # Scrape config for envoy stats
    - job_name: 'envoy-stats'
      metrics_path: /stats/prometheus
      kubernetes_sd_configs:
      - role: pod

      relabel_configs:
      - source_labels: [__meta_kubernetes_pod_container_port_name]
        action: keep
        regex: '.*-envoy-prom'
      - source_labels: [__address__, __meta_kubernetes_pod_annotation_prometheus_io_port]
        action: replace
        regex: ([^:]+)(?::\d+)?;(\d+)
        replacement: $1:15090
        target_label: __address__
      - action: labelmap
        regex: __meta_kubernetes_pod_label_(.+)
      - source_labels: [__meta_kubernetes_namespace]
        action: replace
        target_label: namespace
      - source_labels: [__meta_kubernetes_pod_name]
        action: replace
        target_label: pod_name
```

From the source code that accompanies this book, we've already done that. Let's create the `ConfigMap`:

```
$  kubectl create -f chapters/chapter7/prom/prometheus-configmap.yaml
configmap/istioinaction-prom created
```

Now we configure our Prometheus deployment to mount in the Prometheus configuration map. Here are the relevant snippets from our `prometheus-deployment.yaml`:

```
apiVersion: extensions/v1beta1
kind: Deployment
...
  template:
  ...
    spec:
      containers:
      - args:
        - --storage.tsdb.retention=6h
        - --config.file=/etc/prometheus/prometheus.yml
        image: docker.io/prom/prometheus:v2.3.1
        imagePullPolicy: IfNotPresent
        ...
        volumeMounts:
        - mountPath: /etc/prometheus
          name: config-volume ❷
      ...
      serviceAccount: prometheus
      serviceAccountName: prometheus
      volumes:
      - configMap:
          name: istioinaction-prom ❶
        name: config-volume
```

❶ ConfigMap declared as volume

❷ Mounted into Pod

In the above configuration snippet, you'll notice we are configuring Prometheus to use the `/etc/prometheus/prometheus.yaml` configuration file. We can also see that we are mounting in our Promtheus `ConfigMap` into the `/etc/prometheus` location. Lastly, one thing to keep in mind is Prometheus queries the Kubernetes API for Pods and other information related to what's running in Kubernetes. Therefore, we should configure the correct RBAC permissions so that Prometheus will have the correct permissions to query the Kubernetes API. We attach these RBAC permissions to the `prometheus` service account which we see associated with the above deployment. Please refer to `chapter-files/chapter7/prom/prometheus-deployment.yaml` to see the details on the RBAC configuration.

NOTE **Kubernetes RBAC**

Kubernetes has a sophisticated Role Based Access Control (RBAC) mechanism for limiting access to certain platform resources (Pods, Namespaces, Deployments, etc) including the Kubernetes API. Prometheus must query the Kubernetes API to find the list of endpoints and associated metadata to determine which ones are eligible for metric collection. For more on RBAC in Kubernetes, please see Chapter 12 "Securing the Kubernetes API server" in Marko Luksa's "Kubernetes in Action" www.manning.com/books/kubernetes-in-action

One last thing to consider. Envoy produces **a lot** of metrics. You may wish to prune some of those metrics down that you send to your metrics-collection engine. In the Prometheus configuration file, we can do that use the `metrics_relabel_configs` stanza to do that. For example, to drop certain metric groups that may produce a high volume of distinct metrics, we can drop them from collection like this:

```
metric_relabel_configs:
- source_labels: [ cluster_name ]
  regex: '(outbound|inbound|prometheus_stats).*'
  action: drop
- source_labels: [ tcp_prefix ]
  regex: '(outbound|inbound|prometheus_stats).*'
  action: drop
- source_labels: [ listener_address ]
  regex: '(.+)'
  action: drop
```

Please see the Prometheus documentation on metrics_relabel_configs for more.

Now that we have our Prometheus configuration created, let's create our Prometheus deployment:

```
$  kubectl create -f chapters/chapter7/prom/prometheus-deployment.yaml
```

```
deployment.extensions/prometheus created
serviceaccount/prometheus created
clusterrole.rbac.authorization.k8s.io/prometheus-istioinaction created
clusterrolebinding.rbac.authorization.k8s.io/prometheus-istioinaction created
```

Give it a few moments for the Prometheus pod to come up. When it does, you should see it in the pod listing when you do `kubectl get pod`

```
$  kubectl get pod
NAME                          READY   STATUS    RESTARTS   AGE
apigateway-66c7cbff68-z7jp9   2/2     Running   24         5d
catalog-c89594fb9-kqd2h       2/2     Running   42         6d
prometheus-ffc8f9f4b-sbr8b    1/1     Running   0          1m
statsd-fd6c7477d-5gh7b        1/1     Running   2          6d
```

Let's query our new Prometheus server to verify we're now pulling stats. To do so, let's port-forward it locally like this:

```
$  kubectl port-forward prometheus-ffc8f9f4b-sbr8b 9090:9090
```

Now in our browser, let's navigtate to http://localhost:9090/graph.

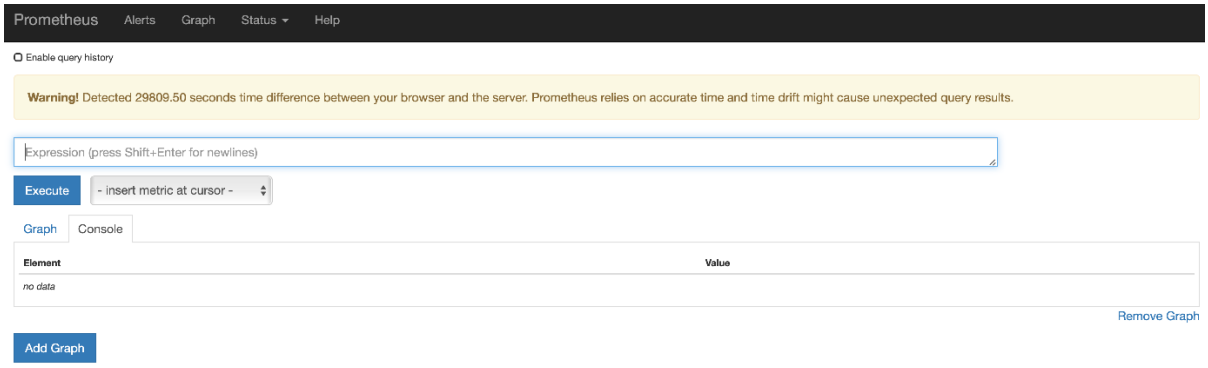


Figure 7.4 A simple out of the box Prometheus expression browser for browsing metrics in Prometheus

If we start typing envoy into the expression browser, you should see an autofill with the metrics we've collected from our Istio service proxies.

If you don't see any metrics pop up, you can try the metric drop down (next to the Execute button) to check whether any show up there. If not, please go back and verify you have the configuration set up correctly for your cluster. Often times incorrect RBAC settings are the culprit here. Please review the permissions you've given to the Prometheus deployment and service account.

Istio does come with a pre-configured Prometheus server (and associated configuration) on our default installation. Please take a look at this configuration file and associated deployment for more information on scraping Kubernetes nodes, the Istio control plane, as well as setting up certificates and certificate authority for scraping services that have Istio mutual TLS enabled. We'll cover how to enable that in Istio in subsequent chapters.

Prometheus is a great way to quickly get started collecting metrics from Istio. Istio has done a lot of the plumbing for you on default installs, but using the information from above you should be able to integrate your own Prometheus server. The important parts are to specify the configuration for Prometheus to find the metrics endpoints (using Kubernetes in this section), and ensuring Prometheus has the correct permissions to do so. After we've collected metrics, let's see how we can visualize them so we can observe what's happening in real time with our services.

7.2.3 Visualize Istio metrics with Grafana

Prometheus at its core is a time-series database and collection toolkit. When we have collected our metrics, we may also wish to display them in dashboards so we can visually see what's happening with our services. Prometheus comes with a simple expression browser to help explore trends and metrics, but Grafana is a popular and powerful open-source graph and visualization tool that works well with Prometheus. In this section, we'll take a look at connecting Grafana to Prometheus and how to query and visualize the metrics we collected. Istio comes with some out of the box dashboards (and pre-configured Prometheus and Grafana integration) that you can leverage.

To get started, let's install Grafana into our cluster. We'll take a stock Grafana Kubernetes Deployment and install it to our cluster:

```
$  kubectl create -f chapters/chapter7/prom/grafana-deployment.yaml
```

You should be able to list the pods and see something similar to this:

```
$ kubectl get pod

NAME                          READY   STATUS    RESTARTS   AGE
apigateway-66c7cbff68-z7jp9   2/2     Running   30         7d
catalog-c89594fb9-kqd2h       2/2     Running   46         7d
grafana-77bfbb4ff5-qw785      1/1     Running   0          1h
prometheus-ffc8f9f4b-sbr8b    1/1     Running   2          1d
statsd-fd6c7477d-5gh7b        1/1     Running   2          6d
```

With Grafana running, let's port forward a local port on our machine to the Grafana pod so we can set up Grafana to integrate with Prometheus and so we can start building dashboards.

```
$ kubectl port-forward deploy/grafana 3000

Forwarding from 127.0.0.1:3000 -> 3000
Forwarding from [::1]:3000 -> 3000
```

Now in your browser, you can go to http://localhost:3000 to see the Grafana instance we set up.

To start building dashboards based on the metrics we collected in Prometheus, we need to set up a Prometheus datasource for Grafana. Navigate to the "Data Sources" tab and lets create a new one pointing to our Prometheus.

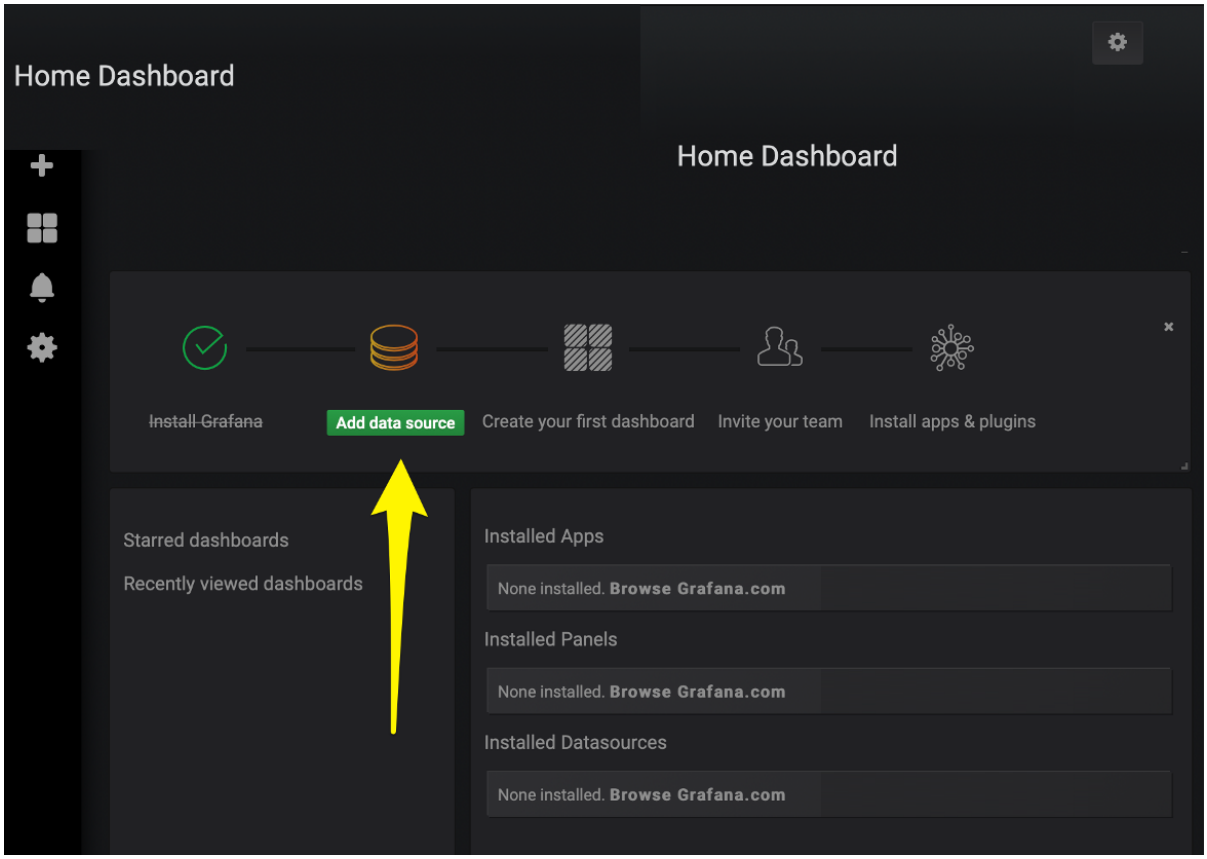


Figure 7.5 Click on "Add data source" to create a new Prometheus data source

To configure the datasource as seen in Figure 7.6, give it a name, change the "Type" to `Prometheus` and add the HTTP URL for the Prometheus server. In our case, it will be `prometheus:9090`. Fill it in like the following image:

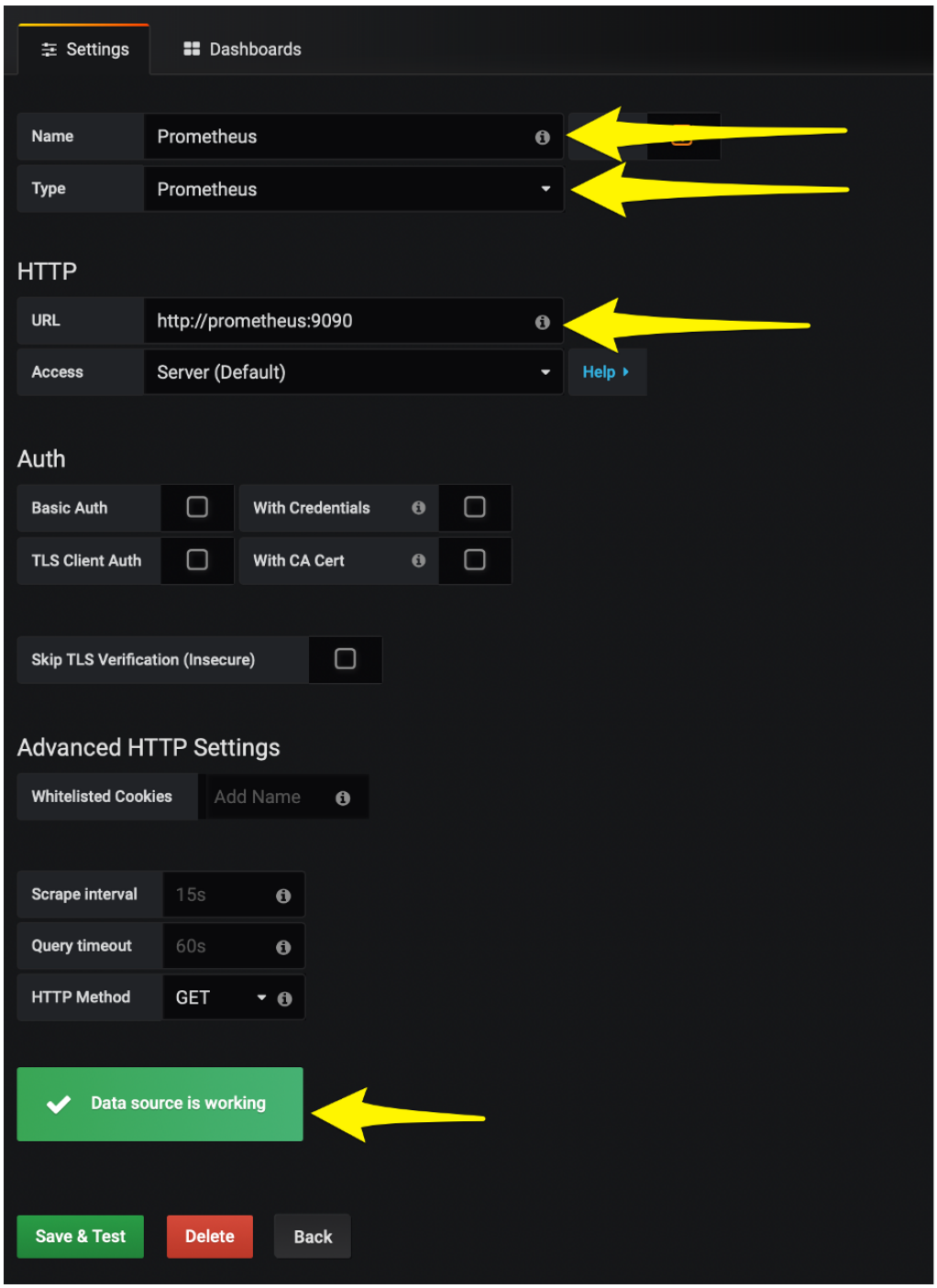


Figure 7.6 Fill in the fields to connect the Prometheus we created to Grafana as a data source

At this point we can start creating new dashboards with the metrics that exist in Prometheus. To do that, click on the left-hand "+" sign and add a new dashboard.

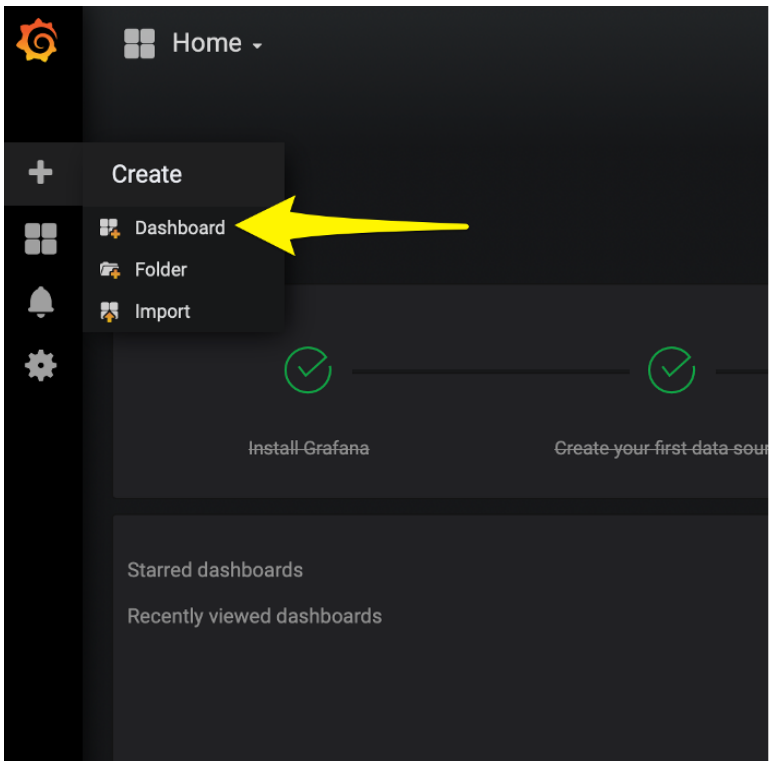


Figure 7.7 Click on the + sign to add a new dashboard

You can choose whichever visualization tool you wish, but for this simple exercise, we'll choose a Graph:

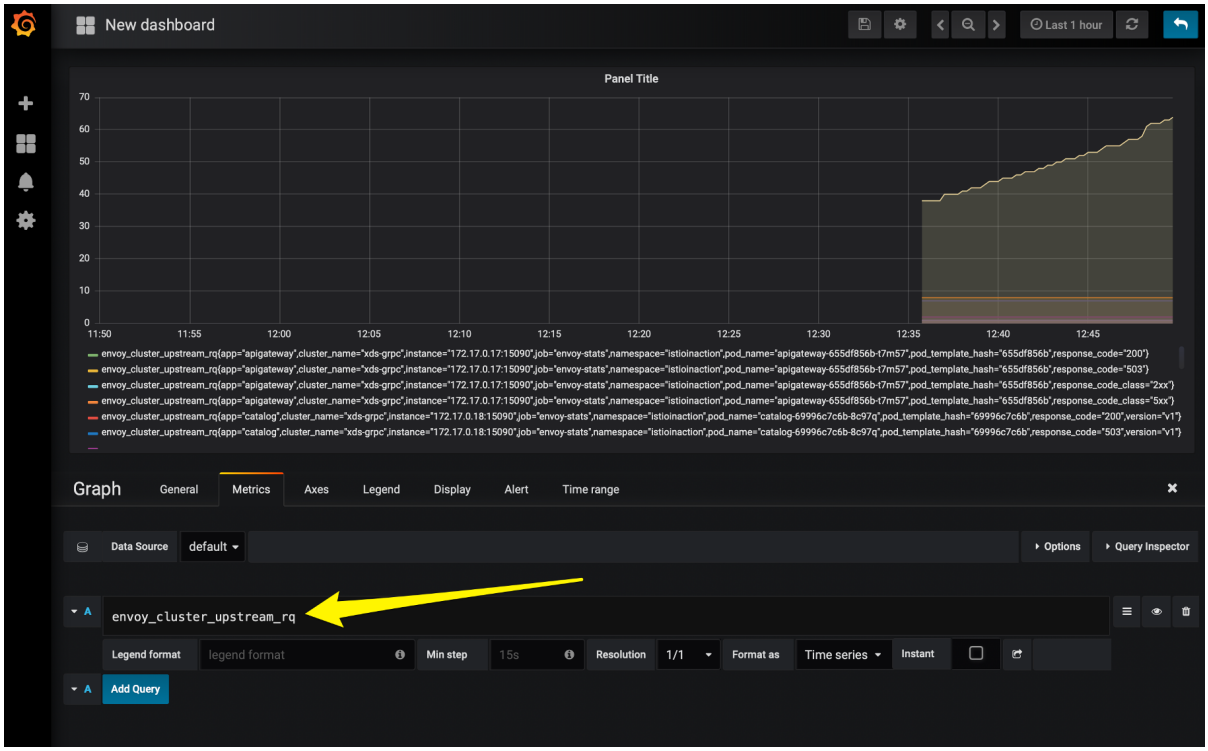


Figure 7.8 View cluster upstream metrics in Grafana

You can now enter some envoy metrics into the query box and get the metrics plotted against the

graph:

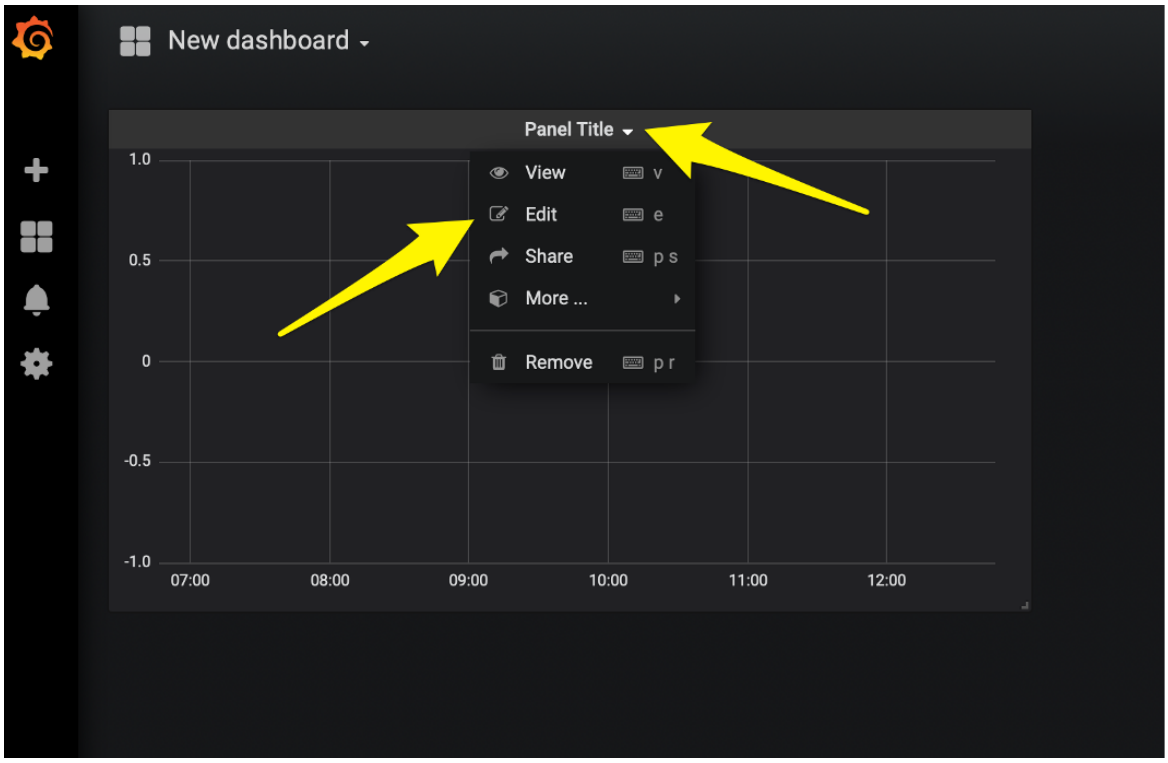


Figure 7.9 You can edit the details of the graph

You can build beautiful and sophisticated visualizations from the Prometheus metrics that have been gathered from Istio. Please check out the Grafana documentation for more. There are a handful of dashboards pre-built for Istio that can also be re-used. These exist on the Grafana website. Lastly, as mentioned, Istio comes with some pre-integrated Prometheus and Grafana deployments as part of the Istio installation. Check out the Istio community documentation for more.

7.3 Creating new metrics to send to Prometheus through Istio-telemetry

So far we've seen how to collect and visualize metrics from the service proxies running in a mesh. The metrics we're able to collect are from a pre-defined set of metrics that the Istio service proxy collects and makes available. In a real production environment, you may find that you will want to fine-tune which metrics you capture from both the set of pre-defined metrics as well as possibly introducing new ones.

The metrics that we collected in the previous section originate from the Istio service proxy and get sent to or pulled into a metrics-collection engine or time-series database. Although there are a lot of metrics, they are a fixed set that are not easily extendable. If we want to introduce a new metric to our observability tools, we'll need to explore a different path.

In Istio, not only can we collect the established signals from the Istio service proxy, but we have access to runtime metadata when requests are made. For example, when service A calls service B, the Istio service proxy on the A service records a set of attributes and metadata about the request like where it originates, where it is beening sent (to service B in this case), the request path and many others. The Istio service proxy periodically sends this metadata to the Istio control plane which is then collated and processed by backend systems. Istio allows a flexible way to extend this backend processing engine which gives us the ability to create our own metrics built on this request metadata.

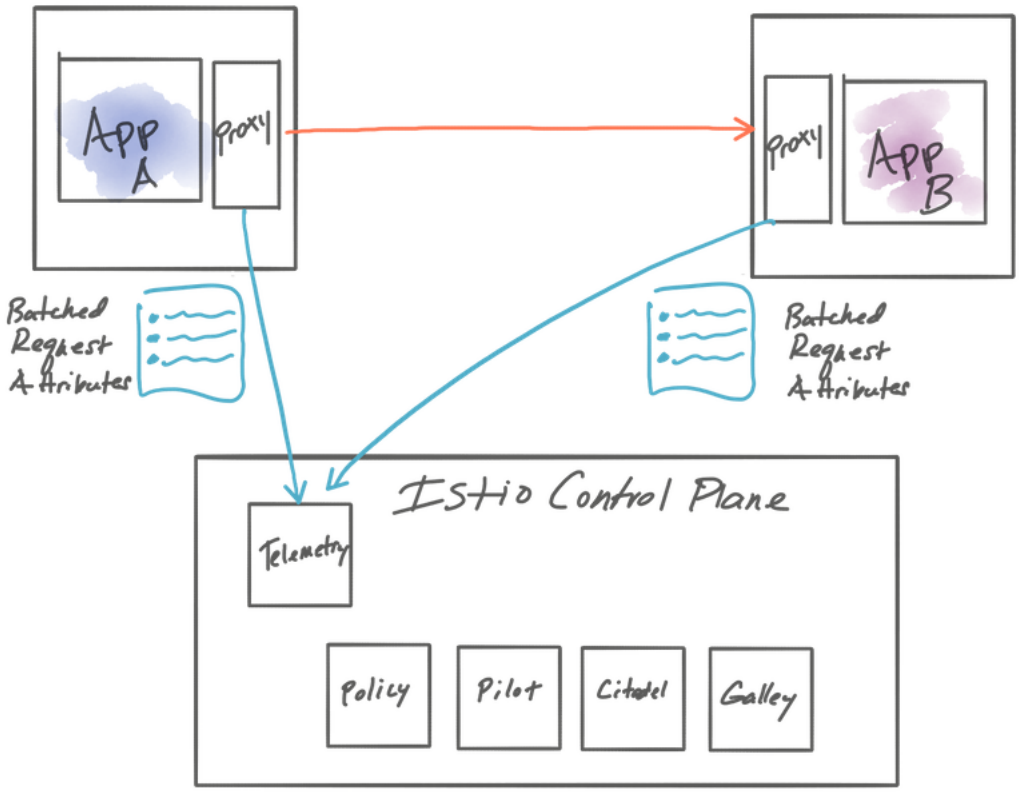


Figure 7.10 Istio records metadata attributes about each request coming or going from the Istio service proxy and periodically reports that back to the Istio control plane

In this section, we'll take a look at how to create new metrics as a result of this attribute collection and how to expose this to Prometheus. In the Policy and Telemetry Aggregation chapter, we'll dig deeper into how this all works including how to write adapters to this functionality.

Creating a new metric to capture in Istio Telemetry requires a few different configuration objects. The first thing we want to do is define **what** attributes will comprise our new metric. Istio refers to these as `dimensions` in it's configuration. Next, we want to tell Istio **where** should these new metrics go. We'll discuss more about Istio Telemetry and Policy, but just know that these components are extendable with adapters and when we specify where to send a metric for this example, we need to specify and configure a specific adapter to handle the metric. Lastly, we need to define what matching criteria are required to trigger creating the metric. In the following

example, we're going to define a specific new metric called `apigatewayrequestcount` that will count the number of requests going to the api gateway component and will only match if a request is sent to the api-gateway component. For those metrics, we will send them to the Prometheus adapter.

Let's start by creating a new `metric` in Istio. To do that, take a look at the following:

```
apiVersion: "config.istio.io/v1alpha2"
kind: metric
metadata:
  name: apigatewayrequestcount
  namespace: istio-system
spec:
  value: "1" ❶
  dimensions:
    source: source.workload.name | "unknown" ❷
    destination: destination.workload.name | "unknown"
    destination_ip: destination.ip
  monitored_resource_type: '"UNSPECIFIED"'
```

❶ What gets recorded

❷ Dimensions that get tracked

This `metric` specifies what request attributes we are interested in and what dimensions of the metric we want to report. For this simple example, let's say we want to create a metric which specifically counts how many requests are made to our apigateway service. In the above, we set the `value` of the metric to `1` and we track the `dimensions` like `source`, `destination` and `destination_ip`. We are using the value of `1` for this metric because we want to count each request **once**. The `value` field can also contain expressions that evaluate to numbers, strings, IP addresses, and others. Please see the Istio reference documentation on metrics for more.

When we have framed out what our metric will look like, we need to specify where this metric should be sent in the Istio attribute-processing engine (a handler) and finally how to bind to that handler. To do that, let's define a Prometheus handler with the following:

```
apiVersion: "config.istio.io/v1alpha2"
kind: prometheus ❶
metadata:
  name: apigatewayrequestcounthandler
  namespace: istio-system
spec:
  metrics:
    - name: apigateway_request_count
      instance_name: apigatewayrequestcount.metric.istio-system ❷
      kind: COUNTER ❸
      label_names:
        - source
        - destination
        - destination_ip
```

❶ A Prometheus handler

❷ Prometheus metric name

❸ Type of metric

Notice, we created the handler as type `prometheus`. There are a handful of out of the box Adapters and handlers such as:

- Prometheus
- SignalFx
- StasD
- Wavefront

And many more. In this case, we specified a `prometheus` handler, the name of the metric to be used in Prometheus (`apigatewayrequest.metric.istio-system`), the specific labels that we want propagated to prometheus based on the dimensions we declared for the metric instance previously, and the "kind" of telemetry we'll be propagated to Prometheus. In this case we are sending a `COUNTER` but other "kinds" can be:

- DISTRIBUTION
- GAUGE
- UNSPECIFIED

Lastly, we want to create a `rule` that binds this `metric` to the `prometheus` handler and specify any conditions that must be satisfied for this rule to fire. In this case, we're specifying a `match` clause that checks the `destination.service` attribute and tests it against the `apigateway.istioinaction.svc.cluster.local` service name.

```
apiVersion: "config.istio.io/v1alpha2"
kind: rule
metadata:
  name: apigatewayrequestcountrule
  namespace: istio-system
spec:
  match: destination.service == "apigateway.istioinaction.svc.cluster.local"
  actions:
    - handler: apigatewayrequestcounthandler.prometheus
      instances:
        - apigatewayrequestcount.metric
```

Now that we've seen which objects we'll need to create our new metric, let's create these objects here:

```
$  kubectl create -f chapters/chapter7/istio-telemetry/new-metric.yaml
metric.config.istio.io/apigatewayrequestcount created
prometheus.config.istio.io/apigatewayrequestcounthandler created
rule.config.istio.io/apigatewayrequestcountrule created
```

Now if we send in a few requests:

```
curl  -H "Host: apiserver.istioinaction.io"  \
  $(make ingress-url)/api/catalog
```

Note, we're running this from the root of the accompanying book source code which has a

Makefile and a make target for getting the ingress url. If you're running your Kubernetes cluster on a cloud provider, you can check the `ExternalIP` of the `istio-ingressgateway` service. If you're running on minikube, you can grab the `NodePort` from the `istio-ingressgateway` service like this:

```
$  minikube service list
|---------------|----------------------|---------------------------------|
|   NAMESPACE   |         NAME         |               URL               |
|---------------|----------------------|---------------------------------|
| default       | kubernetes           | No node port                    |
| istio-system  | grafana              | No node port                    |
| istio-system  | istio-citadel        | No node port                    |
| istio-system  | istio-egressgateway  | No node port                    |
| istio-system  | istio-galley         | No node port                    |
| istio-system  | istio-ingressgateway | http://192.168.64.35:31380      |
|               |                      | http://192.168.64.35:31390      |
|               |                      | http://192.168.64.35:31400      |
|               |                      | http://192.168.64.35:30714      |
|               |                      | http://192.168.64.35:31331      |
|               |                      | http://192.168.64.35:30785      |
|               |                      | http://192.168.64.35:32306      |
|               |                      | http://192.168.64.35:30753      |
```

The first entry for `istio-ingressgateway` is the full URL for the HTTP port for the ingress gateway.

Now we can check the Prometheus server to see whether we have our new metric. First, let's port-forward the prometheus server:

```
$  kubectl -n istio-system port-forward $(kubectl -n istio-system get pod -l app=prometheus
   -o jsonpath='{.items[0].metadata.name}') 9090:9090
```

And navigate to localhost:9090/graph[localhost:9090/graph]. You can start typing `istio_apigateway` and the full metric name should auto-complete:

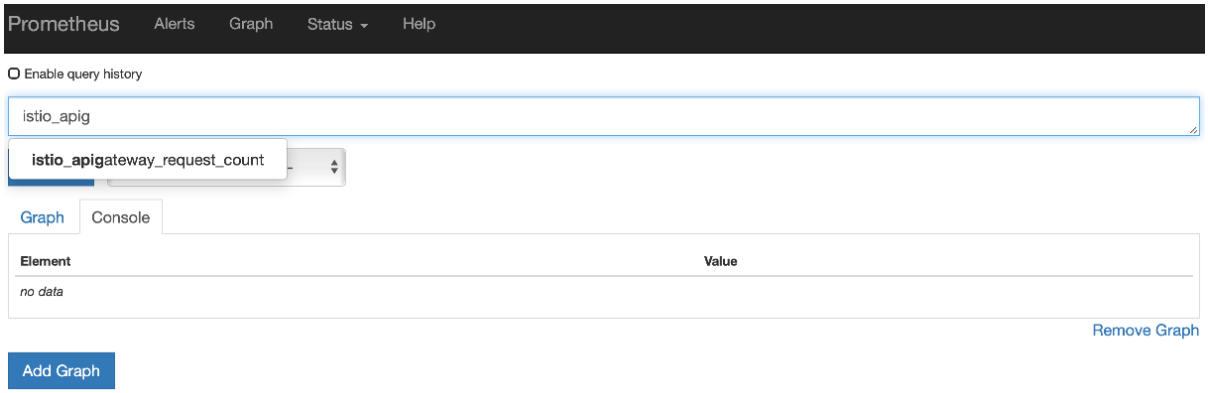


Figure 7.11 Begin typing to get autocomplete on the new metric we just added

Click on the metric name, and click "Execute". You should see values for our new metric:

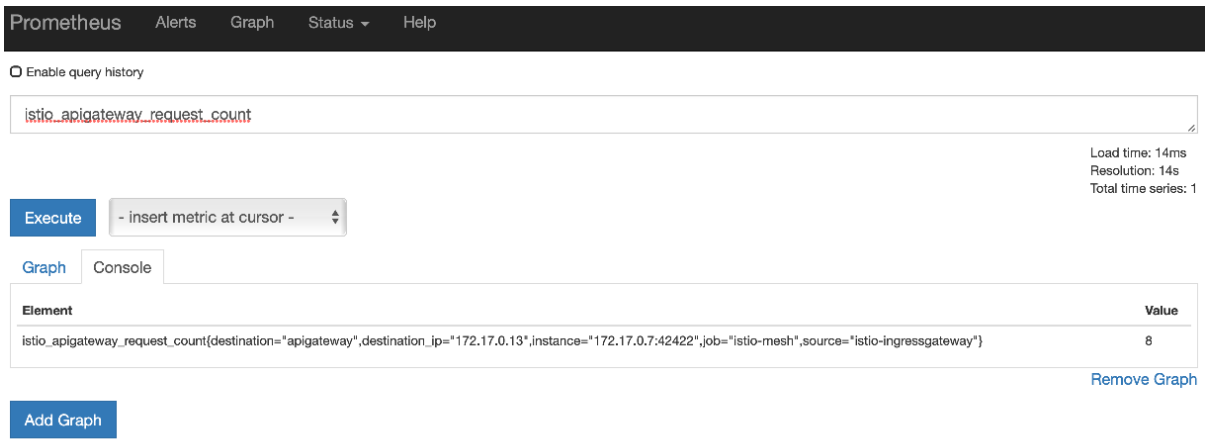


Figure 7.12 Clicking "execute" should query Prometheus for the new metrics we just added

If you'll recall in the previous section, Prometheus pulls or "scrapes" an endpoint for metrics. The `istio-telemetry` component exposes this Prometheus adapter on `istio-telemetry:9093/metrics` which gets scraped by Prometheus. Our new metrics should be exposed via this endpoint and eventually make it to our Prometheus server.

In this section we saw how to create new metrics to track in our Prometheus servers. We based the new metrics off an existing pattern of attribute and metadata collected for each request. We created the new metric by specifying a metric handler and rule using Istio Telemetry constructs. We will cover this general attribute processing pattern in more detail in Chapter XX. When new metrics are being recorded by the Prometheus handler in Istio Telemetry, we can then review them in the Prometheus console as well as Grafana (or any visualization tool you're using for Prometheus).

Telemetry collection is vital for any observable system. In the next section we take a look at another set of signals useful to debug a running distributed system called distributed tracing.

7.4 Distributed tracing with OpenTracing

As we tend to build more applications as microservices, we are creating a network of distributed components that work together to achieve a business objective. When things start to go wrong on the request path, it's critical to understand what's happening so we can diagnose it quickly and fix it. In the previous sections we've seen how Istio can help collect metrics and telemetry related to networking on behalf of the application. In this section, we'll take a look at a concept called distributed tracing and see how it can help diagnose misbehaving requests as it traverses a web of microservices.

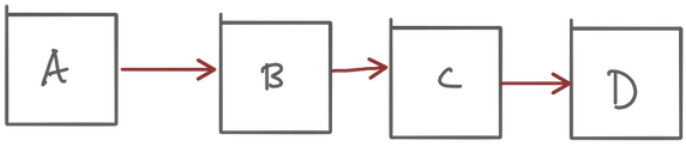


Figure 7.13 Services often take multiple hops to service a request; we need the ability to see what hops were involved for a given request and information about how long it took at each op

In a monolith, if things start to misbehave, we can jump in and start debugging with familiar tools at our disposal. We have debuggers, runtime profilers, and memory analysis tools to find areas where parts of the code introduce latency or trigger faults that cause an application feature to misbehave. With an application made up of distributed parts, we need a new set of tools to accomplish the same things.

Open Tracing is a community-driven specification that captures concepts and APIs related to request flow through a graph of microservices. Capturing contexts and constructing a causal-relationship between services in a request path is known as distributed tracing. Distributed tracing in part relies on developers instrumenting their code and sharing information as it processes a request with a tracing engine to help put together the full picture of a request flow which can be used to identify misbehaving areas of our architecture. The origins of OpenTracing can be found in the Google Dapper paper which describes the distributed-tracing system Google built within its platform. With Istio, we can provide the bulk of the heavy lifting developers would otherwise have to implement themselves and provide distributed tracing as part of the service proxies co-deployed with an application.

7.4.1 How does it work

At its simplest form, distributed tracing with OpenTracing consists of applications creating `Spans`, sharing those spans with an OpenTracing engine, and propagating a trace context to any of the services it subsequently calls. A `Span` is a collection of data about a request representing a "unit of work" within a service or component. This data including things like the "start time" of the operation, the "end time", the operation name, and a set of tags and logs.

Those upstream services in turn do the same thing: create a `Span` capturing its part of the request, sends that to the OpenTracing engine, and further propagates the context to other services. Using these `Spans` and the request context, the distributed-tracing engine can construct a `Trace` which is a causal relationship between services that show direction, timing, and other debugging information. `Spans` have their own ID and a parent ID which is the `Trace` ID. These IDs are used for correlation and are expected to be propagated between services.

OpenTracing implementations include systems like:

- Jaeger
- Zipkin

- LightStep
- Instana

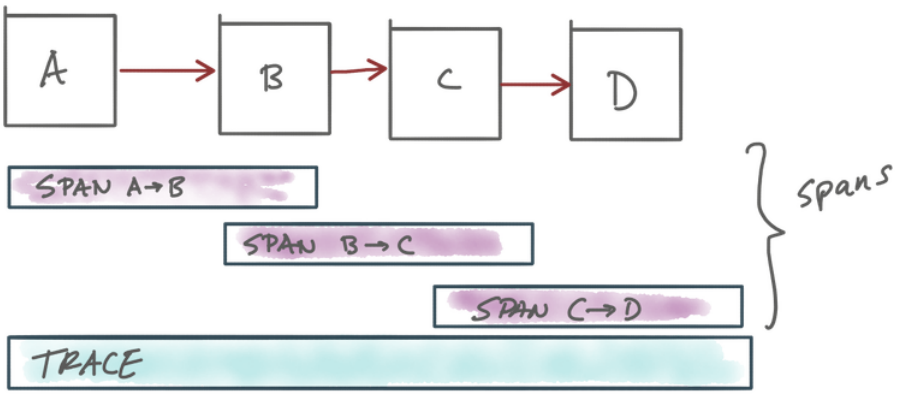


Figure 7.14 With distributed tracing, we can collect Spans for each network hop and capture them in an overall Trace and use them to debug issues within our call graph

Istio can handle sending the `Spans` to the distributed tracing engine. This means you don't need language-specific libraries and application-specific configuration to do this. When a request traverses the Istio service proxy, a new trace is started if there isn't one in progress, and the start/end times for the request are captured as part of the `Span`. Istio appends HTTP headers, commonly known as the Zipkin tracing headers, to the request that can be used to correlate subsequent `Span` objects to the overall `Trace`. If a request comes in to a service and the Istio proxy recognizes the distributed-tracing headers, it will treat it as an in-progress trace and will not try to generate a new one. The following Zipkin tracing headers are used by Istio and the distributed-tracing functionality:

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

For the distributed tracing functionality provided by Istio to work across the entire request call graph, each application will need to propagate these headers to any upstream calls they make. The reason for this is Istio cannot know which calls were a result of which incoming requests. To be able to correctly correlate upstream calls with calls that came into the service, the application must assume the responsibility of propagating these headers. Many times, out of the box RPC frameworks integrate with or directly support OpenTracing and can automatically propagate these headers for you. Either way, the application must ensure these headers get propagated.

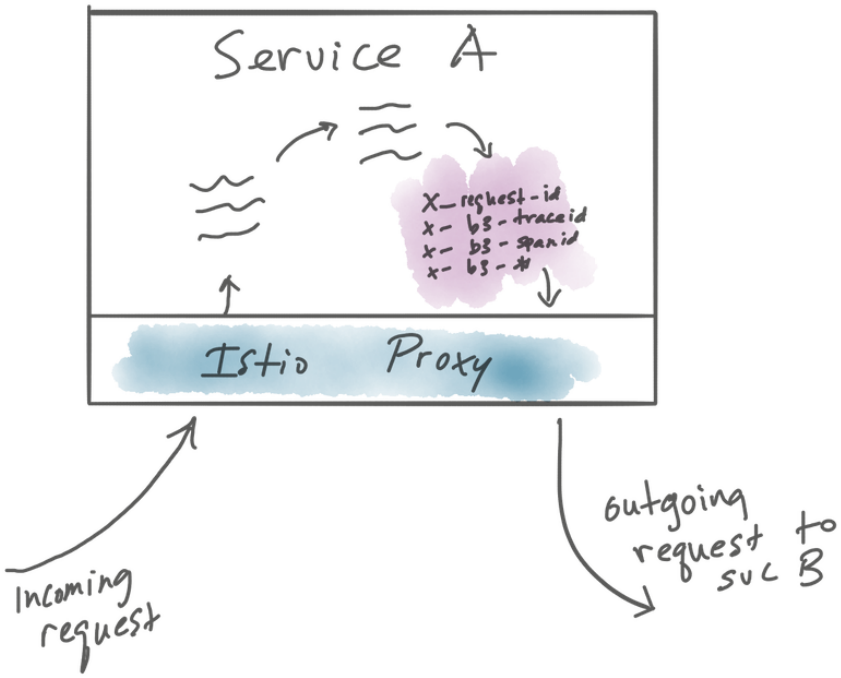


Figure 7.15 The application must propagate the tracing headers

7.4.2 Configuring Istio to perform distributed tracing

To configure Istio to send distributed-tracing spans to an OpenTracing engine, we need to configure the Istio service proxy with the location of the tracing engine. For example, when we first installed Istio in Chapter 2, we installed the default OpenTracing engine called Jaeger. Jaeger is a project from Uber and part of the Cloud Native Computing Foundation (CNCF) which implements the OpenTracing specification. To verify we have Jaeger running with the default installation of Istio:

```
$  kubectl get pod -n istio-system
NAME                                      READY   STATUS    RESTARTS   AGE
grafana-59b8896965-vlf6h                  1/1     Running   0          22h
istio-citadel-6f444d9999-cd97r            1/1     Running   0          22h
istio-egressgateway-6d79447874-n7xdz      1/1     Running   0          22h
istio-galley-685bb48846-z2mp4             1/1     Running   0          17h
istio-ingressgateway-5b64fffc9f-zzwm5     1/1     Running   0          22h
istio-pilot-8645f5655b-md5tn              2/2     Running   0          22h
istio-policy-547d64b8d7-ltds9             2/2     Running   1          22h
istio-telemetry-c5488fc49-nkvd8           2/2     Running   0          22h
istio-tracing-6b994895fd-c4znx            1/1     Running   0          22h
kiali-5f9ffff7cf-dfn6x                    1/1     Running   0          21h
prometheus-76b7745b64-vnltm               1/1     Running   4          22h
```

We should see an `istio-tracing` pod. When our Istio service proxies are configured, we should see a parameter passing in the location of the tracing service:

```
$  kubectl  get pod apigateway-755c6969ff-fj8c6 -o yaml | grep zipkin
    - --zipkinAddress
    - zipkin.istio-system:9411
```

Note although we're using `istio-tracing` which is the Jaeger tracing system, for legacy

reasons, the configuration parameters are still configured with the `zipkin` parmeters. Envoy sends the spans using the Zipkin APIs to our OpenTracing system, Jaeger, listeneing on port `9411` and Jaeger stores the spans and reconstructs the traces with the correct spans based on the Trace ID and Span ID correlation IDs.

To demonstrate Istio automatically injecting the OpenTracing headers and correlation IDs, let's try to use Istio's ingress gateway to call an external `httpbin` service and call an endpoint that displays the request headers. Let's deploy an Istio `VirtualService` that does this routing:

```
$ kubectl create -f chapters/chapter7/tracing/thin-httpbin-virtualservice.yaml
```

Now let's figure out the right URL for minikube

```
$  GATEWAY_RULE=$(make ingress-url)
$  curl -H "Host: httpbin.istioinaction.io" http://$GATEWAY_URL/headers
{
  "headers": {
    "Accept": "*/*",
    "Host": "httpbin.istioinaction.io",
    "User-Agent": "curl/7.54.0",
    "X-B3-Sampled": "0",
    "X-B3-Spanid": "463e3ca7c53e2c85",
    "X-B3-Traceid": "463e3ca7c53e2c85",
    "X-Envoy-Decorator-Operation": "httpbin.org:80/*",
    "X-Envoy-Internal": "true",
    "X-Istio-Attributes":
        "CikKGGRlc3RpbmF0aW9uLnNlcnZpY2UuaG9zdBINEgtodHRwYmluLm9yZwopChhkZXN0aW5hdGlvbi5zZXJ2aWNlLm
        5hbWUSDRILaHR0cGJpbi5vcmcKKgodZGVzdGluYXRpb24uc2VydmljZS5uYW1lc3BhY2USCRIHZGVmYXVsdAokChNkZ
        XN0aW5hdGlvbi5zZXJ2aWNlEg0SC2h0dHBiaW4ub3JnCk8KCnNvdXJjZS51aWQSQRI/a3ViZXJuZXRlczovL2lzdGlv
        LWluZ3Jlc3NnYXRld2F5LTViNjRmZmZjOWYtenp3bTUuaXN0aW8tc3lzdGVt"
  }
}
```

When we called our Istio ingress gateway, we were routed to an external URL `httpbin.org` which is a simple HTTP testing service. When we `GET` the `/headers` endpoint, it returns to us the request headers we used with the request. We can clearly see the `x-b3-\*` Zipkin headers were automatically appended to our request.

Now, let's see how we can use this within our call graph to generate distributed traces so we can observe the runtime call graph of our services and use to debug our services.

7.4.3 Viewing distributed tracing data

When spans get sent to Jaeger (or any OpenTracing engine), we need a way to query and view the Traces and their associated spans. Using the out of the box Jaeger UI, we can do just that. To view the UI, let's port-forward it locally:

```
$  kubectl port-forward deploy/istio-tracing -n istio-system 8080:16686
```

Now if we navigate to http://localhost:8080 we should be able to see the Jaeger UI.

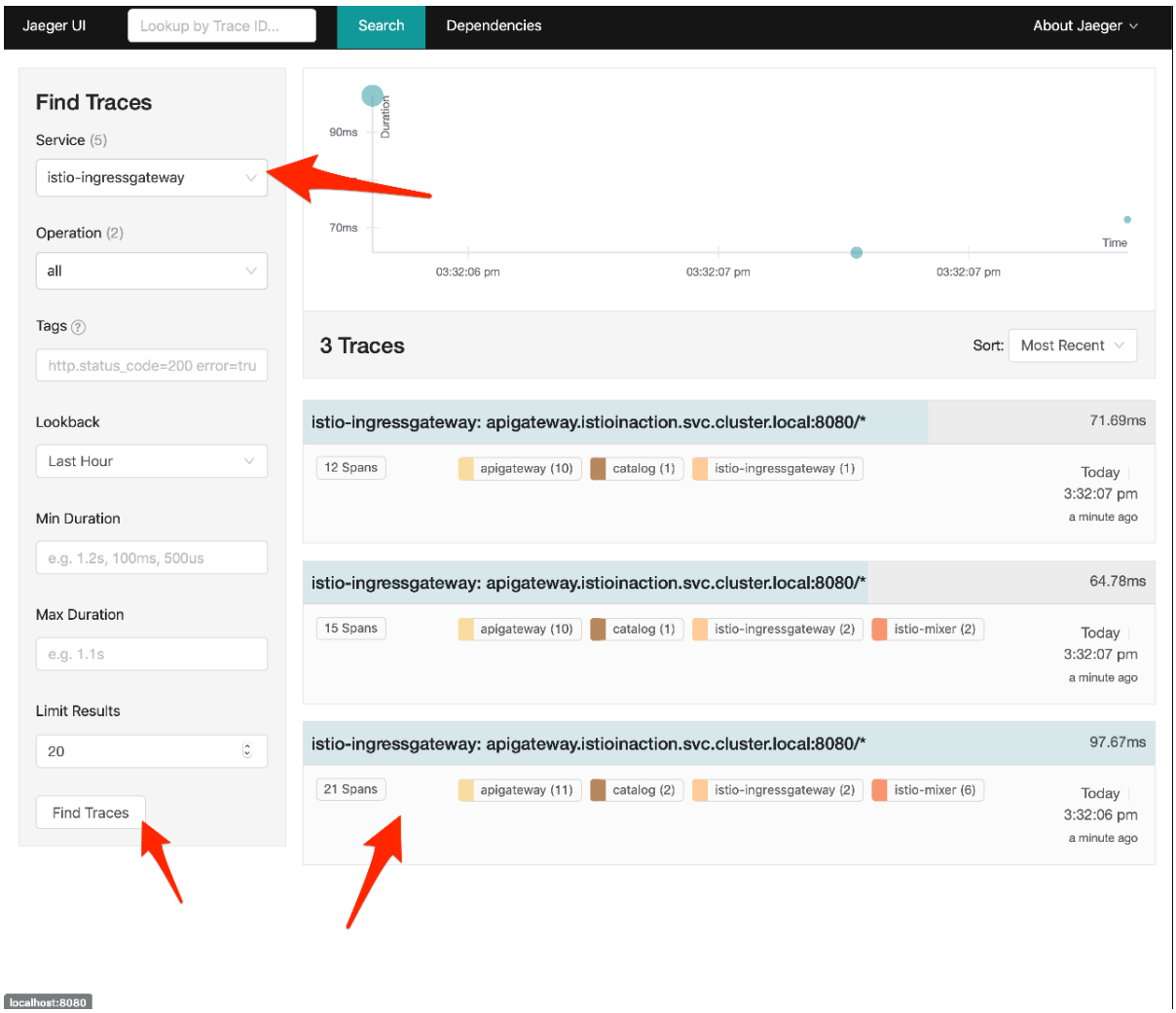


Figure 7.16 Choose the istio-ingressgateway service to see the requests that have come into our cluster

If you click the "Services" dropdown, select `istio-ingressgateway`. On the bottom left, click `Find Traces`. If you don't see any traces, try sending some traffic through the `istio-ingressgateway` :

```
$  GATEWAY_RULE=$(make ingress-url)
$  curl -H "Host: apiserver.istioinaction.io" http://$GATEWAY_URL/api/catalog
```

Now click back to the UI and try `Find Traces` again. You should see a new trace for each attempt you made to call the sample services.

It's possible at this point you still don't see any traces (or much fewer than you would expect). If that is the case, skip to the next section where we discuss trace collection aperture.

If you do see traces (or you've opened the aperture per the next section), you should be able to click on a specific trace and drill deeper into the runtime call graph. Feel free to explore the information collected by Istio.

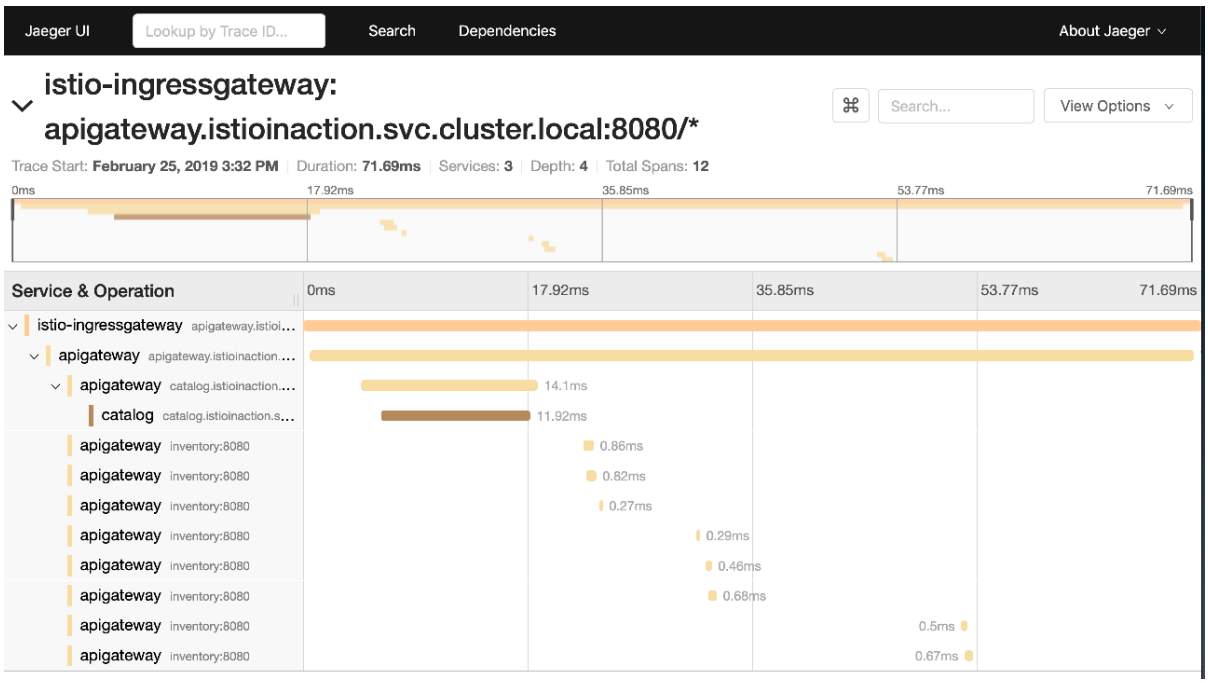


Figure 7.17 Clicking into a specific trace shows more ganular detail like the specific spans that make up the trace

Note, for the sample applications to work correctly, they must progpagate the Zipkin trace headers:

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

7.4.4 Limiting tracing apeture

OpenTracing collection can impose a hefty performance penalty for your system, so you may opt to restrict how frequently you collect distributed traces when your services are running correctly. You can control the amount of trace collection or the "apeture" of the trace collection by configuring the percentage of traces to collect in the system. At the time of this writing, the configuration for the distributed-tracing apeture is set as an environment variable on the `istio-pilot` Deployment. You can edit the amount of tracing to collect by tuning the `PILOT_TRACE_SAMPLING` environment variable like this:

```
$  kubectl -n istio-system edit deploy istio-pilot
```

Find the `PILOT_TRACING_SAMPLING` and tune it between `1.0` and `100.0`. These values represent a % of requests for which tracing will be turned on. This will affect tracing for the entire service

mesh.

Here's an example of what what the Istio Pilot configuration would look like:

```
containers:
- args:
  - discovery
  env:
  - name: POD_NAME
    valueFrom:
      fieldRef:
        apiVersion: v1
        fieldPath: metadata.name
  - name: POD_NAMESPACE
    valueFrom:
      fieldRef:
        apiVersion: v1
        fieldPath: metadata.namespace
  - name: PILOT_CACHE_SQUASH
    value: "5"
  - name: GODEBUG
    value: gctrace=2
  - name: PILOT_PUSH_THROTTLE_COUNT
    value: "100"
  - name: PILOT_TRACE_SAMPLING ❶
    value: "50.0" ❷
```

❶ Env variable to tune

❷ Trace 50% of requests

Another approach to limit the aperture of distributed tracing is to turn it on for only specific requests. If we change the `PILOT_TRACE_SAMPLING` back down to the default of `1.0` we shouldn't see traces for every request we send to our sample apps.

If we would like Istio to record a trace for a specific request, we can add the `x-envoy-force-trace` header to the request. Try it out in our sample application:

```
$  curl -H "x-envoy-force-trace: true"  \
-H "Host: apiserver.istioinaction.io" \
http://$GATEWAY_RUL/api/catalog
```

Every time we send in this `x-envoy-force-trace` header, we can turn on the tracing for that request and for the entire call graph of that request. You can build different tools on top of Istio, like API Gateways or diagnostics services, that can inject this header when we want to know more about a particular request. Building these types of tools is outside the scope of this book.

7.5 Visualization with Kiali

Istio comes with a powerful visualization dashboard from an open-source project named Kiali that can assist understanding the service mesh at runtime. Kiali pulls a lot of the metrics from Prometheus and the underlying platform (like Kubernetes) and establishes a runtime graph of the components in the mesh to give you a visual overview of what services are communicating with others. You can also interact with the graph and dig into areas that could be problems to learn more about what's happening. Kiali is different from Grafana in that it focuses on building a directed graph of how the servcies interact with each other with live-updating metrics. Grafana is great at dashboards with guages, counters, charts, and more, but does not present an interactive drawing or "map" of the services in the cluster. In this section, we'll look at the capabilities of the Kiali dashboard.

Before we get started, we will want to configure Kiali with a username and password so we can log in to the user interface. If you installed Istio with the Helm charts, you may have been prompted to create the username and password already. If so, and you remember your Kiali username and password, skip this section. If not, let's start by creating the username in a Kubernetes `secret` named `kiali`. The username and password will be base64 encrypted with the following commands:

```
$  KIALI_USERNAME=$(read -p 'Kiali Username: ' uval && echo -n $uval | base64)
$  KIALI_PASSPHRASE=$(read -sp 'Kiali Passphrase: ' pval && echo -n $pval | base64)
```

Now we should create the Kiali secret for Kubernetes:

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Secret
metadata:
  name: kiali
  namespace: istio-system
  labels:
    app: kiali
type: Opaque
data:
  username: $KIALI_USERNAME
  passphrase: $KIALI_PASSPHRASE
EOF

secret/kiali configured
```

NOTE **Handling secrets securely**

These username and password fields are base64 encoded and are NOT encrypted. You can use application level encryption for the secrets or a way to retrieve them dynamically using something like HashiCorp Vault. Please see XXXX for more information.

