Red Hat
Training and Certification

Student Workbook (ROLE)

SM 1.1 DO328

Building Resilient Microservices with Istio and Red Hat OpenShift Service Mesh

Edition 2

Building Resilient Microservices with Istio and Red Hat OpenShift Service Mesh

SM 1.1 DO328
Building Resilient Microservices with Istio and Red Hat OpenShift Service Mesh
Edition 2 20200910
Publication date 20200910

Authors: Jordi Sola Alaball, Pablo Solar Vilariño, Marek Czernek, Ravi Srinivasan, Eduardo Ramirez Ronco, Jaime Ramirez Castillo
Editor: Nicole Muller

Contributors: David Sacco, Sajith Sugathan, Zachary Gutterman, Richard Allred, Sam Ffrench

Document Conventions

References

"References" describe where to find external documentation relevant to a subject.

Note

"Notes" are tips, shortcuts or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.

Important

"Important" boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring a box labeled "Important" will not cause data loss, but may cause irritation and frustration.

Warning

"Warnings" should not be ignored. Ignoring warnings will most likely cause data loss.

Introduction

DO328 Building Resilient Microservices with Istio and Red Hat OpenShift Service Mesh

The Red Hat OpenShift Service Mesh platform facilitates managing service interaction, provides service tracing, and creates a visual representation of communication pathways between microservices deployed on Red Hat OpenShift Container Platform.

Building Resilient Microservices with Istio and Red Hat OpenShift Service Mesh (DO328) is a hands-on, lab-based course that teaches students how to install, configure, and manage Red Hat® OpenShift® Service Mesh. In this course, students learn about service monitoring, service management, distributed tracing, load balancing, and service resilience.

Course Objectives

- Install, configure, and manage Red Hat OpenShift Service Mesh.
- This course, together with *Introduction to Containers, Kubernetes, and Red Hat OpenShift* (DO180) and *Red Hat OpenShift Developer I: Containerizing Applications* (DO288), prepares the student to take the *Red Hat Certified Specialist in Building Resilient Microservices* exam (EX328).

Audience

- System and Software Architects
- Software Developers

Prerequisites

- The courses *Introduction to Containers, Kubernetes, and Red Hat OpenShift* (DO180) and *Red Hat OpenShift Developer I: Containerizing Applications* (DO288), or demonstrate equivalent experience with containers and Kubernetes is strongly recommended, but not required.
- The course *Implementing Microservice Architectures (DO283)*, or demonstrate equivalent experience creating microservice applications is strongly recommended, but not required.

Orientation to the Classroom Environment

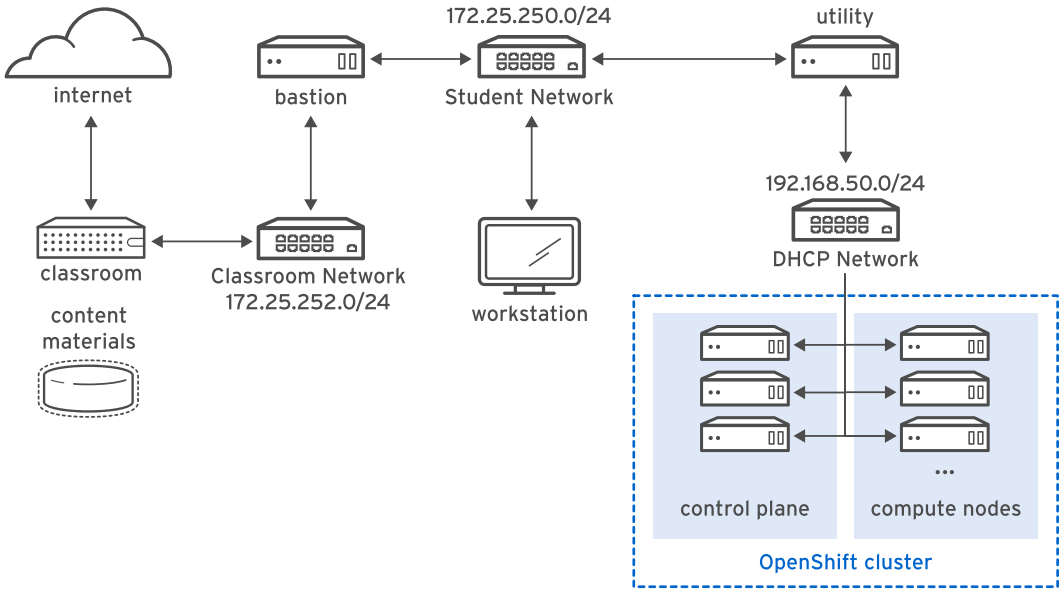


Figure 0.1: Classroom environment

In this course, the main computer system used for hands-on learning activities is **`workstation`**. The system called **`bastion`** must always be running. These two systems are in the **`lab.example.com`** DNS domain.

All student computer systems have a standard user account, **`student`**, which has the password **`student`**. The **`root`** password on all student systems is **`redhat`**.

Classroom Machines

Machine name	IP addresses	Role
workstation.lab.example.com	172.25.250.9	Graphical workstation used by students
bastion.lab.example.com	172.25.250.254	Router linking student's VMs to classroom servers
classroom.lab.example.com	172.25.252.254	Server hosting the classroom materials required by the course
utility.lab.example.com	172.25.250.253	Server providing supporting services required by the OCP cluster including DHCP and NFS and routing to the OCP servers.
master01.ocp4.example.com	192.168.50.10	OpenShift control plane.

Machine name	IP addresses	Role
master02.ocp4.example.com	192.168.50.11	OpenShift control plane.
master03.ocp4.example.com	192.168.50.12	OpenShift control plane.
worker01.ocp4.example.com	192.168.50.13	OpenShift compute node.
worker02.ocp4.example.com	192.168.50.14	OpenShift compute node.
worker03.ocp4.example.com	192.168.50.15	OpenShift compute node.

The **bastion** system acts as a router between the network that connects the student machines and the classroom network. If **bastion** is down, other student machines may not function properly or may even hang during boot.

The **utility** system acts as a router between the network that connects the OpenShift cluster machines and the student network. If **utility** is down, the OpenShift cluster will not function properly or may even hang during boot.

Several systems in the classroom provide supporting services. Two servers, **content.example.com** and **materials.example.com**, are sources for software and lab materials used in hands-on activities. Information on how to use these servers is provided in the instructions for those activities.

Students use the **workstation** machine to access a dedicated OpenShift cluster, for which they have cluster administrator privileges.

Controlling Your Systems

You are assigned a remote computer in a Red Hat Online Learning classroom, which is accessed through a web application hosted at **http://rol.redhat.com/**. Students should log in to this site using their Red Hat Customer Portal user credentials.

Controlling the Virtual Machines

The virtual machines in your classroom environment are controlled through a web page. The state of each virtual machine in the classroom is displayed on the page under the **Lab Environment** tab.

Machine States

Virtual Machine State	Description
active	The virtual machine is running and available (or, when booting, soon will be).
stopped	The virtual machine is completely shut down.
building	The initial creation of the virtual machine is being performed.

Depending on the state of a machine, a selection of the following actions is available.

Classroom/Machine Actions

Button or Action	Description
CREATE	Create the ROL classroom. Creates all of the virtual machines needed for the classroom and starts them. Can take several minutes to complete.
DELETE	Delete the ROL classroom. Destroys all virtual machines in the classroom. **Caution: Any work generated on the disks is lost.**
START	Start all virtual machines in the classroom.
STOP	Stop all virtual machines in the classroom.
OPEN CONSOLE	Open a new tab in the browser and connect to the console of the virtual machine. Students can log in directly to the virtual machine and run commands. In most cases, you should log in to the **`workstation`** virtual machine and use **`ssh`** to connect to the other virtual machines.
ACTION → Start	Start (power on) the virtual machine.
ACTION → Shutdown	Gracefully shut down the virtual machine, preserving the contents of its disk.
ACTION → Power Off	Forcefully shut down the virtual machine, preserving the contents of its disk. This is equivalent to removing the power from a physical machine.
ACTION → Reset	Forcefully shut down the virtual machine and reset the disk to its initial state. **Caution: Any work generated on the disk is lost.**

At the start of an exercise, if instructed to reset a single virtual machine node, click **ACTION → Reset** for only the specific virtual machine.

At the start of an exercise, if instructed to reset all virtual machines, click **ACTION → Reset**

If you want to return the classroom environment to its original state at the start of the course, you can click **DELETE** to remove the entire classroom environment. After the lab has been deleted, you can click **CREATE** to provision a new set of classroom systems.

Warning
The **DELETE** operation cannot be undone. Any work you have completed in the classroom environment up to that point will be lost.

The Autostop Timer

The Red Hat Online Learning enrollment entitles students to a certain amount of computer time. To help conserve allotted computer time, the ROL classroom has an associated countdown timer, which shuts down the classroom environment when the timer expires.

To adjust the timer, click **+** to add one hour to the timer. Note that there is a maximum time of twelve hours.

Internationalization

Per-user Language Selection

Your users might prefer to use a different language for their desktop environment than the system-wide default. They might also want to use a different keyboard layout or input method for their account.

Language Settings

In the GNOME desktop environment, the user might be prompted to set their preferred language and input method on first login. If not, then the easiest way for an individual user to adjust their preferred language and input method settings is to use the Region & Language application.

You can start this application in two ways. You can run the command **`gnome-control-center region`** from a terminal window, or on the top bar, from the system menu in the right corner, select the settings button (which has a crossed screwdriver and wrench for an icon) from the bottom left of the menu.

In the window that opens, select Region & Language. Click the **Language** box and select the preferred language from the list that appears. This also updates the **Formats** setting to the default for that language. The next time you log in, these changes will take full effect.

These settings affect the GNOME desktop environment and any applications such as **`gnome-terminal`** that are started inside it. However, by default they do not apply to that account if accessed through an **`ssh`** login from a remote system or a text-based login on a virtual console (such as **`tty5`**).

Note

You can make your shell environment use the same **`LANG`** setting as your graphical environment, even when you log in through a text-based virtual console or over **`ssh`**. One way to do this is to place code similar to the following in your **`~/.bashrc`** file. This example code will set the language used on a text login to match the one currently set for the user's GNOME desktop environment:

```
i=$(grep 'Language=' /var/lib/AccountsService/users/${USER} \
  | sed 's/Language=//')
if [ "$i" != "" ]; then
    export LANG=$i
fi
```

Japanese, Korean, Chinese, and other languages with a non-Latin character set might not display properly on text-based virtual consoles.

Individual commands can be made to use another language by setting the **`LANG`** variable on the command line:

```
[user@host ~]$ LANG=fr_FR.utf8 date
jeu. avril 25 17:55:01 CET 2019
```

Subsequent commands will revert to using the system's default language for output. The **locale** command can be used to determine the current value of **LANG** and other related environment variables.

Input Method Settings

GNOME 3 in Red Hat Enterprise Linux 7 or later automatically uses the IBus input method selection system, which makes it easy to change keyboard layouts and input methods quickly.

The Region & Language application can also be used to enable alternative input methods. In the Region & Language application window, the **Input Sources** box shows what input methods are currently available. By default, **English (US)** may be the only available method. Highlight **English (US)** and click the **keyboard** icon to see the current keyboard layout.

To add another input method, click the **+** button at the bottom left of the **Input Sources** window. An **Add an Input Source** window displays. Select your language, and then your preferred input method or keyboard layout.

When more than one input method is configured, the user can switch between them quickly by typing **Super**+**Space** (sometimes called **Windows**+**Space**). A *status indicator* will also appear in the GNOME top bar, which has two functions: It indicates which input method is active, and acts as a menu that can be used to switch between input methods or select advanced features of more complex input methods.

Some of the methods are marked with gear icons, which indicate that those methods have advanced configuration options and capabilities. For example, the Japanese **Japanese (Kana Kanji)** input method allows the user to pre-edit text in Latin and use **Down Arrow** and **Up Arrow** keys to select the correct characters to use.

US English speakers may also find this useful. For example, under **English (United States)** is the keyboard layout **English (international AltGr dead keys)**, which treats **AltGr** (or the right **Alt**) on a PC 104/105-key keyboard as a "secondary shift" modifier key and dead key activation key for typing additional characters. There are also Dvorak and other alternative layouts available.

Note

Any Unicode character can be entered in the GNOME desktop environment if you know the character's Unicode code point. Type **Ctrl**+**Shift**+**U**, followed by the code point. After **Ctrl**+**Shift**+**U** has been typed, an underlined **u** character displays, indicating that the system is waiting for Unicode code point entry.