Now you can install Kiali. If you have already installed Kiali (or was installed by previous steps

along with Istio), then you need to restart the Kiali Pod so that these secrets take effect:

```
$  kubectl delete pod -n istio-system  $(k get pod -n istio-system | grep kiali | awk '{ print $1 }')
pod "kiali-5f9ffff7cf-4lqtr" deleted
```

Now, we'll want to port forward to the Kiali pod so we can view the dashboard locally.

```
$  istioctl dashboard kiali --port 8080
```

The Kiali consle is now accssible from http://localhost:8080.

Logging in to the console with the username and password you created previously should take you to the main dashboard:

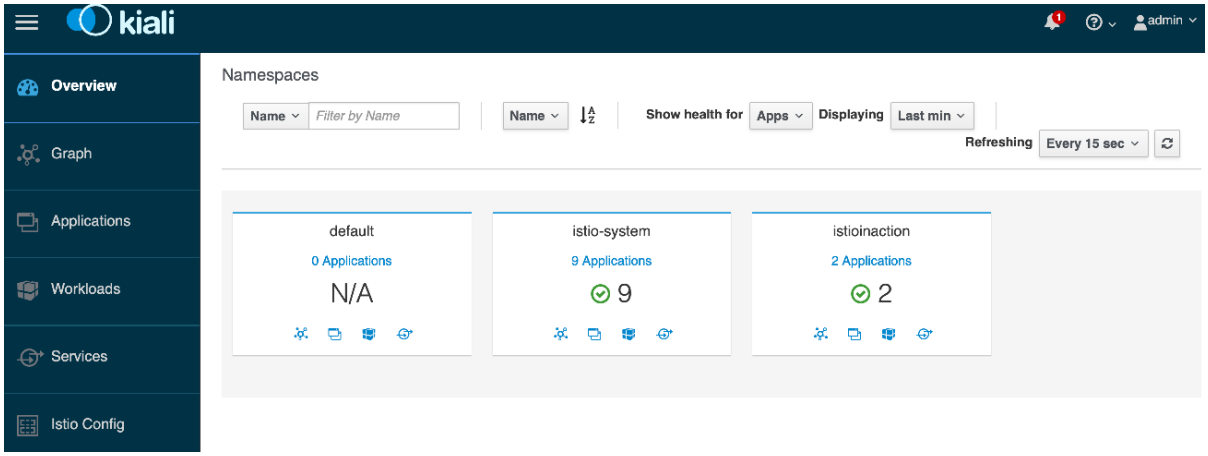


Figure 7.18 Overview dashboard of the service mesh using Kiali

In this overview dashboard, you can see the different namespaces and how many applications are running in each. You also have a visual indication of the overall health of the applications running in the respective namespaces. If you click on the `2 applications` link in the above screen, you will be taken to the overview of all of the applications in that namespace. If you have any issues with the applications, you will be given more information about what's happening with the traffic. For example:

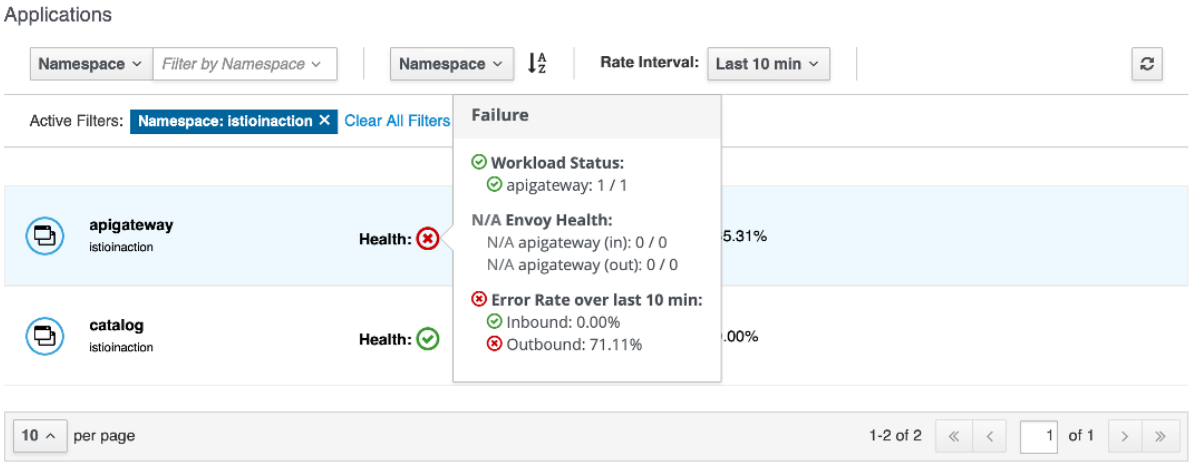


Figure 7.19 Information about application health in a particular Kubernetes namespace

To get some meaningful reporting in the Kiali dashboard, let's make a few calls to the application:

```
$  curl -H "Host: apiserver.istioinaction.io" http://192.168.64.35:31380/api/catalog
```

NOTE **Understanding Kiali workload vs application**

In Kiali, you'll notice a distinction between a `workload` and an `application`. For our sample application, they're effectively the same, but the big distiction between the two is this:

A workload is a running binary that can be deployed as a set of identical-running replicas. For example, in Kubernetes this would be the Pods part of a Deployment. If we had a "service A" deployment with 3 replicas, this would be a workload.

An application is a grouping of workloads and associated constructs like services and configuration. In Kubernetes this would be a "service A" along with a "service B" and maybe a "database". These would each be their own workload, but they would together make up a Kiali application.

If we click on the `catalog` application, we'll be taken to an application-specific dashboard with "Inbound Metrics" and "Outbound Metrics". If you send some traffic to the sample application using the above, you should see some metrics in the "Inbound Metrics" tab. You will not see any visualizations form the "Outbound Metrics" tab for the `catalog` service since it doesn't call any upstream services.

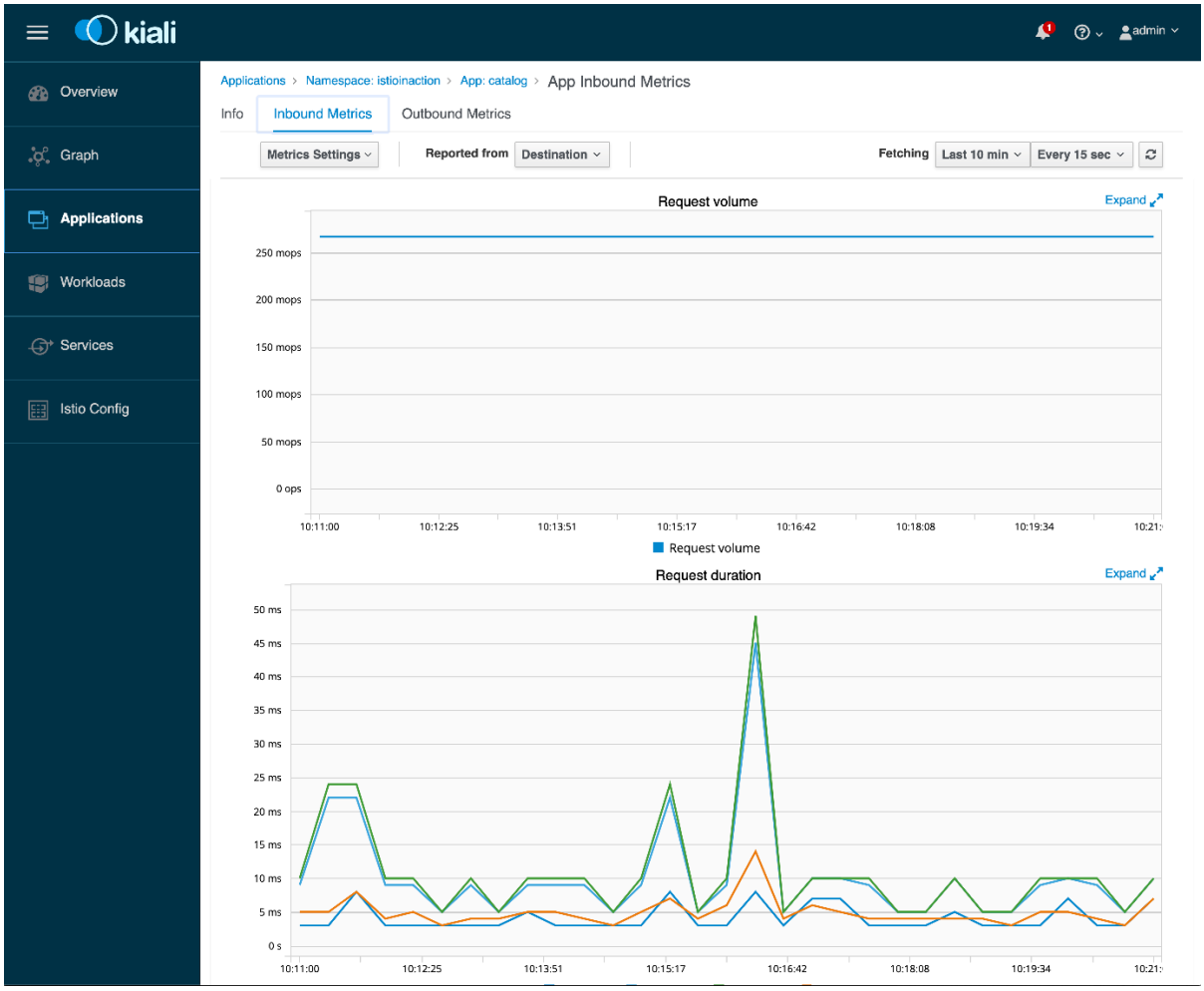


Figure 7.20 Visualize the inbound and outbound traffic for the catalog application in Kiali

You can also get a visual graph depiction of the set of services in our namespace. If you click on the "Graph" tab on the left-hand side and select the `istioinaction` namespace from the upper-left-hand corner, you will see a visualzation of our services.

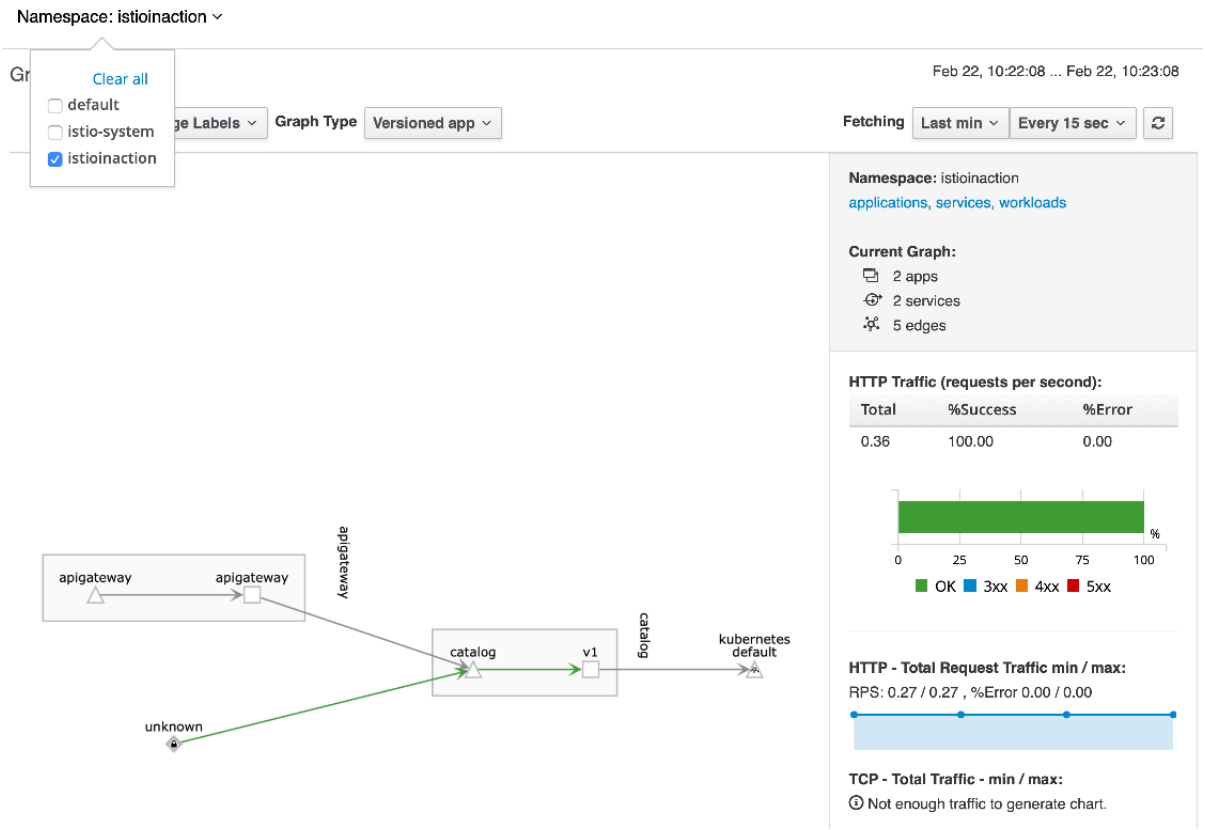


Figure 7.21 Simple visual graph of the services in our namespace and how they're connected to each other

From the graph, we can observe the following things about the mesh:

- Traversal and flow of traffic
- How many bytes, requests, etc
- Multiple traffic flows for multiple versions (think, canary release, weighted routing)
- Request/second, % of total traffic when multiple versions
- Application health based on network traffic
- HTTP/TCP traffic
- Quickfly identify networking failures

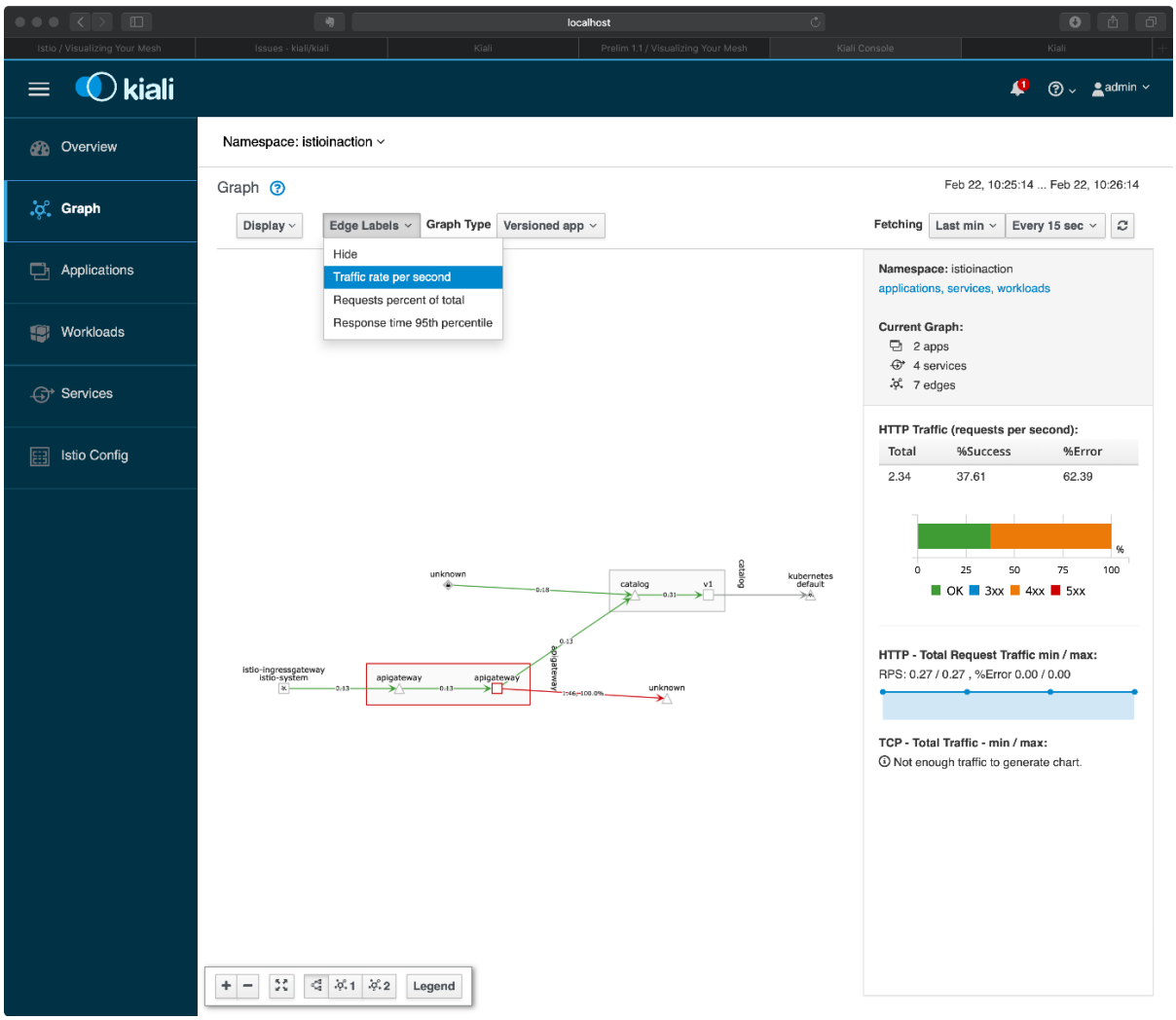


Figure 7.22 Example of identifying errant requests using the Kiali dashboard while watching traffic rate to each destination

Feel free to explore the dashboard while sending in traffic.

The last thing we'll point out about Kiali is it can discover misconfigurations in Istio resources. If you click on the "Istio Config" section, you'll see it does some semantic validation of the Istio resources and rules:

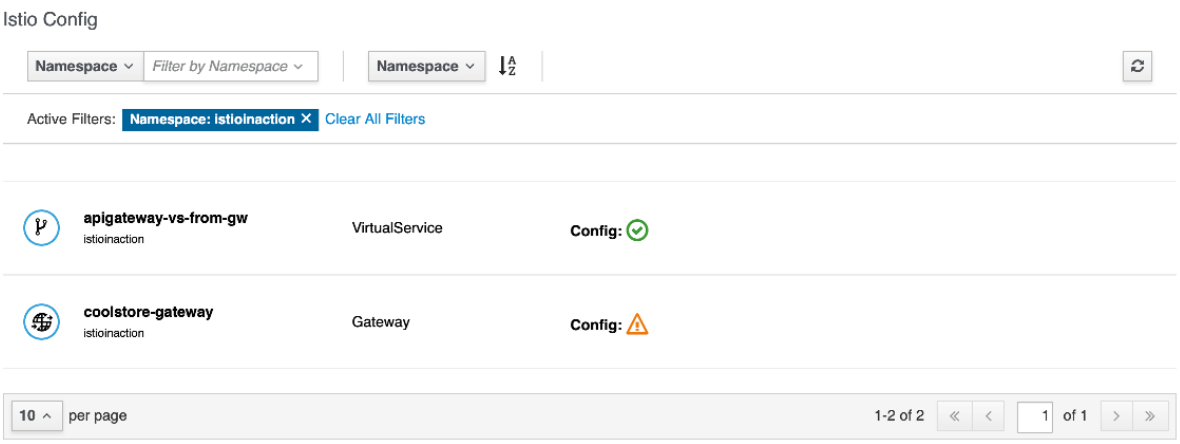


Figure 7.23 Catching semantic Istio configuration errors with Kiali

If we click on the offending resource (in this case, the `coolstore-gateway`) we see it can catch things like conflicting configurations in the overall mesh:

Figure 7.24 Semantic validation of the gateway resource

Kiali can also do the following Istio resource validations:

- VirtualService pointing to non-existant Gateway
- Routing to destinations that do not exist
- More than one VirtualService for the same host
- Service subsets not found

See the Kiali documentation for more.