For example, the lowercase Greek letter lambda has the code point U+03BB, and can be entered by typing **Ctrl**+**Shift**+**U**, then **03BB**, then **Enter**.

System-wide Default Language Settings

The system's default language is set to US English, using the UTF-8 encoding of Unicode as its character set (**en_US.utf8**), but this can be changed during or after installation.

From the command line, the **root** user can change the system-wide locale settings with the **localectl** command. If **localectl** is run with no arguments, it displays the current system-wide locale settings.

To set the system-wide default language, run the command **`localectl set-locale LANG=`*`locale`***, where *locale* is the appropriate value for the **`LANG`** environment variable from the "Language Codes Reference" table in this chapter. The change will take effect for users on their next login, and is stored in **`/etc/locale.conf`**.

```
[root@host ~]# localectl set-locale LANG=fr_FR.utf8
```

In GNOME, an administrative user can change this setting from Region & Language by clicking the **Login Screen** button at the upper-right corner of the window. Changing the **Language** of the graphical login screen will also adjust the system-wide default language setting stored in the **`/etc/locale.conf`** configuration file.

Important

Text-based virtual consoles such as **`tty4`** are more limited in the fonts they can display than terminals in a virtual console running a graphical environment, or pseudoterminals for **`ssh`** sessions. For example, Japanese, Korean, and Chinese characters may not display as expected on a text-based virtual console. For this reason, you should consider using English or another language with a Latin character set for the system-wide default.

Likewise, text-based virtual consoles are more limited in the input methods they support, and this is managed separately from the graphical desktop environment. The available global input settings can be configured through **`localectl`** for both text-based virtual consoles and the graphical environment. See the **`localectl`**(1) and **`vconsole.conf`**(5) man pages for more information.

Language Packs

Special RPM packages called *langpacks* install language packages that add support for specific languages. These langpacks use dependencies to automatically install additional RPM packages containing localizations, dictionaries, and translations for other software packages on your system.

To list the langpacks that are installed and that may be installed, use **`yum list langpacks-*`**:

```
[root@host ~]# yum list langpacks-*
Updating Subscription Management repositories.
Updating Subscription Management repositories.
Installed Packages
langpacks-en.noarch      1.0-12.el8        @AppStream
Available Packages
langpacks-af.noarch      1.0-12.el8        rhel-8-for-x86_64-appstream-rpms
langpacks-am.noarch      1.0-12.el8        rhel-8-for-x86_64-appstream-rpms
langpacks-ar.noarch      1.0-12.el8        rhel-8-for-x86_64-appstream-rpms
langpacks-as.noarch      1.0-12.el8        rhel-8-for-x86_64-appstream-rpms
langpacks-ast.noarch     1.0-12.el8        rhel-8-for-x86_64-appstream-rpms
...output omitted...
```

To add language support, install the appropriate langpacks package. For example, the following command adds support for French:

```
[root@host ~]# yum install langpacks-fr
```

Use **`yum repoquery --whatsupplements`** to determine what RPM packages may be installed by a langpack:

```
[root@host ~]# yum repoquery --whatsupplements langpacks-fr
Updating Subscription Management repositories.
Updating Subscription Management repositories.
Last metadata expiration check: 0:01:33 ago on Wed 06 Feb 2019 10:47:24 AM CST.
glibc-langpack-fr-0:2.28-18.el8.x86_64
gnome-getting-started-docs-fr-0:3.28.2-1.el8.noarch
hunspell-fr-0:6.2-1.el8.noarch
hyphen-fr-0:3.0-1.el8.noarch
libreoffice-langpack-fr-1:6.0.6.1-9.el8.x86_64
man-pages-fr-0:3.70-16.el8.noarch
mythes-fr-0:2.3-10.el8.noarch
```

Important

Langpacks packages use RPM *weak dependencies* in order to install supplementary packages only when the core package that needs it is also installed.

For example, when installing *langpacks-fr* as shown in the preceding examples, the *mythes-fr* package will only be installed if the *mythes* thesaurus is also installed on the system.

If *mythes* is subsequently installed on that system, the *mythes-fr* package will also automatically be installed due to the weak dependency from the already installed *langpacks-fr* package.

References

`locale`(7), **`localectl`**(1), **`locale.conf`**(5), **`vconsole.conf`**(5), **`unicode`**(7), and **`utf-8`**(7) man pages

Conversions between the names of the graphical desktop environment's X11 layouts and their names in **`localectl`** can be found in the file **`/usr/share/X11/xkb/rules/base.lst`**.

Language Codes Reference

Note

This table might not reflect all langpacks available on your system. Use **`yum info langpacks-`*`SUFFIX`*** to get more information about any particular langpacks package.

Language Codes

Language	Langpacks Suffix	$LANG value
English (US)	en	en_US.utf8

Language	Langpacks Suffix	$LANG value
Assamese	as	as_IN.utf8
Bengali	bn	bn_IN.utf8
Chinese (Simplified)	zh_CN	zh_CN.utf8
Chinese (Traditional)	zh_TW	zh_TW.utf8
French	fr	fr_FR.utf8
German	de	de_DE.utf8
Gujarati	gu	gu_IN.utf8
Hindi	hi	hi_IN.utf8
Italian	it	it_IT.utf8
Japanese	ja	ja_JP.utf8
Kannada	kn	kn_IN.utf8
Korean	ko	ko_KR.utf8
Malayalam	ml	ml_IN.utf8
Marathi	mr	mr_IN.utf8
Odia	or	or_IN.utf8
Portuguese (Brazilian)	pt_BR	pt_BR.utf8
Punjabi	pa	pa_IN.utf8
Russian	ru	ru_RU.utf8
Spanish	es	es_ES.utf8
Tamil	ta	ta_IN.utf8
Telugu	te	te_IN.utf8

Chapter 1

Introducing Red Hat OpenShift Service Mesh

Goal

Describe the basic concepts of microservice architecture and Red Hat OpenShift Service Mesh.

Objectives

- Describe the basic concepts behind a distributed architecture and Red Hat OpenShift Service Mesh.
- Describe the fundamental architecture of OpenShift Service Mesh components.

Sections

- Creating a Lab Environment (Guided Exercise)
- Describing OpenShift Service Mesh Concepts (and Quiz)
- Describing the OpenShift Service Mesh Architecture (and Quiz)
- Verifying OpenShift Credentials (Guided Exercise)

Guided Exercise

Creating a Lab Environment

In this exercise, you will start the provisioning process of a dedicated OpenShift cluster using the Red Hat Online Learning Environment (ROL). You will also create a new account for using the Quay.io public container image registry.

Outcomes

You should be able to provision a dedicated Red Hat OpenShift cluster that you will use for all exercises in this course.

You will also create a new account for using the Quay.io container image registry.

Before You Begin

To perform this exercise, ensure you have access to the Red Hat Online Learning Environment (ROL).

The following procedure describes how to provision an OpenShift cluster from the Red Hat Online Learning platform. You must complete the following procedure before attempting any of the course activities.

Note

The provisioning of the cluster takes approximately 10-15 minutes.

1. Open a browser and navigate to the Red Hat Online Learning (ROL) platform at `https://rol.redhat.com`. Log in with your credentials. If you do not have access to ROL, visit Red Hat Learning Subscription [https://www.redhat.com/en/services/training/learning-subscription].

2. Access the *Building Resilient Microservices with Istio and Red Hat OpenShift Service Mesh – DO328* course and click **ONLINE TRAINING**.

3. Click the **Lab Environment** tab, and then click **CREATE** to start provisioning the OpenShift cluster.

   Once the cluster is provisioned, you should see a list of virtual machines in the classroom. The status of all the virtual machines should be **`active`**.

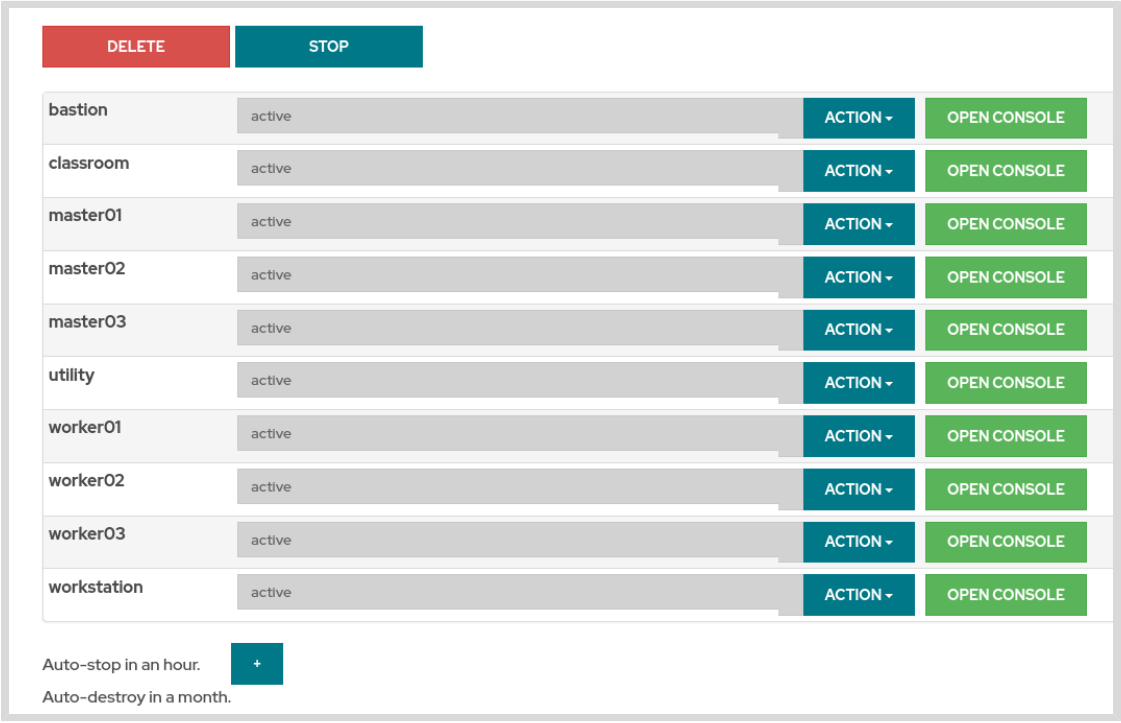


Figure 1.1: Provisioning a cluster

From this page, you can also control the environment, such as deleting the cluster, and stopping or restarting the cluster.

Warning

By default, The cluster is configured to automatically stop after one hour. Click the green plus icon at the bottom to increase the duration. You can increase the duration up to a maximum of 12 hours.

4. While the OpenShift cluster is being provisioned, create a new account in Quay.io. If you already have an account in Quay.io, then you can skip this step.

   If you do not have an account in Quay.io, create a new account to store container images for applications that you will build in this course.

   Refer to the detailed steps in the appendix *Creating a Quay Account*.

Finish

This exercise has no command to finish it. Provisioning the cluster takes some time. Continue to the next section in this chapter while you wait for the cluster to become ready.

This concludes the guided exercise.

Describing OpenShift Service Mesh Concepts

Objectives

After completing this section, you should be able to describe the basic concepts behind a distributed architecture and Red Hat OpenShift Service Mesh.

Describing the Challenges of Microservice Architectures

Microservice architectures are a method of dividing traditional, monolithic enterprise applications into a set of small, modular services. Using a microservice approach to application development means that each part of your application can scale more easily, is more maintainable, and is ideal for deployment on a cloud platform. This approach has been successfull in recent years, including at large companies such as Netflix and Amazon.

Despite introducing many benefits, microservices create several architectural challenges that administrators and developers must understand in order to build a robust and resilient microservices application.

Some of these challenges are related to the development of the microservices themselves.

Development challenges

An early issue developers run into is *service discovery*. Because services are often changing their IP address, each service needs to be easily discoverable and referred to by a static name. Another issue developers encounter is developing for *elasticity*, or the ability to scale up or down in response to demand. To support elasticity, and leverage one of the most critical benefits of a microservice architecture, developers need to design a system that is scalable as well as have an orchestration solution that appropriately responds to demand.

Security challenges

Security is critical nowadays, so microservices need to implement *authentication* techniques to validate and trust communication peers. Because microservice architectures imply a high degree of communication, authentication becomes a critical feature. Microservices must validate communication peers are *authorized*, and reject unauthorized requests.