7.6 Summary

Come back to this…

8 Istio Security: Effortlessly secure

This chapter covers:

- End-user and service-to-service authentication and authorization
- How Istio uses the SPIFFE specification for issuing identities to workloads
- How auto mTLS is implemented
- Handling service-to-service authentication and authorization within the service mesh
- Handling end-user authentication and authorization

In the previous chapters, we learned all about getting traffic securely into the cluster and we put into practice the benefits gained from adopting a service mesh, which injects high-level networking capabilities into the service proxies, enabling features such as monitoring, tracing, resiliency, and the fine-grained control on routing ingress traffic to our workloads. Another capability provided out of the box is being "secure by default". Which improves developer productivity and massively increases security by protecting against eavesdropping, man-in-the-middle attacks, reply attacks, etc.

In this chapter, we'll see what it means to "secure by default", how it works, how service-to-service and end-user authentication is implemented, and the access control that we have over services in the service mesh. But before getting to the features we'll take a refresher in security topics and investigate how the landscape shifted when comparing monolithic applications and microservice-based ones.

8.1 Application Security refresher

Application security comprises all activities that contribute towards the goal of protecting application data, which is of critical value and may not be compromised, stolen, or otherwise accessed by an unauthorized user.

To protect the data and ensure that only authorized clients have access to it we need:

- Authentication and Authorization of the user before allowing access to the resource
- Encryption of data in transit, to protect it from eavesdropping while it's passing through multiple networking devices to reach the server.

NOTE

- **Authentication** is the process in which a client or server proves it's identity using something they know (a password), something they have (a device, a certificate), or something they are (a unique trait such as a fingerprint).
- **Authorization** is the process of allowing or denying an already authenticated user to perform an operation such as creating, reading, updating, or deleting a resource.

The authentication of communicating parties in the world wide web is done using digitally signed certificates provisioned by the Public Key Infrastructure. The PKI is a framework that defines the process of providing the server (such as your web app) with a digital certificate as a proof of its identity and providing the client with the means of verifying the validity of the digital certificate. To dive deeper into how the PKI works check Appendix XX: "A brief overview of the Public Key Infrastructure".

The certificates provisioned by the PKI have a public and private key. The public key is contained in the certificate presented to the client as a means of authentication, and it's used by the client to encrypt data before transmitting those through the public network back to the server. Meanwhile, only the server with the private key can decrypt the data. In this manner, data is secure in transit.

NOTE **X.509 certificates**

The standard format for public key certificates is known as X.509 certificates. In the continuation of the text X.509 certificates and digital certificates will be used interchangeably.

The Internet Engineering Task Force defined the Transport Layer Security protocol that makes use of the PKI (but is not limited to) and the provisioned X.509 certificates to facilitate both authentication and encryption of traffic.

8.1.1 Traffic encryption via TLS and End-user authentication

The TLS protocol uses X.509 certificates as the primary mechanism for authenticating the validity of the servers and securely exchanging keys for symmetric encryption of the traffic in a process known as the TLS Handshake, briefly explained below:

1. The client initiates the handshake with a ClientHello, containing the TLS version and the encryption methods supported by the client.
2. The server responds with ServerHello and it's X.509 certificate containing server identity data, and the public key
3. The client verifies that the server's certificate data are not tampered with and validates the chain of trust
4. On a successful validation the client sends the server a secret key which is a randomly generated byte of string encrypted with the server's public key.
5. The server uses it's private key to decrypt the secret key and uses it for encrypting a "finished" message sent as a response back to the client.
6. The client sends the server an encrypted "finished" message using the secret key, and the TLS handshake is completed.

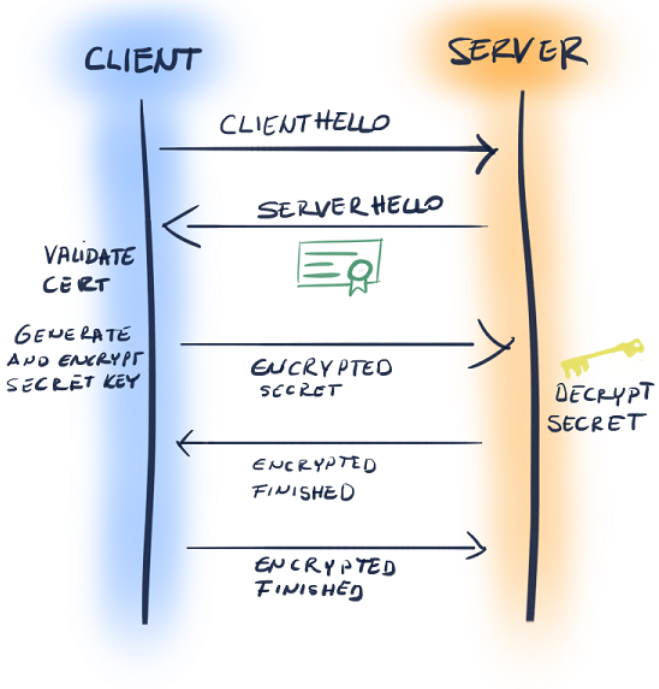


Figure 8.1 Steps of a TLS Handshake

The result of the TLS Handshake is that the client has authenticated the server and has exchanged the symmetric key securely. This symmetric key will be used to encrypt traffic between the client and the server for the duration of this connection, due to being more performant then asymmetric encryption

This process for the end-user is done transparently by the browser and denoted by the green lock in the address bar, ensuring that the receiving party is authenticated and that the traffic is encrypted and only the receiving party can decrypt it. Meanwhile, authenticating the end-user to the server is an application detail and there are multiple methods to do so, but all of those revolve around the user knowing a password and then receiving a session cookie or a JWT Token, which

preferably is short-lived and contains information to authenticate the users' subsequent requests to the server. Istio supports end-user authentication when using JWT Tokens, we'll see this in action in the section "End-user authentication and authorization" later on.

8.1.2 Service to service authentication

In contrast to end-user authentication, where only the server certificate is authenticated in service-to-service authentication both the client and the server need to provide an X.509 certificate as proof of their identity. This process is known as mutual authentication and is facilitated by the protocol mutual TLS which is abbreviated to mTLS.

8.1.3 Authorization

After the client (a service or an end-user) authenticates, which means that he identifies to the server "who" he is, then the server checks "what" operations this identity is allowed to perform, and accordingly admits or rejects the request. For example, in web applications authorization typically takes the form if a user is allowed to create, read, update, or delete a resource.

8.1.4 Comparison of security in Monoliths and Microservices

It should come to no surprise that both microservices and monoliths need to implement end-user and service-to-service authentication and authorization, the differences in how those are implemented stems from the sheer number of microservices and all the interconnections between them that need to be protected. Monoliths have fewer interconnections and are running on a static set of virtual machines, which makes IPs a good source of identity, and as such it is used in certificates for authentication. Authorization is either implemented in a reverse proxy or the application.

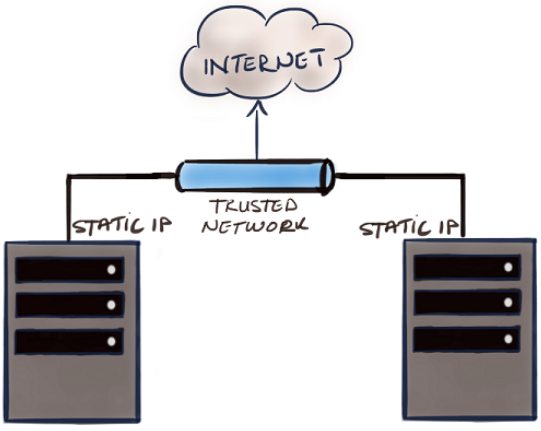


Figure 8.2 The monolithic application running on-premises with static IPs

Microservices on the other hand easily grow into the hundreds and thousands of services, which makes operating the services in a static environment unmaintainable. For this reason, dynamic environments are utilized such as cloud computing and container orchestration, where services are scheduled into numerous VMs and are short-lived. This makes traditional methods such as

the IP a nonreliable source of identity. To put salt into the wound, the services are not necessarily run within the same network either and could span different cloud providers and even run on-premises.

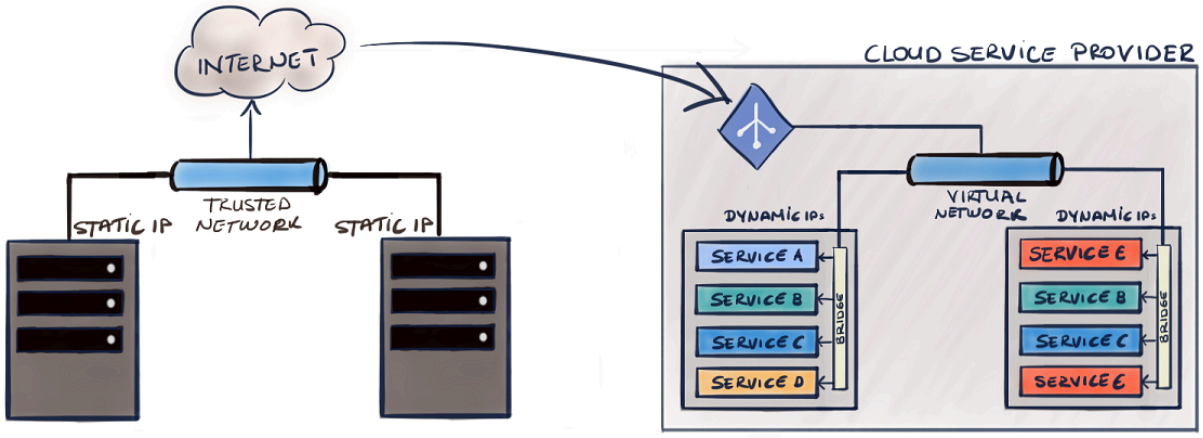


Figure 8.3 Microservices running on the cloud and on-premises with many interconnections

To resolve the challenges of providing Identity in high dynamic and heterogeneous environments Istio uses the SPIFFE specification.

8.2 SPIFFE - Secure Production Identity Framework for Everyone

SPIFFE is a set of open source-standards for providing identity to workloads in highly dynamic and heterogeneous environments. To issue and bootstrap identity SPIFFE defines the following specifications:

- The SPIFFE ID that uniquely identifies a service within a mesh
- The Workload Endpoint bootstraps the identity of a workload
- The Workload API signs and issues the certificate containing the SPIFFE ID
- The SPIFFE Verifiable Identity Document the certificate issued by the Workload API

Those specifications define the process of issuing the identity to a workload, defined in the SPIFFE ID format; how to encode it in a SPIFFE Verifiable Identity Document; and how the Control Plane component (Workload API) and data plane component (Workload Endpoint) work together to verify, assign and validate the identity of a workload.

As those specifications are implemented by Istio, they warrant a deeper investigation.

8.2.1 SPIFFE ID - Workload Identity

SPIFFE Identity is an RFC 3986 compliant URI composed in the following format `spiffe://trust-domain/path`. The two variables here are:

- The `trust-domain` which represents the issuer of identity such as an individual or

organization and
- The `path` that uniquely identifies a workload within the trust domain.

The details on how the path identifies the workload are left open-ended and can be decided by the implementer of the SPIFFE specification, in this chapter we'll see how Istio uses Kubernetes service accounts for defining the path that identifies the workload.

8.2.2 Workload API

Represents the control plane component of the SPIFFE specification that exposes endpoints for workloads to fetch digital certificates that define their identity, in a format known as the SPIFFE Verifiable Identity Document (SVID).

The Workload APIs two main functions are:

- Issuing certificates to workloads, to do so it has the Certificate Authority private key for signing certificate signing requests made by the workloads and
- Exposing an API to make its features available to Workload Endpoints

The specification sets a restriction on workloads. Those must not possess secrets or other information that defines their identity. As otherwise, the system is easily exploitable by a malicious user who gets access to those secrets. As a consequence of the restriction, workloads lack a means of authentication and therefore cannot initiate secure communication with the Workload API. To resolve this SPIFFE defines the Workload Endpoint specification which represents the data plane component and performs all activities to bootstrap the identity of a workload, such as initiating a secure communication with the Workload API and fetching SVIDs without being susceptible to eavesdropping or man-in-the-middle attacks.

8.2.3 Workload Endpoint

The Workload Endpoint represents the data plane component of the SPIFFE specification. It is in the proximity of every workload and provides the following functionalities:

- Workload Attestation is the process by which the workload endpoint verifies the identity of a workload. Using methods such as Kernel introspection or orchestrator interrogation.
- Workload API exposure initiates and maintains a secure communication to the Workload API so that SVIDs are fetched and rotated, and exposes those to the Workload using secure inter-process communication.

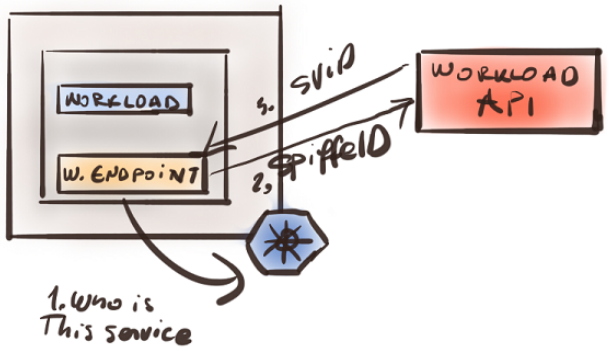


Figure 8.4 Issuing an identity for a workload

In Figure 8.4, we can see an overview of the steps needed to issue identity to workloads:

1. The workload endpoint verifies the identity of the workload (Workload Attestation) and forms the SPIFFE ID
2. The workload endpoint submits the SPIFFE ID (encoded in a Certificate Signing Request) to the Workload API
3. Receives the signed certificate which represents the SPIFFE Verifiable Identity Document and makes it available to the Workload

8.2.4 SPIFFE Verifiable Identity Document

SVIDs are documents that represent a verifiable workload identity. Being verifiable is the most important property as otherwise, the receiving party could not trust the identity of the workload. The specification defines two types of documents X.509 certificates and JWT tokens that meet the criteria to represent SVIDs as both are composed of the following components:

- The SPIFFE ID which represents the workload identity
- A valid signature to ensure that the SPIFFE ID is not tampered with and optionally
- A public key to build secure communication channels between workloads

Istio implements X.509 certificates for SVIDs. It does so by encoding the SPIFFE ID as a uniform resource identifier in the subject alternative name extensions. Using X.509 certificates has the additional benefit that workloads can mutually authenticate and encrypt traffic between each other.

NOTE **Required steps for a workload to receive a SVID**

1. The Workload Endpoint using Workload Attestation verifies the integrity of the workload and creates a Certificate Signing Request (CSR) using the SPIFFE ID.
2. The Workload Endpoint submits the CSR to the Workload API for signing.
3. The Workload API signs the CSR and responds with a digitally signed certificate that has the SPIFFE ID associated with the URI extension of the subject alternative name, this certificate is the SPIFFE Verifiable Identity Document.
4. The SVID is made available to the workload using the Unix Domain Socket, which is a secure inter-process communication socket.

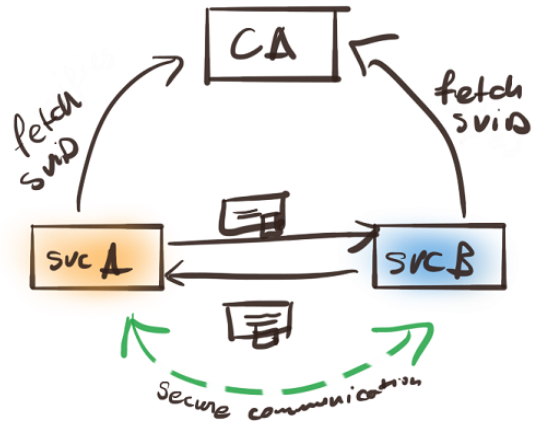


Figure 8.5 Workloads fetching their SVIDs and initiating secure communication

By implementing the SPIFFE specification, Istio automatically ensures that all workloads have their identity provisioned and receive certificates as proof of their identity. Those certificates are used for mutual authentication and to encrypt all service-to-service communication. Hence this feature is called Auto mTLS.

8.2.5 How Istio implements SPIFFE

In the SPIFFE the two components that work together to provide workloads with identity are:

- The Workload Endpoint bootstrapping identity and
- The Workload API issuing certificates

In Istio the Workload Endpoint specification is implemented by the Istio Proxy, as it is in proximity to the workloads it bootstraps the identity and fetches certificates from Istio CA which is the component of Istiod and implements the Workload API specification.

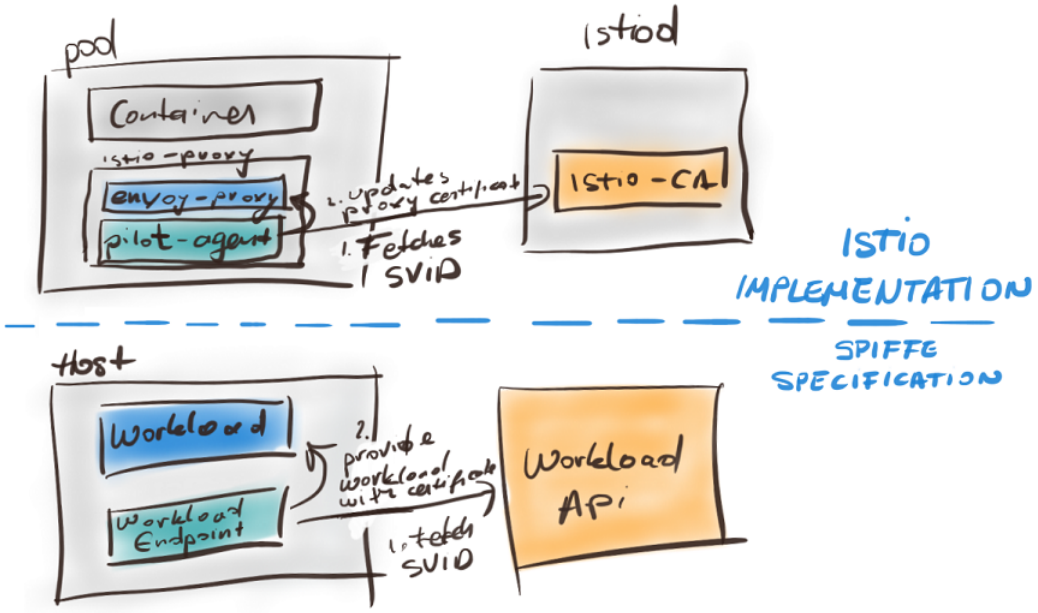


Figure 8.6 Mapping of Istio components to the Spiffe specification

Mapping the SPIFFE components to Istio's implementation

- The Workload Endpoint is implemented by the Pilot-Agent that performs identity bootstrapping
- The Workload API is implemented by Istio CA that issues certificates
- The Workload for whom the identity is issued in Istio is the service proxy

This is the high level of how Istio implements SPIFFE, but let's take the extra mile and see the process step by step, which will ensure that we understand this process and that it sticks with us.

8.2.6 Step by step bootstrapping of Workload Identity

Every pod initialized in Kubernetes by default has a secret mounted in the following path `/var/run/secrets/kubernetes.io/serviceaccount/`.

This secret contains all the data needed to securely talk to the Kubernetes API server, those being

- The ca.crt for validating issued certificates by Kubernetes API server
- The namespace representing where the pod is located and
- The token contains a set of claims for the service account representing the pod.

For the Identity bootstrapping process the most important element here is the token, which was issued by Kubernetes API and its payload cannot be modified as otherwise, it would fail the signature validation. The payload contains data that identifies the application:

Listing 8.1 The payload of the service account token

```
{
  "iss": "kubernetes/serviceaccount",
  "kubernetes.io/serviceaccount/namespace": "istioinaction",
  "kubernetes.io/serviceaccount/secret.name": "default-token-jl68q",
  "kubernetes.io/serviceaccount/service-account.name": "default",
  "kubernetes.io/serviceaccount/service-account.uid": "074055d3-05ca-4968-943a-598b90d1072c",
  "sub": "system:serviceaccount:istioinaction:default"
}
```

The pilot-agent decodes the token and uses the payload data to create the SPIFFE ID `spiffe://cluster.local/ns/istioinaction/sa/default` that is used in the Certificate Signing Request (CSR) as a Subject Alternative Name extension of type URI. Both the token and the CSR are sent in the request to the Istio CA to get an issued certificate for the CSR.

The Istio CA before signing the CSR validates that the token was issued by the Kubernetes API using the TokenReview API [2], on a successful validation the CSR is signed and the resulting certificate is returned to the Pilot-agent.

The Pilot-agent uses the Secrets Discovery Service to forward the certificate and the key to the Envoy proxy, which marks the end of the Identity bootstrapping process. As now the proxy can identify itself to clients, and initiate a mutually authenticated connections.

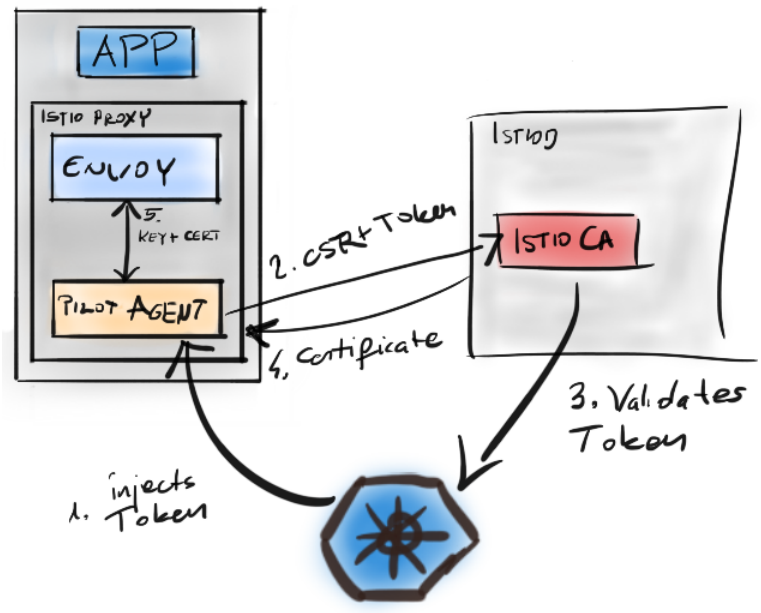


Figure 8.7 Issuing a SVID in Kubernetes with Istio

Elaboration of the points seen in Figure 8.7:

1. Service account token is assigned to the Istio Proxy container
2. The token and a Certificate Signing Request are sent to Istiod

3. Istiod validates the token using the Kubernetes Token Review API
4. On success, it signs the certificate and provides it as a response
5. The Pilot Agent uses the Secrets Discovery Service of Envoy to configure it to use the certificate containing the identity.

And that's the whole process of how Istio implements the SPIFFE specification to provision workload identity. This process is done automatically for every workload that has the Istio-proxy sidecar injected. And having identities provisioned enables workloads to mutually authenticate and encrypt traffic between each other using mTLS. Putting the two keywords together we get "Auto mTLS".

8.3 Auto mTLS in Action

It's comforting to know that all service-to-service traffic for workloads containing the service proxy is authenticated and encrypted, and this is enabled by default with the Istio installation. Having an automated process that issues certificates, validates, and rotates them is very important, as historically this process being managed by humans was error-prone, and security is not where we'd like to have errors. All of this makes the implementation of Istio very desirable for enterprises that are dealing with highly-sensitive data.

To simplify the adoption of mutual authentication Istio by default allows unauthenticated traffic from workloads that lack a service proxy. In continuation, we'll refer to workloads without a service proxy as legacy workloads, and we'll see how to permit or deny clear-text requests.

8.3.1 Reset our workspace

Before we get started, let's clean up our environment so we can start from the same clean slate, by simply executing the commands below:

```
$  make chapter8-cleanup
$  kubectl create namespace istioinaction
```

8.3.2 Setting up the environment

To demonstrate the capabilities of mutual TLS we'll set up three services as shown in Figure 8.8. The API Gateway and the Catalog services are already familiar to us, we added the sleep service, which will represent a legacy workload that cannot mutually authenticate.

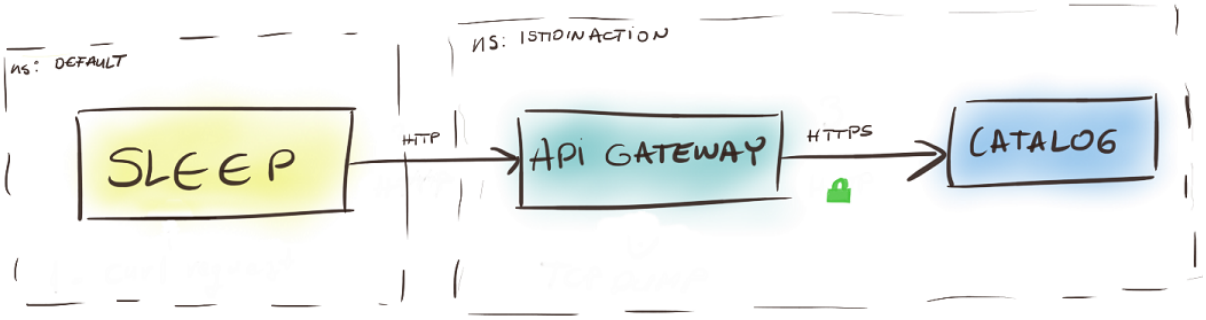


Figure 8.8 The three workloads we'll setup

To install the services execute the commands below:

```
$  kubectl -n istioinaction apply -f <(istioctl kube-inject \
     -f services/catalog/kubernetes/catalog.yaml) ❶
$  kubectl -n istioinaction apply -f <(istioctl kube-inject \
     -f services/apigateway/kubernetes/apigateway.yaml) ❶
$  kubectl -n default apply -f chapters/chapter8/sleep.yaml ❷
```

❶ Workloads

❷ Legacy workloads

Test that services are correctly set up by executing a clear-text request from the sleep legacy to the API Gateway workload:

```
$  kubectl -n default exec $(kubectl -n default get pod -l app=sleep \
     -o jsonpath={.items..metadata.name}) \
     -c sleep \
     -- curl -s apigateway.istioinaction.svc.cluster.local/api/catalog \
     -o /dev/null -w "%{http_code}"

200
```

The successful response shows that the services are set up correctly and that the API Gateway accepted a plain text request from the sleep service. By default, Istio permits clear-text requests, to enable teams to gradually adopt the service mesh, without causing outages while some legacy workloads are migrated and others not. Furthermore, this is configurable via the PeerAuthentication custom resource.

8.3.3 Understanding Istio's Peer Authentication resource

The Peer Authentication custom resource enables configuration of workloads to either strictly require peer authentication, or to be permissive and accept traffic from unauthenticated resources. The former is achieved with the STRICT mutual authentication mode and the latter with the PERMISSIVE mutual authentication mode. The mutual authentication mode can be configured in different scopes:

- **Mesh-wide peer authentication policies** apply to all workloads of the service mesh
- **Namespace-wide peer authentication policies** apply to all workloads within a

namespace

- **Workload specific peer authentication policies** apply to all workloads that match the selector specified in the policy.

Let's get introduced to all the different scopes by taking practical examples. Let's begin by applying a STRICT mode to every workload within the service mesh. This will cause requests from the Sleep legacy workload to the API Gateway workload to fail.

DENYING ALL NON AUTHENTICATED TRAFFIC USING A MESH-WIDE PEERAUTHENTICATION POLICY

To create a mesh-wide peer authentication policy it must be in the same namespace as the Istio installation and it must be named `"default"`. If you followed the instructions in the book your Istio installation will be in the `istio-system` namespace. With the definition below we'll apply a mesh-wide policy that permits only mutually authenticated (mTLS) traffic into workloads:

```
apiVersion: "security.istio.io/v1beta1"
kind: "PeerAuthentication"
metadata:
  name: "default"                ❶
  namespace: "istio-system"      ❷
spec:
  mtls:
    mode: STRICT                 ❸
```

❶ Mesh wide policies must be named "default"

❷ Istio installation namespace.

❸ Mutual TLS mode

Apply the policy to the cluster by executing:

```
$  kubectl apply -f chapters/chapter8/meshwide-strict-peer-authn.yaml
```

And verify that clear-text requests from the sleep service are not permitted anymore:

```
$  kubectl -n default exec $(kubectl -n default get pod -l app=sleep \
     -o jsonpath={.items..metadata.name}) \
     -c sleep -- curl \
     -s apigateway.istioinaction.svc.cluster.local/api/catalog

command terminated with exit code 56
```

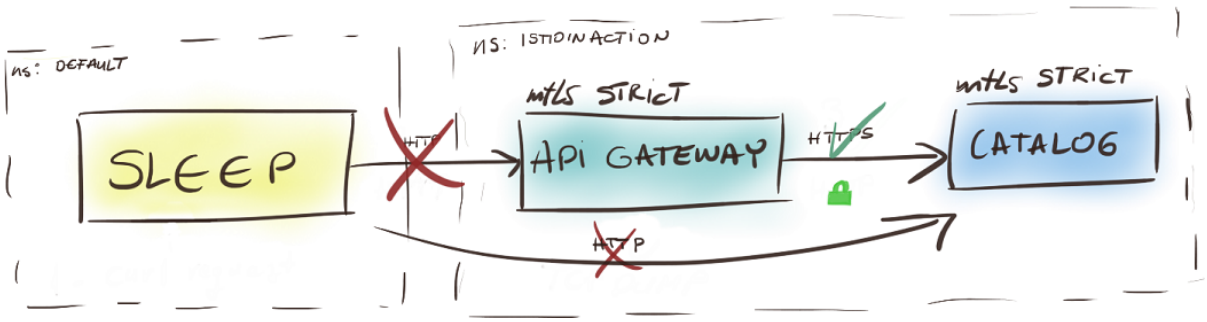


Figure 8.9 Workloads are configured to reject non-mutually authenticated traffic

This verifies that the clear-text request was not permitted. Having a STRICT mutual authentication requirement is a great default, but for ongoing projects, such a drastic change is not feasible as coordination between multiple teams is needed to migrate their workloads. A better approach is to gradually increase the restrictions that you put in place and allow a timeframe in which teams can migrate their services to become part of the service mesh. The PERMISSIVE mutual authentication does just that, as it permits workloads to accept both encrypted and clear-text requests.

PERMITTING NON MUTUALLY AUTHENTICATED TRAFFIC USING THE NAMESPACE-WIDE PEER AUTHENTICATION POLICY

Using a namespace wide policy we can override the mesh wide policy and apply more specific Peer Authentication requirements for workloads within a namespace. With the Peer Authentication resource below we ensure that workloads within the `istioinaction` namespace accept clear-text traffic from legacy workloads:

```
apiVersion: "security.istio.io/v1beta1"
kind: "PeerAuthentication"
metadata:
  name: "default"                 ❶
  namespace: "istioinaction"      ❷
spec:
  mtls:
    mode: PERMISSIVE              ❸
```

❶ To apply to the entire namespace the name has to be “default”

❷ Specifies the namespace to apply the policy

❸ PERMISSIVE allows HTTP traffic

But we definitely can do better. We want to allow unauthenticated traffic from the Sleep workload to the API Gateway, but we can still keep STRICT mutual authentication requirements for the Catalog workload. This would keep the attack surface area smaller in cases when our network security is compromised.

APPLYING WORKLOAD SPECIFIC PEERAUTHENTICATION POLICIES

To target only the API Gateway we updated the earlier peer authentication policy to specify the selector which needs to be matched:

```
apiVersion: "security.istio.io/v1beta1"
kind: "PeerAuthentication"
metadata:
  name: "apigateway"
  namespace: "istioinaction"
spec:
  selector:
    matchLabels:
      app: "apigateway"       ❶
  mtls:
    mode: PERMISSIVE
```

❶ Workloads matching the label will use PERMISSIVE mode of mTLS

This way the mesh wide strict mutual authentication policy is overridden for the API Gateway workload, but not for the Catalog workload. Apply the policy to the cluster by executing the command below:

```
$  kubectl apply -f chapters/chapter8/workload-permissive-peer-authn.yaml
```

And verify that clear-text requests are accepted in the API Gateway:

```
$  kubectl -n default exec $(kubectl -n default get pod -l app=sleep \
     -o jsonpath={.items..metadata.name}) \
     -c sleep -- curl \
     -s apigateway.istioinaction.svc.cluster.local/api/catalog
```

Let's additionally verify that clear-text requests to the catalog service are rejected by triggering a request from the legacy workload to the Catalog workload:

```
$  kubectl -n default exec $(kubectl -n default get pod -l app=sleep \
     -o jsonpath={.items..metadata.name}) \
     -c sleep -- curl \
     -s catalog.istioinaction.svc.cluster.local/items

command terminated with exit code 56
```

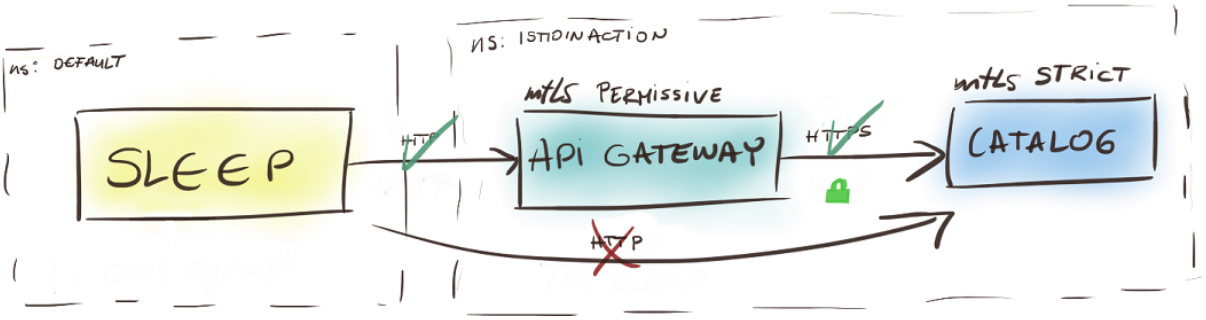


Figure 8.10 API Gateway accepts HTTP traffic. Catalog service requires mutual authentication.

With these two policies we ensured that using permissive peer authentication the Sleep legacy workload can make plain text requests to the API Gateway, meanwhile, the Catalog service is strictly requiring mutual authentication and doesn't accept unencrypted traffic, as shown in Figure 8.8.

> **NOTE** We have a good understanding of how Istio works by now, but let's hit the point home once more. The creation of the PeerAuthentication custom resource is listened to by Istiod, which transforms the custom resource into Envoy specific configuration and applies it to the service proxies using the Listener Discovery Services (LDS). The configured policies are evaluated for every incoming request.

DIFFERENT MUTUAL AUTHENTICATION MODES

Most of the time you will be using either the STRICT or PERMISSIVE modes. But there are an additional of two modes:

- **UNSET** - Inherit the peer authentication policy of the parent
- **DISABLE** - Target your application directly by completely bypassing the Istio-Proxy.

Good! We put the Peer Authentication custom resource to use, which allows us to specify the type of traffic to tunnel to the workload, such as mutually authenticated traffic, clear-text traffic, and as well as bypassing the service proxy and forward requests directly to the application. In the next section let's verify that the traffic is encrypted when using mutual TLS.

EAVESDROPPING SERVICE-TO-SERVICE TRAFFIC USING TCPDUMP

The Istio proxy comes preinstalled with the tcpdump command line utility. This utility captures and analyzes network traffic going through your system. For security purposes, it requires privileged permissions and by default, those are turned off. To turn on privileged permissions, update the Istio installation by setting the property `values.global.proxy.privileged=true` using istioctl:

```
$  istioctl manifest apply --set profile=demo \
     --set values.global.proxy.privileged=true
```

Verify that the above command updated the template1 that will be used to inject proxies by doing a kube diff on the API Gateway installation:

Listing 8.2 Fields that are changed in the Istio Proxy injection template

```
$  kubectl -n istioinaction diff -f <(istioctl kube-inject \
     -f services/apigateway/kubernetes/apigateway.yaml)

  image: docker.io/istio/proxyv2:1.5.1
  # shortened for readability
  securityContext:
-   allowPrivilegeEscalation: false
+   allowPrivilegeEscalation: true
    capabilities:
      drop:
      - ALL
-   privileged: false
+   privileged: true
```

As injecting the proxy with elevated privileges is our intention, let's apply it to the cluster:

```
$  kubectl -n istioinaction apply -f <(istioctl kube-inject \
     -f services/apigateway/kubernetes/apigateway.yaml)
```

TIP Elevated permissions on the service proxy provide a vector of attack for malicious users. DO NOT install Istio with elevated proxies in production clusters. For quick debugging, of one service it is recommended to change the fields (shown in the Listing 8.2.) by editing the deployment, i.e. `kube edit`.

As soon as the new API Gateway pods are ready, sniff out the pod traffic by executing the tcpdump command below:

```
$  kubectl -n istioinaction exec $(kubectl -n istioinaction get pod \
     -l app=apigateway -o jsonpath={.items..metadata.name}) \
     -c istio-proxy \
     -- sudo tcpdump -l --immediate-mode -vv -s 0 \
     '(((ip[2:2] - ((ip[0]&0xf)<<2)) - ((tcp[12]&0xf0)>>2)) != 0)'
```

Open a second terminal to trigger a request from the Sleep legacy workload to the API Gateway:

```
$  kubectl -n default exec $(kubectl -n default get pod -l app=sleep \
     -o jsonpath={.items..metadata.name}) \
     -c sleep \
     -- curl -s apigateway.istioinaction.svc.cluster.local/api/catalog

    10-1-0-65.sleep.istioinaction.svc.cluster.local.53058 > apigateway-56d5f78bf5-jd6n6.8080
# output cleaned up
        GET /api/catalog HTTP/1.1
        Host: apigateway
        User-Agent: curl/7.64.0
        Accept: */*
```

The output shows that headers of the request from the Sleep workload to the API Gateway are in clear-text. Looking more in the output in the terminal we can see the entire JSON body, eek! Meanwhile, the traffic from the API Gateway to the Catalog workload is encrypted and secure as

shown below:

```
    apigateway-56d5f78bf5-jd6n6.37390 > 10-1-0-67.catalog.istioinaction.svc.cluster.local.3000:
    Flags [P.], cksum 0x158d (incorrect -> 0x8bb5), seq 117740199:117740421, ack 4263845226,
    win 229, options [nop,nop,TS val 4158075820 ecr 1546201650], length 222
13:18:40.437146 IP (tos 0x0, ttl 64, id 28644, offset 0, flags [DF], proto UDP (17), length 68)
```

This verifies that the traffic between mutually authenticated workloads is encrypted in transit. Furthermore, it showcases how insecure it is to have legacy services within the service mesh, as data to and from it are in clear-text and easy to be sniffed out (by your cloud provider, government, etc) while it is in transit through multiple networking devices.