Operation challenges

Microservices, like any other software, can fail. In microservice architectures, a failing microservice or element may cascade the error, causing a massive impact on the whole application. Microservices must be *resilient* to failures of peers or dependencies to avoid service failures and SLA breaks.

As applications grow larger and more complex, monitoring them becomes far more difficult. In contrast to a monolithic architecture, microservices are by nature distributed, which can make consolidating information more of a challenge. *Monitoring* (measuring microservices performance and usage), centralized *logging* (capturing and relating logs from all microservices) and *tracing* (correlating requests to multiple microservices belonging to the same user transaction) became desired features for any microservice architecture to be maintained.

Microservice orchestration platforms such as Red Hat OpenShift provide some of those capabilities, such as discovering services or elasticity. Some of the other features require more

specialized solutions. Recently, developers would implement these features such as service resilience in code, leading them to copy the same code from service to service and creating poor separation between service code and network management.

When applications consist of only a few microservices, replicating the same code is not a major problem. When the number of microservices increases, however, maintenance and the ability to make changes grows exponentially more difficult.

Describing a Service Mesh

Service mesh is a technology designed to address microservice architecture problems. This technology abstracts developers from many of the microservice architectural problems. Service mesh technology creates a centralized point to control features for many or all microservices in an application.

Service mesh technology operates at the network communication level. That is, service mesh components capture or intercept traffic to and from microservices, either modifying requests, redirecting them, or creating new requests to other services. Service mesh technology does all of this without requiring code-level changes.

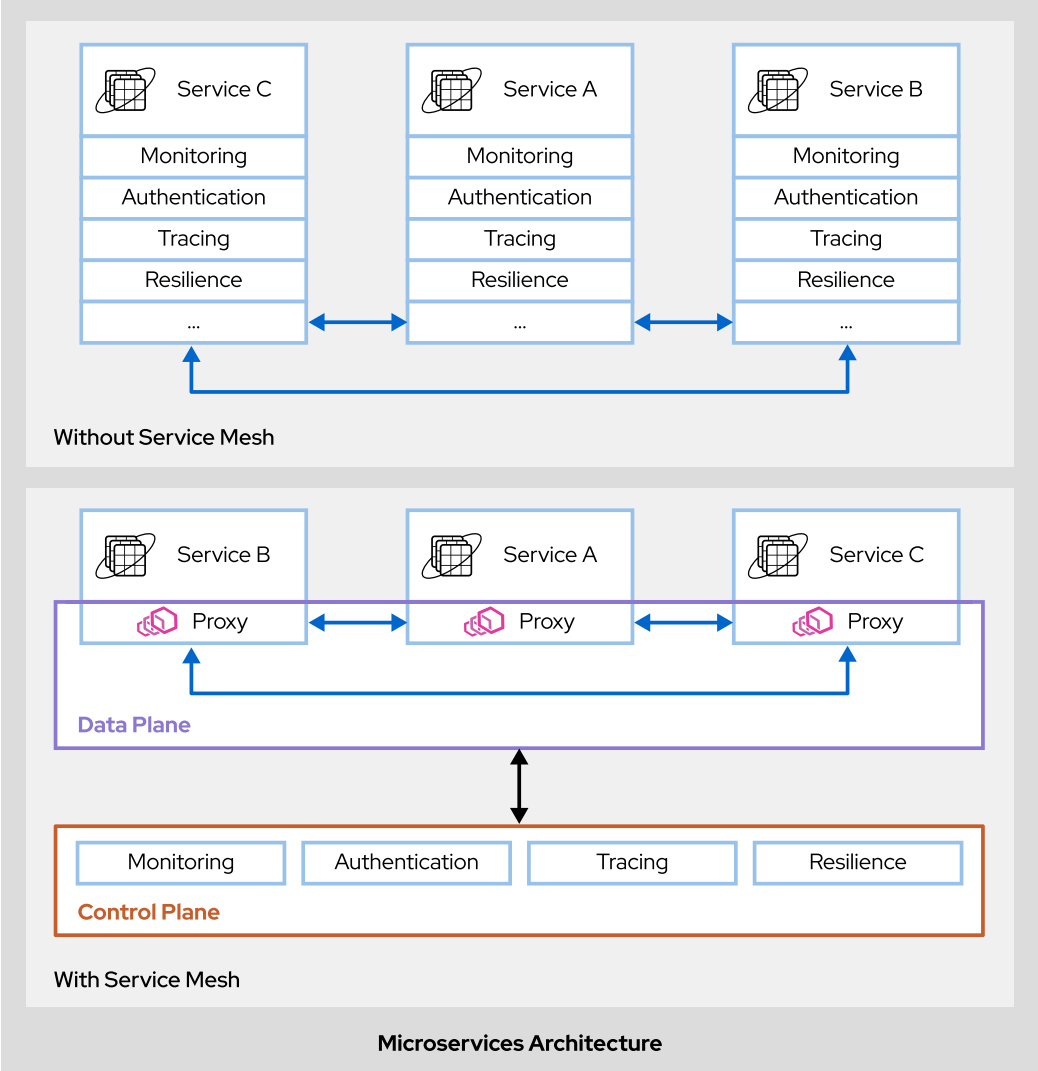


Figure 1.2: Desired features of microservice architectures

Red Hat OpenShift Service Mesh

Red Hat OpenShift Service Mesh implements the service mesh technology in Red Hat OpenShift Container Platform. This implementation complements OpenShift Container Platform capabilities, adding many of the desired features of microservice architectures.

OpenShift Service Mesh incorporates and extends several open source projects and orchestrates them to provide an improved developer experience:

Istio and Maistra

Istio is the core implementation of the service mesh architecture for the Kubernetes platform. Istio creates a control plane that centralizes service mesh capabilities and a data plane that makes up the structure of the mesh. The data plane controls communications between services by injecting sidecar containers that capture traffic between microservices.

Maistra is a project based on Istio that adapts features to the edge cases of deployment in OpenShift Container Platform. Maistra also adds extended features to Istio, such as simplified

multitenancy, explicit sidecar injection, and the use of OpenShift routes instead of Kubernetes ingress.

Jaeger and ElasticSearch

Jaeger is an open source traceability server that centralizes and displays traces associated with a single request between multiple services. Maistra is responsible for sending the traces to Jaeger and Jaeger is responsible for displaying traces. Microservices in the mesh are responsible for generating request headers needed for other components to generate and aggregate traces.