VERIFY THAT WORKLOAD IDENTITIES ARE TIED TO THE WORKLOAD SERVICE ACCOUNT

Before we close the mutual authentication section, let's check that the issued certificates are valid SVID Documents and have the SPIFFE ID encoded in them, and that it matches the workload service account. Use the openssl command utility to checkout the contents of the X.509 certificate of the Catalog workload:

```
$  kubectl -n istioinaction exec $(make apigateway-pod) -c istio-proxy \
     -- openssl s_client -showcerts \
     -connect catalog.istioinaction.svc.cluster.local:80 \
     -CAfile /var/run/secrets/istio/root-cert.pem | \
     openssl x509 -in /dev/stdin -text -noout
```

With this convoluted command we are querying the certificate of the Catalog service and redirecting its output to the openssl command that renders it in human readable format. On the output we see that the received certificate contains the SPIFFE ID set as the uniform resource identifier in the subject alternative name extensions, and it is set to the workloads service account:

```
# shortened for brevity
X509v3 Subject Alternative Name: critical
     URI:spiffe://cluster.local/ns/istioinaction/sa/catalog
```

Using the openssl verify utility, let's make sure that the contents of the X.509 SVID are valid by checking it's signature against the CA root certificate which is mounted to the istio-proxy container in the following path `/var/run/secrets/istio/root-cert.pem`. Get a shell into the running pod by executing:

```
$  kubectl -n istioinaction exec --stdin --tty \
     $(make apigateway-pod) -c istio-proxy -- /bin/bash
```

And verify the certificate:

```
$  openssl verify -CAfile /var/run/secrets/istio/root-cert.pem \
     <(openssl s_client -connect \
     catalog.istioinaction.svc.cluster.local:80 -showcerts 2>/dev/null)

/dev/fd/63: OK
```

On successful validation, an `OK` message is displayed in the command output. Great! Exit from the remote shell by typing `exit`.

Now as we verified all the components that facilitate peer to peer authentication we are assured that the issued identities are verifiable. Having verifiable identities is the precursor to being able to control access. In other words, as we know the identity of a workload, then we can proceed to define the operations that it is allowed to perform.

8.4 Authorizing Service to service traffic

Authorization is the process that defines whether an authenticated user is allowed to perform an operation, such as accessing a resource, deleting it etc. Policies are formed in conjunction with the authenticated user (the "who") and authorization (the "what"), and simply stated the goal of policies is to define who can do what.

Istio provides the AuthorizationPolicy custom resource which exposes a simple API to define either mesh-wide, namespace, or workload-specific policies for workloads within a service mesh. This facilitates features such as workload-to-workload and end-user-to-workload authorization, which are of paramount importance to improve security by reducing the scope of attack malicious users have if they get hold of the identity of one of the services.

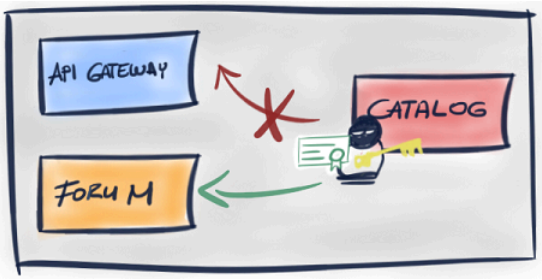


Figure 8.11 Reducing attack scope to only what the stolen identity was authorized

In the next section let's check out how authorization is implemented in Istio.

8.4.1 Understanding Authorization in Istio

The service proxy is the "authorization engine" as it contains all the policies for determining if one request must be denied or allowed. This makes authorization in Istio extremely fast as decisions are taken directly from the proxy. The proxies are configured with the AuthorizationPolicy custom resource, that when applied to the cluster is picked up by Istiod and configures the service proxies of the targeted workloads.

PROPERTIES OF AN AUTHORIZATION POLICY

The AuthorizationPolicy custom resource provides a simple API for all authorization features. It can be applied to the entire mesh, to a namespace, or using workload selectors can be applied only to a set of matching workloads. The resource specification provides three fields:

- The **selector** field defines the subset of workloads to which the policy applies
- The **rules** field defines a list of rules that if any is matched this policy will be activated for the request being processed
- The **action** field defines if the request is to be allowed or denied. Applies only if the policy is activated due to one of the rules matching.

The rules property is more complex and warrants more investigation.

UNDERSTANDING AUTHORIZATION POLICY RULES

Authorization Policy Rules is a list of rules, where each specifies the source of the connection and the operation that will activate the rule. Authorization Policies are activated only if one of its rules matches the source and the operation. In that case, the policy is activated and the connection will either be allowed or denied according to the action property. The fields of a single rule are:

- The **from** field specifies the source of the request and it can be one of the following types:
  - **principals**: a list of source identities (i.e. SPIFFE ID), as such mTLS needs to be enabled and the source needs to be an Istio workload. There is a negated property notPrincipals, which applies if the request is not from a principal.
  - **namespaces**: a list of namespaces that matches the source namespace, value is retrieved from the SVID of the peer, as such mTLS needs to be enabled.
  - **ipBlocks**: list of single IPs or CIDRs that matches the source ip.
- The **to** field specifies the operations of the request, such as the host of the request, the method of the request, etc.
- The **when** field specifies a list of conditions that need to be met after the rule has matched.

> **NOTE** A full list of the all the properties for Authorization Policies are documented by Istio at istio.io/latest/docs/reference/config/security/authorization-policy/

All of this sounds more complex than it is, let's put these concepts into action.

8.4.2 Setting up the workspace

We'll continue with the same state as left in the earlier sections if you've made modifications and want a fresh start then proceed to execute the command below:

```
$  make chapter8-setup-authn
```

Let's recapitulate the workloads that we have running (see Fig. 8.10):

- The Sleep legacy workload is deployed in the `default` namespace and used to trigger clear-text HTTP requests
- The API Gateway workload is deployed in the `istioinaction` namespace and accepts unauthenticated requests from the default namespace
- The Catalog workload is deployed in the `istioinaction` namespace and accepts requests only from authenticated services within its namespace

8.4.3 Behavioral differences when an Authorization Policy is applied to a workload

There is a design decision on how authorization policies affect workloads that could come as a surprise, and waste many hours of debugging. Once learned though (or bitten by it) it's easy to remember. If one or more "allow" authorization policies are applied to a workload, access to that workload is denied by default, when not explicitly allowed by one of the authorization policies.

Let's illustrate with an example if we create an authorization policy that allows requests to the API Gateway for the path `/api/catalog*` as shown below:

```
apiVersion: "security.istio.io/v1beta1"
kind: "AuthorizationPolicy"
metadata:
  name: "allow-catalog-requests-in-api-gw"
  namespace: istioinaction                  ❶
spec:
  selector:
    matchLabels:
      app: apigateway                       ❷
  rules:
  - to:
    - operation:
        paths: ["/api/catalog*"]            ❸
  action: ALLOW                             ❹
```

❶ Applies to workloads within the istioinaction namespace

❷ Selector reduces the workloads affected only to the ones matching the label app=apigateway

❸ An expression that matches requests prefixed with the path /api/catalog

❹ If one of the rules is matched the request is allowed, the default value of action is ALLOW

Due to its simplicity, instead of applying this authorization policy to the cluster, let's mentally play out the outcomes for the following two requests:

```
$  kubectl -n default exec $(make sleep-pod) \
      -c sleep -- curl \
      -sSL apigateway.istioinaction.svc.cluster.local/api/catalog      ❶
```

```
$   kubectl -n default exec $(make sleep-pod) \
      -c sleep -- curl \
      -sSL apigateway.istioinaction.svc.cluster.local/hello/world      ❷
```

❶ The request on the path `/api/catalog` is allowed as the authorization policy matches the path and the action allows the request.

❷ The request on the path `/hello/world` is denied as no authorization policy explicitly allows the request.

To simplify the thought process, and not have to question yourself for every service "Is any authorization policy applied to workload X?" it's recommended to add a "deny" catch-all policy, which is activated when no other policy applies to an incoming connection. This changes the thought process to "We by default deny requests if not explicitly specified otherwise", as it handles the majority of deny requests and you need only to specify the connections that are allowed.

8.4.4 Denying all requests by default with a Catch all policy

To increase security and in the process simplify our thought process in the future let's define a mesh-wide policy that denies all requests if not explicitly allowed by an authorization policy, in other words, a catch-all deny policy, achieved with the AuthorizationPolicy below:

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
 name: deny-all
 namespace: istio-system    ❶
spec: {}                    ❷
```

❶ Policies in the Istio installation namespace target workloads mesh-wide

❷ Policies without any defined rule deny every request

Let's apply the deny-all policy to the cluster:

```
$  kubectl apply -f chapters/chapter8/policy-deny-all-mesh.yaml
```

Wait for a short period until the proxies receive the new configuration and trigger a request from the sleep service to verify that our request fails authorization:

```
$  kubectl -n default exec $(make sleep-pod) \
     -c sleep -- curl \
     -sSL apigateway.istioinaction.svc.cluster.local/api/catalog

RBAC: access denied
```

From the output, we can see that the deny-all authorization policy has kicked into effect and denied that the request with the message `RBAC: access denied`.

NOTE Just as the lack of any rule is an indicator that no requests are allowed, the opposite, the presence of an empty rule means that all requests are allowed. For an example check the Appendix XX "Commonly used Authorization Policies"

8.4.5 Allowing requests originating from a single namespace

To allow requests from the Sleep workload to the API Gateway workload, we'd come up with a quick solution to allow all unauthenticated traffic from the default namespace, where the sleep service resides. Achieved with the AuthorizationPolicy below:

```
apiVersion: "security.istio.io/v1beta1"
kind: "AuthorizationPolicy"
metadata:
 name: "apigateway-allow-view-default-ns"
 namespace: istioinaction                ❶
spec:
  rules:
  - from:                                 ❷
    - source:
        namespaces: ["default"]
    to:                                   ❸
    - operation:
        methods: ["GET"]
```

❶ Workloads targeted within the namespace istioinaction

❷ The rule matches only from requests sourcing from the default namespace

❸ The rule matches only for operations of the GET HTTP method.

But in our case this won't work! The Sleep service is a legacy workload and cannot authenticate itself. As such the API Gateway proxy cannot validate the authenticity of the namespace that the request originates from.

To solve this we can either:

1. Inject a service proxy to the sleep service
2. Allow non-authenticated requests in the API Gateway

The recommended approach is to inject the service proxy in the Sleep workload, which would bootstrap the identity and perform mutual authentication with other workloads, enabling those to verify the source of the request and as such the namespace. But for demonstration purposes let's suppose that the first approach is not possible (the entire team is on holiday). We are forced to take the second and less secure approach and allow non-authenticated requests. But we'll limit the attack surface area to only the API Gateway.

8.4.6 Allowing requests from non-authenticated legacy workloads

To allow requests from non-authenticated workloads we need to drop the `from` field entirely, as shown below:

```
apiVersion: "security.istio.io/v1beta1"
kind: "AuthorizationPolicy"
metadata:
 name: "apigateway-allow-unauthenticated-view"
 namespace: istioinaction
spec:
  selector:
    matchLabels:
      app: apigateway
  rules:
  - to:
    - operation:
        methods: ["GET"]
```

To apply this policy only to the API Gateway we added the selector `app: apigateway`, this way the Catalog service will still require mutual authentication. Apply to the cluster by executing:

```
$  kubectl apply -f chapters/chapter8/allow-unauthenticated-view-default-ns.yaml
```

Retrying the request from the Sleep service to the API Gateway we get the following error response `error calling Catalog service`. This is an application error, not an Istio error. API Gateway received the request from the Sleep service, but the request from the API Gateway to the Catalog service was denied by the mesh-wide `deny-all` policy. Remember why we added this `deny-all` policy? It simplifies the thought process, we didn't add an allow policy to admit requests in the Catalog service and hence it was rejected. Let's fix this in the next section.

8.4.7 Allowing requests from a single principal

As both the API Gateway and the Catalog workloads can mutually authenticate, we can specify a policy that allows incoming requests to the Catalog workload only if it originates from the API Gateway workload, as shown below:

```
apiVersion: "security.istio.io/v1beta1"
kind: "AuthorizationPolicy"
metadata:
 name: "catalog-viewer"
 namespace: istioinaction
spec:
 selector:
   matchLabels:
     app: catalog
 rules:
 - from:
   - source:
       principals: ["cluster.local/ns/istioinaction/sa/apigateway"]     ❶
   to:
   - operation:
       methods: ["GET"]
```

❶ Allow requests only if they can prove they have the identity of the `apigateway` workload.

Apply to the cluster by executing:

```
$  kubectl apply -f chapters/chapter8/catalog-viewer-policy.yaml
```

Now if we try once more we will see that our request successfully reaches the Catalog workload:

```
$   kubectl -n default exec $(make sleep-pod) \
      -c sleep -- curl \
      -sSL apigateway.istioinaction.svc.cluster.local/api/catalog

[
  {
    "id": 0,
    "color": "teal",
    "department": "Clothing",
    "name": "Small Metal Shoes",
    "price": "232.00"
  }
]
```

But more importantly, we have strict authorization policies in place that if the identity of a workload is stolen it would limit the damage to the smallest scope possible.

8.4.8 Conditional matching of policies

Using the when property of Authorization Policies we can define a condition when a policy should be triggered or not. In the upcoming section XX “Authorize end-users based on JWT claims” we list the example below:

```
apiVersion: "security.istio.io/v1beta1"
kind: "AuthorizationPolicy"
metadata:
  name: "allow-mesh-all-ops-admin"
  namespace: istio-system
spec:
  rules:
    - from:
      - source:
          requestPrincipals: ["auth@istioinaction.io/*"]
      when:
      - key: request.auth.claims[group]                 ❶
        values: ["admin"]                               ❷
```

❶ Specifies the Istio Attribute

❷ Specifies a list of values that has to match

This policy would allow all requests from the request principal `auth@istioinaction.io/\*` but only when the condition is fulfilled that he has the group admin as a claim. Alternatively, we

can use the `notValues` property to define all the values for which this policy should not be applied. Let's illustrate with an example. The when condition below allows all requests that are not in the users group:

```
when:
- key: request.auth.claims[group]
  notValues: ["users"]
```

A full list for Istio Attributes that can be used in conditions can be found on the following link istio.io/latest/docs/reference/config/security/conditions/

> **NOTE** The difference between Principal and Request Principal is that the former is for Peer to Peer Authentication, and the latter for end-user Request Authentication. This will be clarified in the sections to come.

8.4.9 Understanding value match expressions

As we saw in the earlier examples values do not always have to exactly match. Istio supports simple match expressions to make rules more versatile:

- **Exact matching** of values, for example, `GET` matches only the exact value
- **Prefix matching** of values, for example, `/api/catalog*` would match all values starting with that prefix, such as `/api/catalog/1`
- **Suffix matching** of values, for example, `\*.istioinaction.io` would match all of its subdomains, such as `login.istioinaction.io`
- **Presence matching** matches all values and is denoted with `\*`. This specifies that a field must be present, but the value is not important and can be anything.

UNDERSTANDING HOW POLICY RULES ARE EVALUATED

Up to now we observed policy properties in isolation and elaborated how those work. Now let's take a more complex policy and break it down concretely to what requests it would apply to.

```
apiVersion: "security.istio.io/v1beta1"
kind: "AuthorizationPolicy"
metadata:
  name: "allow-mesh-all-ops-admin"
  namespace: istio-system
spec:
  rules:
    - from:                                                    ❶
      - source:
          principals: ["cluster.local/ns/istioinaction/sa/apigateway"]
      - source:
          namespaces: ["default"]
      to:
      - operation:
          methods: ["GET"]
          paths: ["/users*"]
      - operation:
          methods: ["POST"]
          paths: ["/data"]
```

```
      when:
      - key: request.auth.claims[group]
        values: ["beta-tester", "admin", "developer"]
    - to:                                                    ❷
      - operation:
          paths: ["*.html", "*.js", "*.png"]
```

❶ First rule

❷ Second rule

For this authorization policy to apply to a request, either the first rule OR the second rule needs to match. Diving deeper into the cases when a first rule would match:

```
- from:                                  ❶
  - source:
      principals: ["cluster.local/ns/istioinaction/sa/apigateway"]
  - source:
      namespaces: ["default"]
  to:                                    ❷
  - operation:
      methods: ["GET"]
      paths: ["/users*"]
  - operation:
      methods: ["POST"]
      paths: ["/data"]
  when:                                  ❸
  - key: request.auth.claims[group]
    values: ["beta-tester", "admin", "developer"]
```

❶ the sources

❷ the operations

❸ the conditions

For this rule to match a request we need matches in all three properties. Meaning that one source defined in the sources list needs to match with one operation defined in the operations list and finally all of the conditions needs to match. In other words one source defined in “from” is AND-ed with one of the operations defined in “to” and both are AND-ed with all the conditions specified in “when”. Taking a closer look to operations explains what we understand with having one operation match:

```
to:
- operation:             ❶
    methods: ["GET"]         ❷
    paths: ["/users*"]       ❷
- operation:             ❸
    methods: ["POST"]        ❹
    paths: ["/data"]         ❹
```

❶ First operation

❷ Two properties that need to match for the first operation to match

❸ Second operation

❹ Two properties that need to match for the second operation to match

In order for this rule to have an operation match either the first OR the second operation needs to match. For an operation to match all of its properties need to match, meaning that the properties are AND-ed together.

Meanwhile, it is different in the **when** condition. All conditions need to match. Those are AND-ed together.

8.4.10 Understanding the order in which Authorization Policies are evaluated

The complexity of policies always arises when many are applied to a workload, and then it is difficult to understand the order. Many solutions use a priority field to define the order. Istio uses a different approach on how policies are evaluated:

1. The DENY policies are evaluated initally if no deny policy is matched then
2. The ALLOW policies are evaluated, if one matches then the request is allowed, otherwise
3. If a catch-all policy is defined then it is evaluated only when no DENY policies or ALLOW policies were activated by the incoming request and
4. If the workload has “allow” policies assigned to it but nonmatches, the request is denied.

This completes the authentication and authorization for workload-to-workload requests. In the next section, we’ll put in action the end-user authentication and authorization capabilities.

8.5 End-User authentication and authorization

We mentioned briefly that end-user authentication and authorization is supported by Istio when using JWT Tokens. Before diving into the details on how authentication and authorization of requests work, let’s take a brief refresher on JWT Tokens, readers with basic knowledge can skip to the next section.

8.5.1 What is a JSON Web Token?

JSON Web Token (JWT) is a compact claims representation format that is used to securely transmit information between two parties as JSON objects. The secure component comes from the verifiable signature of the JWT Token. JWT Tokens consist of the following three parts:

- Header - composed of the type and the hashing algorithm
- Payload - contains the user claims
- Signature - used to verify the authenticity of the JWT Token

Those three parts, the header, payload, and signature are separated by dots (.) and stored in base 64 URL encoded format, which makes it perfect for usage in HTTP requests.

Let’s checkout the contents of a token located under `chapters/chapter8/enduser/user.jwt` and decode its Payload:

```
$  cat chapters/chapter8/enduser/user.jwt | cut -d '.' -f2 | \
      base64 --decode

{
  "exp": 4743986578,                                    ❶
  "group": "user",                                      ❷
  "iat": 1590386578,                                    ❸
  "iss": "testing@secure.istio.io",                     ❹
  "sub": "9b792b56-7dfa-4e4b-a83f-e20679115d79"         ❺
```

❶ Expiration time after which the claim is not valid

❷ Custom claim specifying the user group

❸ The time the token is issued at

❹ The principal that issued this token

❺ The subject, i.e. a unique identifier for the subject for whom the token was issued

The data above represents the claims of the subject. The claims enable the service to determine the identity and authorization of a client. For example, the token in the listing above belongs to a subject in the user group. Concluded due to the claim `group: user`. This information will be used by the service to decide the level of access for this subject. For claims to be trusted those need to be verifiable.

HOW IS A JWT TOKEN ISSUED AND VALIDATED?

The JWT token is issued by an authentication server which contains a private key for signing the token and a public key for validating it. The public key is known as JSON Web Key Set and is exposed at a well-known endpoint, from which other services retrieve the public key to validate tokens issued by the authentication server.

NOTE **There are multiple solutions to set up an authentication server:**

1. It can be implemented in the application backend framework itself,
2. As a service on its own such as OpenIAM (openiam.com), KeyCloak (keycloak.org), or
3. As an Identity-as-a-Service solution such as Auth0 (auth0.com), Okta (okta.com), etc.

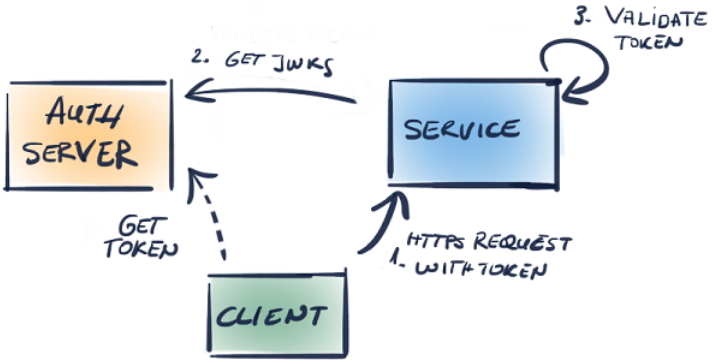


Figure 8.12 Service retrieves JWKS to validate the token presented by the client

The validation is done by decrypting the signature and comparing it with the hash of the token data. If those match the token claims can be trusted.

8.5.2 End-user authentication and authorization at the Ingress Gateway

Istio workloads can be configured to authenticate and authorize end-user requests with JWT Tokens. Although this can be done for all workloads it's preferably done at Istio's Ingress Gateway. This improves performance as invalid requests are rejected early on, but as well enables Istio's to redact the JWT Token from the request so that subsequent services cannot accidentally leak it, which poses risks such as to reply attacks, where it's used to make requests on behalf of the subject.

> **NOTE** **Istio's Token Service**
>
> Istio plans the implementation of the Token Service that would replace end-user tokens with short-lived tokens at Istio's Ingress Gateway. The Token Service generated tokens are valid only within the service mesh and contain all the claims of the end user-token, but due to their short lifespan their leakage is less severe. Currently, work on this service has not started.

SETTING UP THE WORKSPACE

Let's remove all the resources created up to now and start from a fresh environment:

```
$ make chapter8-cleanup
```

Proceed to create the namespace and setup the workloads:

```
$  kubectl create ns istioinaction
$  kubectl -n istioinaction apply -f <(istioctl kube-inject \
     -f services/catalog/kubernetes/catalog.yaml)
$  kubectl -n istioinaction apply -f <(istioctl kube-inject \
     -f services/apigateway/kubernetes/apigateway.yaml)
```

Before setting up authentication and authorization we need to admit traffic into Istio's IngressGateway using a Gateway custom resource. Additionally, a VirtualService is needed to

route the traffic to the API Gateway workload, those resources can be applied by executing the command below:

```
$ kubectl apply -f chapters/chapter8/enduser/ingress-gw-for-apigateway.yaml
gateway.networking.istio.io/apigateway-gateway created
virtualservice.networking.istio.io/apigateway-virtualservice created
```

Now, the workspace is prepared to start trying out the RequestAuthentication resource.

8.5.3 Validating JWT Tokens with RequestAuthentication

The main purpose of the RequestAuthentication custom resource is to validate JWT Tokens and to extract and store the claims of valid tokens into filter metadata, which are used by authorization policies to take actions based on the data. For example, if a request with the claim `group: admin` is validated, this value will be stored as filter metadata which is used by Authorization Policies to allow or deny the request.

There can be three different outcomes based on the end-user requests:

- Requests with valid tokens are admitted into the cluster and its claims are made available to policies in the form of filter metadata
- Requests with invalid tokens are rejected
- Requests without tokens are admitted into the cluster but lack identity, as there are no claims to be stored as filter metadata.

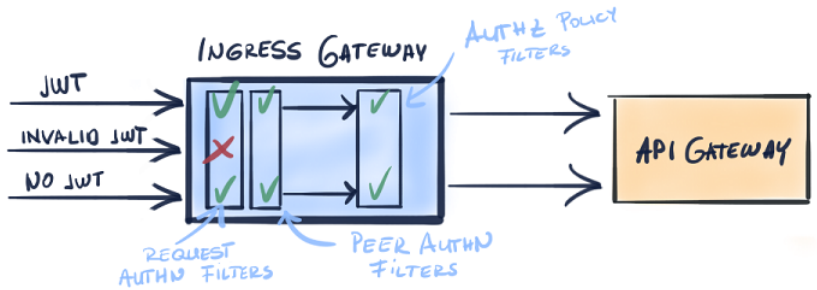


Figure 8.13 Requests passing through envoy filters in order

The difference between the request with a JWT token and the request without, is that the former was validated by the Request Authentication filter and gets the claims stored in its connection filter metadata, meanwhile the later lacks those.

In the next section, we'll create a Request Authentication resource and showcase all the above-mentioned cases with practical examples.

CREATING A REQUEST AUTHENTICATION RESOURCE

The RequestAuthentication resource defined below targets only Istio's IngressGateway. It configures the ingress gateway to validate the tokens that are issued from `auth@istioinaction.io`.

```
apiVersion: "security.istio.io/v1beta1"
kind: "RequestAuthentication"
metadata:
 name: "jwt-token-request-authn"
 namespace: istio-system                    ❶
spec:
  selector:
    matchLabels:
      app: istio-ingressgateway
 jwtRules:
 - issuer: "auth@istioinaction.io"          ❷
   jwks: |                                  ❸
     { "keys": [{"e":"AQAB","kid":"##REDACTED##","kty":"RSA","n":"##REDACTED##"}]}
```

❶ The request authentication is applied to workloads within istioinaction namespace

❷ The token should be issued from `auth@istioinaction.io`

❸ The token should be verifiable by the JWKS.

Apply the resource to the cluster by executing:

```
$ kubectl apply -f chapters/chapter8/enduser/jwt-token-request-authn.yml
```

With the request authentication resource created let's verify the three different types of requests and their expected outcome.

REQUESTS WITH TOKENS FROM VALID ISSUERS ARE ACCEPTED

Let's make a request with valid JWT token, which is stored in the file `chapters/chapter8/enduser/user.jwt`:

```
$  USER_TOKEN=$(< chapters/chapter8/enduser/user.jwt); \
   curl -H "Host: apiserver.istioinaction.io" \
        -H "Authorization: Bearer $USER_TOKEN" \
        -sSl -o /dev/null -w "%{http_code}" $(make ingress-url)/api/catalog

200
```

Great! The successful response code shows that the authentication was successful, and as there were no authorization policies applied to the workload it was allowed by default.

REQUESTS WITH TOKENS FROM INVALID ISSUERS ARE REJECTED

For demonstration let's make a request with a token issued by **old-auth@istioinaction.io**, located in the file `chapters/chapter8/enduser/not-configured-issuer.jwt`:

```
$  WRONG_ISSUER=$(< chapters/chapter8/enduser/not-configured-issuer.jwt); \
   curl -H "Host: apiserver.istioinaction.io" \
        -H "Authorization: Bearer $WRONG_ISSUER" \
        -sSl $(make ingress-url)/api/catalog

Jwt issuer is not configured
```

Understandably, as there is no Request Authentication that validates tokens from this issuer it is

rejected.

REQUESTS WITHOUT TOKENS ARE ADMITTED INTO THE CLUSTER

For this case let's execute a curl request without a token:

```
$  curl -H "Host: apiserver.istioinaction.io" \
     -sSl -o /dev/null -w "%{http_code}" $(make ingress-url)/api/catalog

200
```

The successful response code shows that the request was admitted into the cluster. And this is kinda confusing, you'd expect that requests without tokens to be rejected. But in practice, there are many cases of requests that do not have tokens, such as serving the frontend of the application. For this reason, rejecting requests without tokens requires a little extra work.

DENYING REQUESTS WITHOUT JWT TOKENS

Let's create an authorization policy that denies requests targeting the API Gateway without a JWT Token:

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
 name: app-gw-requires-jwt
 namespace: istio-system
spec:
 selector:
   matchLabels:
     app: istio-ingressgateway
 action: DENY
 rules:
 - from:
   - source:
       notRequestPrincipals: ["*"]             ❶
   to:
   - operation:
       hosts: ["apiserver.istioinaction.io"]   ❷
```

❶ Must lack any request principal

❷ Matches only for this host

This policy makes use of the property notRequestPrincipals and the "*" value, which means that the source matches for all requests that lack the request principal property. The Request Principal property gets its value from two claims that are extracted by the Request Authentication filter from the token and stored in filter metadata. The two claims being issuer and subject in the format `iss/sub`.

Apply the resource to the cluster by executing:

```
$ kubectl apply -f chapters/chapter8/enduser/app-gw-requires-jwt.yaml
```

And verify by execution a token less request:

```
$ curl -H "Host: apiserver.istioinaction.io" \
     -sSl -o /dev/null -w "%{http_code}" $(make ingress-url)/api/catalog

403
```

Great! We prohibited requests without tokens, and as such we ensure that only authenticated end users have full access to the endpoints exposed by the API Gateway. This achieves denying unauthenticated requests, another regular requirement for real-world apps is to allow different levels of access for different users.

DIFFERENT LEVEL OF ACCESS BASED ON JWT CLAIMS

In this example, we'll allow regular users to read data from the API, but prohibit writing. Meanwhile, for administrators, we'll allow full access. For the requests the regular user token is found in the file `chapters/chapter8/enduser/user.jwt` and the admin user token is found in the file `chapters/chapter8/enduser/admin.jwt`. The tokens differ that the regular user has the claim **group: user** and the admin has the claim **group: admin**.

Let's setup an AuthorizationPolicy to allow regular users to read data when they are targeting the apiserver:

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
 name: allow-all-with-jwt-to-apiserver
 namespace: istio-system
spec:
 selector:
   matchLabels:
     app: istio-ingressgateway
 action: ALLOW
 rules:
 - from:
   - source:
       requestPrincipals: ["auth@istioinaction.io/*"] ❶
   to:
   - operation:
       hosts: ["apiserver.istioinaction.io"]
       methods: ["GET"]
```

❶ Request principal represents the end user request principal

And with the AuthorizationPolicy below we allow all operations to an admin user:

```
apiVersion: "security.istio.io/v1beta1"
kind: "AuthorizationPolicy"
metadata:
  name: "allow-mesh-all-ops-admin"
  namespace: istio-system
spec:
  rules:
    - from:
      - source:
          requestPrincipals: ["auth@istioinaction.io/*"]
```

```
      when:
      - key: request.auth.claims[group]
        values: ["admin"]                 ❶
```

❶ Allows only requests containing this claim

Apply these resources to the cluster by executing

```
$ kubectl apply -f \
     chapters/chapter8/enduser/allow-all-with-jwt-to-apiserver.yaml
$ kubectl apply -f chapters/chapter8/enduser/allow-mesh-all-ops-admin.yaml
```

Now proceed to verify that the regular user can read data:

```
$  USER_TOKEN=$(< chapters/chapter8/enduser/user.jwt);
   curl -H "Host: apiserver.istioinaction.io" \
     -H "Authorization: Bearer $USER_TOKEN" \
     -sSl -o /dev/null -w "%{http_code}" $(make ingress-url)/api/catalog

200
```

But writing is not allowed to a regular user:

```
$  USER_TOKEN=$(< chapters/chapter8/enduser/user.jwt);
   curl -H "Host: apiserver.istioinaction.io" \
     -H "Authorization: Bearer $USER_TOKEN" \
     -XPOST $(make ingress-url)/api/catalog \
     --data '{"id": 2, "name": "Shoes", "price": "84.00"}'

RBAC: access denied
```

Now let's verify that for the administrator writing is allowed:

```
$  ADMIN_TOKEN=$(< chapters/chapter8/enduser/admin.jwt);
   curl -H "Host: apiserver.istioinaction.io" \
     -H "Authorization: Bearer $ADMIN_TOKEN" \
     -XPOST -sSl -w "%{http_code}" $(make ingress-url)/api/catalog \
     --data '{"id": 2, "name": "Shoes", "price": "84.00"}'

200
```

Great! The successful response shows that the request with the claim group: admin was admitted into the cluster. This concludes end-user authentication and authorization. In the next section let's learn about Request Identity.

8.6 What is a request identity anyway?

Request Identity is represented by the validated data that is collected for a request by the PeerAuthentication and RequestAuthentication resources and made available to Authorization Policies in the form of filter metadata.

Some of the data made that is made available as filter metadata are:

- Principal - the workload identity defined by the PeerAuthentication

- Namespace - the workload namespace defined by the PeerAuthentication
- Request Principal - the end-user request principal defined by the RequestAuthentication
- Request Authentication Claims - the end-user claims that were extracted from the end-user token

To observe the collected metadata we can configure the service proxies to log those to standard output. Let's start with the metadata collected by the Request Authentication resource.

8.6.1 Checking out Request Authentication collected metadata

By default the envoy filter logger doesn't print the metadata in the logs, but we can increase the logging with the command below:

```
$ istioctl proxy-config log $(make ingress-pod) --level filter:debug
```

Generate logs by making a request with the admin user:

```
$  ADMIN_TOKEN=$(< chapters/chapter8/enduser/admin.jwt);
   curl -H "Host: apiserver.istioinaction.io" \
     -H "Authorization: Bearer $ADMIN_TOKEN" \
     -XPOST -sSl -w "%{http_code}" $(make ingress-url)/api/catalog \
     --data '{"id": 2, "name": "Shoes", "price": "84.00"}'

200
```

Now query the logs and find out the stored metadata:

```
$  kubectl logs $(make ingress-pod) -c istio-proxy

2020-07-31T11:11:40.579665Z    debug    envoy filter    [src/envoy/http/authn/authn_utils.cc:74]
    ProcessJwtPayload: json object is
        {"iat":1591545071,"exp":4745145071,"group":"admin","iss":"auth@istioinaction.io",
        "sub":"218d3fb9-4628-4d20-943c-124281c80e7b"} ❶
2020-07-31T11:11:40.589781Z    debug    envoy filter
    [src/envoy/http/authn/origin_authenticator.cc:95] JWT validation succeeded ❷
2020-07-31T11:11:40.589825Z    debug    envoy filter
    [src/envoy/http/authn/filter_context.cc:66]
    Set principal from origin:
        auth@istioinaction.io/218d3fb9-4628-4d20-943c-124281c80e7b #3 Request Principal is set
2020-07-31T11:11:40.589829Z    debug    envoy filter
    [src/envoy/http/authn/origin_authenticator.cc:106] Origin authenticator succeeded
2020-07-31T11:11:40.589940Z    debug    envoy filter
    [src/envoy/http/authn/http_filter.cc:88] Saved Dynamic Metadata: ❸
fields {
  key: "request.auth.claims"
  value {
    struct_value {
      fields {
        key: "group"
        value {
          list_value {
            values {
              string_value: "admin"
      # shortened
      }
      fields {
        key: "iss"
        value {
          list_value {
            values {
```

```
              string_value: "auth@istioinaction.io"
      # shortened
}
fields {
  key: "request.auth.principal"
  value {
    string_value: "auth@istioinaction.io/218d3fb9-4628-4d20-943c-124281c80e7b"
  }
}
```

❶ JWT data is decoded

❷ JWT is validated

❸ Claims saved as filter metadata.

On the output, we can see that the Request Authentication filter extracted the claims of end-user token and stored those as filter metadata. Those filter metadata are available for policies to take actions based upon them. Similarly Peer Authentication extracts and stores data in filter metadata.

8.6.2 Checking out Peer Authentication collected data

Peer Authentication is occurring between Ingress Gateway and the API Gateway, let's configure the API Gateway to log the metadata by executing the command below:

```
$ istioctl proxy-config log $(make apigateway-pod) --level filter:debug
```

Proceed to query the logs in the API Gateway workload:

```
$  kubectl logs $(make apigateway-pod) -c istio-proxy

2020-07-31T11:21:00.775345Z    debug    envoy filter
    [src/envoy/http/authn/filter_context.cc:35] Set peer from X509:
    cluster.local/ns/istio-system/sa/istio-ingressgateway-service-account
2020-07-31T11:21:00.775348Z    debug    envoy filter
    [src/envoy/http/authn/filter_context.cc:61] Set principal from peer:
    cluster.local/ns/istio-system/sa/istio-ingressgateway-service-account ❶
2020-07-31T11:21:00.775351Z    debug    envoy filter
    [src/envoy/http/authn/origin_authenticator.cc:106] Origin authenticator succeeded
2020-07-31T11:21:00.775698Z    debug    envoy filter
    [src/envoy/http/authn/http_filter.cc:88] Saved Dynamic Metadata:
fields {
  key: "request.auth.principal"
  value {
    string_value: "cluster.local/ns/istio-system/sa/istio-ingressgateway-service-account"
  }
}
fields {
  key: "source.namespace"
  value {
    string_value: "istio-system"
  }
}
fields {
  key: "source.principal"
  value {
    string_value: "cluster.local/ns/istio-system/sa/istio-ingressgateway-service-account"
  }
}
```

```
fields {
  key: "source.user"
  value {
    string_value: "cluster.local/ns/istio-system/sa/istio-ingressgateway-service-account"
  }
}
```

❶ Principal set from the SPIFFE ID, i.e. Workload service account.

Here we can see that the claims of the ingress gateway workload extracted from the Peer Authentication are made available in the filter metadata.

8.6.3 Overview of the flow of one request

Every request targeting a workload will go through the following filters:

- **JWT Authentication filter**: An Envoy filter that does JWT validation based on the JWT specification in authentication policies, and extracts the claims such as the authentication claims and custom claims, those are stored as filter metadata.
- **Peer Authentication filter**: An Envoy filter that enforces service authentication requirements and extracts authenticated attributes (peer identity such as source namespace and principal).
- **Authorization filter**: This is the authorization engine that checks the filter metadata collected by the previous filters and authorizes the request based on the policies applied to the workload.

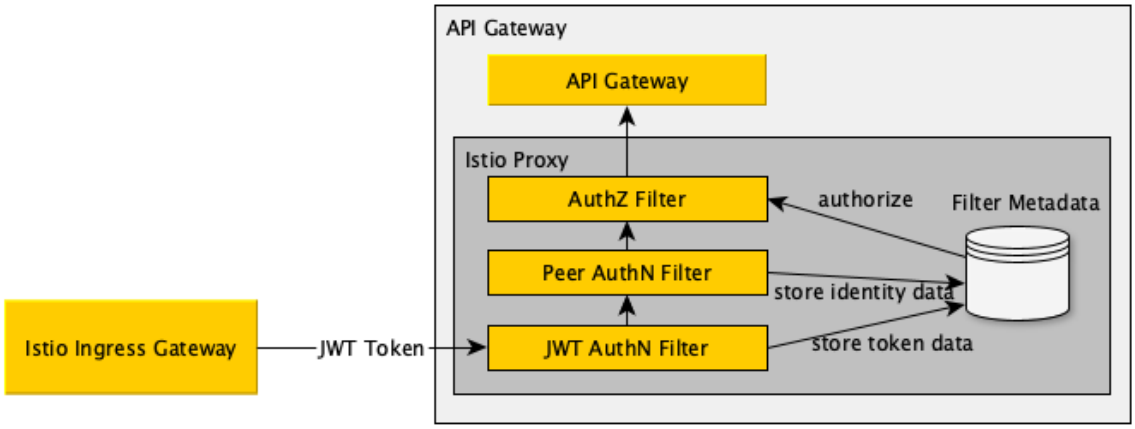


Figure 8.14 Collection of validated data in Filter Metadata

Now let's see the scenario where a regular user makes a POST request:

- The request passes the JWT Authentication filter, which as a result extracts the claims from the token and stores those in the filter metadata, this provides the request with an identity
- Peer to peer authentication is performed between the IngressGateway and the API Gateway. The peer-to-peer authentication filter extracts the Identity of the client and stores those in the filter metadata
- Authorization filters are executed in order:
  - **Deny authorization filters**: No deny authorization filter matches

- **Allow authorization filters**: The *allow-all-with-jwt-to-apiserver* authorization filter matches. Hence request is authorized.
- **Last (catch-all) authorization filter**: Filter would have been executed only if no prior filter has handled the request.

8.7 Summary

In this section, we looked at the security challenges that have to be dealt with in highly dynamic, distributed, and hybrid environments. We explored how Istio implements SPIFFE to issue identity to the Istio Workloads and how those Identities are used to enable Auto mTLS. We learned that issuing Identities to workloads was a precursor to being able to set up strict authorization policies.

Specifically, we configured the authentication and authorization policies uisng the following Istio resources:

- PeerAuthentication is used for defining peer to peer authentication and applying strict authentication requirements ensures that traffic is encrypted and cannot be eavesdropped. Vice versa we were introduced to the PERMISSIVE policy that allows an Istio workload to accept both encrypted traffic and clear-text traffic and how it can be used to slowly migrate without having downtime.
- AuthorizationPolicy is used for authorizing service-to-service requests and end-user requests based on the set of verifiable metadata extracted from either the workload identity or the end-user JWT Token
- RequestAuthentication is used for authenticating end-user requests containing JWT tokens.

Notes

1. signallake.com/innovation/soaNov05.pdf

2. A minor deviation from the SPIFFE specification, according to which the Workload Endpoint (i.e. Istio-Agent) should do the workload attestation