Jaeger relies on ElasticSearch for distributed storage and indexing for logging and tracing data. ElasticSearch is an open source, distributed, JSON-based search and analytics engine.

Kiali and Prometheus

Kiali provides service mesh observability. Kiali discovers microservices in the service mesh and their interactions and visually represents them. It also captures information about communication and services, such as the protocols used, service versions, and failure statistics.

Prometheus is used by OpenShift Service Mesh to store telemetry information from services. Kiali depends on Prometheus to obtain metrics, health status, and mesh topology.

Grafana

Optionally, Grafana can be used to analyze service mesh metrics. Grafana provides mesh administrators with advanced query and metrics analysis.

3scale

The 3scale Istio adapter is an optional component that integrates OpenShift Service Mesh with Red Hat 3scale API Management solutions. The default OpenShift Service Mesh installation does not include this component.

References

Istio
https://istio.io/

Maistra
https://maistra.io/

Jaeger
https://www.jaegertracing.io/

ElasticSearch
https://www.elastic.co/elasticsearch/

Kiali
https://kiali.io/

Prometheus
https://prometheus.io/

Grafana
https://grafana.com/

For updates and latest product notes, refer to the *Service Mesh Release notes* section in the *Service Mesh* chapter at
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.4/html-single/service_mesh/index

Quiz

Introducing OpenShift Service Mesh

Choose the correct answers to the following questions:

1. Which two of the following features does Red Hat OpenShift Service Mesh add to applications deployed on an OpenShift cluster? (Choose two.)

a. Scalability. Allows services to adapt the number of replicas to the load requirements.
b. Traceability. Allows services to capture traces of the requests between services.
c. Resiliency. Provides services with tools to tolerate failures in dependent services.
d. Idempotency. Deploying applications multiple times always obtains the same results.

2. Which three of the following open source projects are used in Red Hat OpenShift Service Mesh? (Choose three.)

a. Git
b. Kubernetes
c. Maistra
d. Kiali
e. PostgreSQL
f. Jaeger

3. Which of the following architectural challenges of microservice architectures is addressed by Red Hat OpenShift Service Mesh?

a. Authentication
b. API
c. Elasticity
d. Pipeline

4. An application consisting of multiple microservices is in production. It is complicated for architects to clarify which services communicate with others. Which two Red Hat OpenShift Service Mesh components can you use to clarify those connections? (Choose two.)

a. Kiali, because visualizing the service mesh clarifies service interactions.
b. Jaeger, because tracing queries shows what services are used.
c. Grafana, because it can display what nodes have a higher level of usage.
d. None of the above. OpenShift Service Mesh needs architects to manually trace all connections.

Solution

Introducing OpenShift Service Mesh

Choose the correct answers to the following questions:

1. Which two of the following features does Red Hat OpenShift Service Mesh add to applications deployed on an OpenShift cluster? (Choose two.)

a. Scalability. Allows services to adapt the number of replicas to the load requirements.
b. Traceability. Allows services to capture traces of the requests between services.
c. Resiliency. Provides services with tools to tolerate failures in dependent services.
d. Idempotency. Deploying applications multiple times always obtains the same results.

2. Which three of the following open source projects are used in Red Hat OpenShift Service Mesh? (Choose three.)

a. Git
b. Kubernetes
c. Maistra
d. Kiali
e. PostgreSQL
f. Jaeger

3. Which of the following architectural challenges of microservice architectures is addressed by Red Hat OpenShift Service Mesh?

a. Authentication
b. API
c. Elasticity
d. Pipeline

4. An application consisting of multiple microservices is in production. It is complicated for architects to clarify which services communicate with others. Which two Red Hat OpenShift Service Mesh components can you use to clarify those connections? (Choose two.)

a. Kiali, because visualizing the service mesh clarifies service interactions.
b. Jaeger, because tracing queries shows what services are used.
c. Grafana, because it can display what nodes have a higher level of usage.
d. None of the above. OpenShift Service Mesh needs architects to manually trace all connections.

Describing the OpenShift Service Mesh Architecture

Objectives

After completing this section, you should be able to describe the fundamental architecture of OpenShift Service Mesh components.

Red Hat OpenShift Service Mesh Architecture

Red Hat OpenShift Service Mesh consists of two logical components, a *control plane*, and a *data plane*. The following diagram shows the components in the data plane and the control plane:

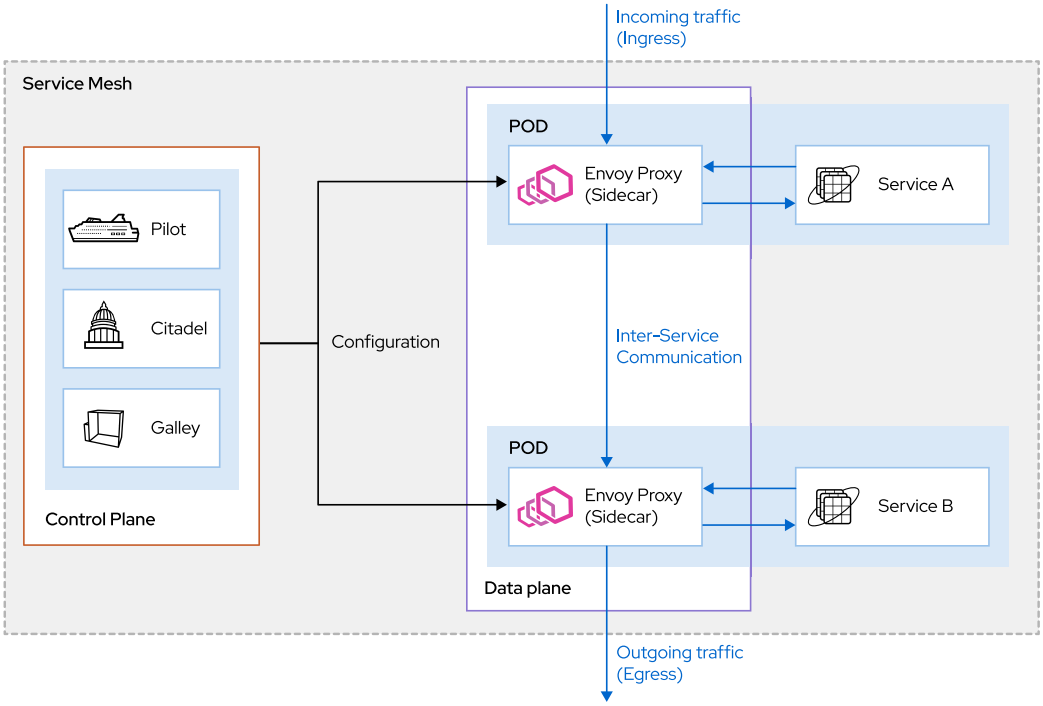


Figure 1.3: Red Hat OpenShift Service Mesh Architecture

The data plane consists of a set of *proxies*, which are deployed alongside applications in an OpenShift cluster. The proxies are deployed as *sidecars*, an auxiliary container running in the same pod as the application, and providing some supplementary functionality.

The control plane manages and configures the proxies. It enforces access control and usage policies and collects metrics from the proxies in the service mesh.

Data Plane Components

The *Envoy* proxy is the main component in the data plane. It handles all data flowing between the services in a service mesh. The Envoy proxy also collects all metrics related to the services in the mesh.

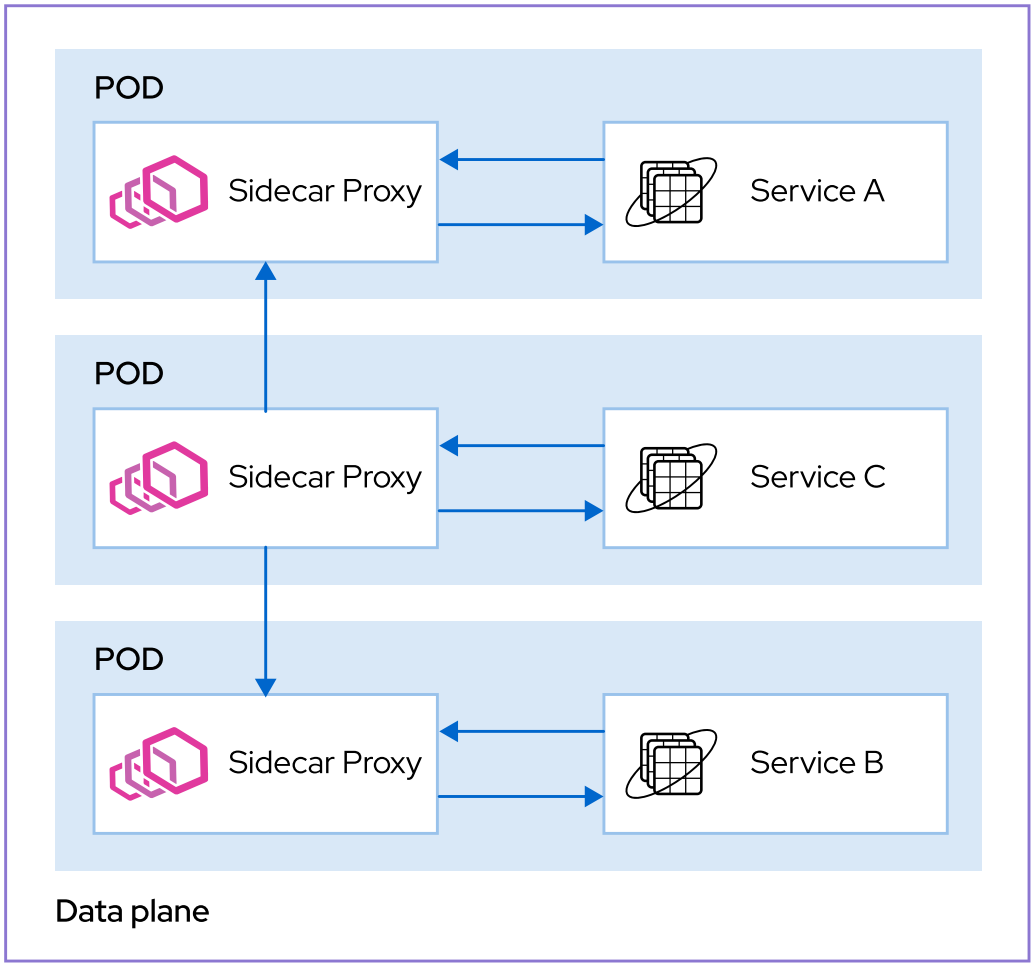


Figure 1.4: Envoy Proxies

The control plane automatically injects an instance of the Envoy proxy as a sidecar to a service whenever that service is deployed to the OpenShift cluster. All incoming (ingress) and outgoing (egress) network traffic between services flows through the proxies. The service offloads functionality such as access control, network routing and rate limiting, ingress and egress traffic control, and more to the service mesh.

The data plane in a service mesh performs the following tasks:

- **Service discovery:** Tracks the services deployed in a mesh.
- **Health checks:** Track the state (healthy or unhealthy) of the services deployed in a mesh.
- **Traffic shaping and routing:** Control the flow of network data between services. Includes tasks such as throttling the amount of traffic, routing based on content, circuit breaking, controlling the amount of traffic that should be routed among multiple versions of a service, load balancing and more.
- **Security:** Perform authentication and authorization, and secure communication using mutual transport layer security (mTLS) between services in a mesh.
- **Metrics and Telemetry:** Gather metrics, logs, and distributed tracing information from services in the mesh.

Control Plane Components

The control plane manages the configuration and policies for the service mesh. The control plane does not directly handle the network traffic in the mesh, but maintains configuration and policies that are enforced by the data plane.

The control plane consists of the following components:

Pilot
Maintains the configuration data for the service mesh. Pilot provides service discovery for the Envoy proxy sidecars, traffic management capabilities for intelligent routing (for example, A/B tests), and resiliency (timeouts, retries, and circuit breakers).

Citadel
Issues and rotates TLS certificates. Citadel provides authentication for inter-service communication, with built-in identity and credential management. You can enforce policies based on service identity rather than relying on network details such as IP addresses and host names.

Galley
Monitors the service mesh configuration and then validates, processes, and distributes the configuration to the proxies.

References
For more information, refer to the *Service Mesh Architecture* chapter in the *Service Mesh* guide at
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.4/html-single/service_mesh/index#service-mesh-architecture

Istio Architecture
https://archive.istio.io/v1.4/docs/ops/deployment/architecture

Quiz

Describing the OpenShift Service Mesh Architecture

Choose the correct answers to the following questions:

1. **Which component of the control plane is responsible for security and certificate management?**
   a. Pilot.
   b. Galley.
   c. Citadel.
   d. None of the above.

2. **Which two of the following statements about the control plane are correct? (Choose two.)**
   a. It injects an instance of the Envoy proxy as a sidecar to the application pod.
   b. It handles all the incoming and outgoing traffic in a service mesh.
   c. It runs as a sidecar alongside all applications in the service mesh.
   d. It is responsible for monitoring the health of all services in a service mesh.
   e. It is responsible for collecting logging data from all services in a service mesh.
   f. It is responsible for maintaining the configuration of the service mesh.

3. **A large e-commerce application is deployed on a service mesh, and consists of three services: an Apache web server to serve static assets such as images; a PHP/Nginx based front-end service that handles the HTML web user interface; and a Node.js back-end service to handle ordering, billing, and customer management.**
   All three services are packaged and deployed as containers; one container per service. Assuming that the service mesh handles the traffic for all three services, how many sidecar proxies are injected into the data plane for this application?
   a. 1
   b. 2
   c. 3
   d. 4
   e. 6

4. Which two of the following statements about the Pilot component in the control plane are correct? (Choose two.)

a. Pilot handles all the incoming and outgoing traffic in a service mesh.

b. Pilot runs as a sidecar alongside all applications in the service mesh.

c. Pilot provides service discovery functionality in the service mesh.

d. Pilot provides circuit breaker functionality in a service mesh.

e. Pilot enforces access control and usage policies in a service mesh.

f. Pilot validates the service mesh configuration and distributes the configuration to the proxies.

Solution

Describing the OpenShift Service Mesh Architecture

Choose the correct answers to the following questions:

1. **Which component of the control plane is responsible for security and certificate management?**
   a. Pilot.
   b. Galley.
   c. Citadel.
   d. None of the above.

2. **Which two of the following statements about the control plane are correct? (Choose two.)**
   a. It injects an instance of the Envoy proxy as a sidecar to the application pod.
   b. It handles all the incoming and outgoing traffic in a service mesh.
   c. It runs as a sidecar alongside all applications in the service mesh.
   d. It is responsible for monitoring the health of all services in a service mesh.
   e. It is responsible for collecting logging data from all services in a service mesh.
   f. It is responsible for maintaining the configuration of the service mesh.

3. **A large e-commerce application is deployed on a service mesh, and consists of three services: an Apache web server to serve static assets such as images; a PHP/Nginx based front-end service that handles the HTML web user interface; and a Node.js back-end service to handle ordering, billing, and customer management.**
   All three services are packaged and deployed as containers; one container per service. Assuming that the service mesh handles the traffic for all three services, how many sidecar proxies are injected into the data plane for this application?
   a. 1
   b. 2
   c. 3
   d. 4
   e. 6

4. Which two of the following statements about the Pilot component in the control plane are correct? (Choose two.)

a. Pilot handles all the incoming and outgoing traffic in a service mesh.

b. Pilot runs as a sidecar alongside all applications in the service mesh.

c. Pilot provides service discovery functionality in the service mesh.

d. Pilot provides circuit breaker functionality in a service mesh.

e. Pilot enforces access control and usage policies in a service mesh.

f. Pilot validates the service mesh configuration and distributes the configuration to the proxies.

Guided Exercise

Verifying OpenShift Credentials

In this exercise, you will configure the lab environment to access the dedicated OpenShift cluster that Red Hat Online Learning (ROL) provides for each student.

Outcomes

You should be able to access your dedicated OpenShift cluster from your lab environment.

Before You Begin

To perform this exercise, ensure you have access to:

- The Red Hat Online Learning Environment (ROL).
- The access credentials of your dedicated cluster.
- A free user account in the Quay.io public container registry.

Important

The following activity requires that you provision a dedicated OpenShift cluster in *Guided Exercise: Creating a Lab Environment*. Refer to this activity to create a new OpenShift cluster.

You also need to create new Quay.io account before starting this activity. Refer to the appendix *Creating a Quay Account*.

The **`lab-configure`** command saves the connection information of your OpenShift cluster in a configuration file. If you made a mistake, or if you want to change any values, rerun the command to modify the configuration.

Note

To avoid problems while pasting text into the workstation machine, open the Firefox browser on workstation and navigate to https://rol.redhat.com. Copy the necessary values from Firefox and paste them into the GNOME Terminal window.

1. Run the **`lab-configure`** command to configure your workstation environment to connect to your OpenShift cluster, which you provisioned from ROL in an earlier exercise.

The **`lab-configure`** command provides interactive prompts. It provides a set of defaults for all the prompts except the username for your Quay.io account. Accept these default values and enter your Quay.io username when prompted.

1.1. Run the **`lab-configure`** command. The OpenShift API URL is automatically filled in. Accept the default value by pressing the **`Enter`** key.

```
[student@workstation ~]$ lab-configure

This script configures the connection parameters to access the OpenShift cluster
 for your lab scripts

· Enter the OpenShift API URL: https://api.ocp4.example.com:6443
```

1.2. The script attempts to determine the correct wildcard domain for your cluster, as displayed in the following output. If the domain is incorrect, make the necessary changes.

```
· Enter the Wildcard Domain: apps.ocp4.example.com
```

1.3. The script attempts to determine the correct web console URL for your cluster, as displayed in the following output. If the URL is incorrect, make the necessary changes. Ensure you include **`https://`** if it is not part of the web console URL provided to you.

```
· Enter the Web Console URL: https://console-openshift-
console.apps.ocp4.example.com:6443
```

1.4. You will need a unprivileged developer user account for running the labs in the course. Accept the default user account named **`developer`**.

```
· Enter the Developer User name: developer
```

1.5. Accept the default password for the developer user account (**`developer`**).

```
· Enter the Developer User Password: developer
```

1.6. You will also need a user with cluster administrator permissions to install the service mesh. Accept the default administrator account named **`admin`**.

```
· Enter the Cluster Administrator User name: admin
```

1.7. Accept the default password for the administrator user account (**`redhat`**).

```
· Enter the Cluster Administrator User Password: redhat
```

1.8. Do not press **`Enter`** to accept the default value. Enter your Quay.io user account name:

```
· Enter the Quay.io Account User Name: quayuser
```

2. The **`lab-configure`** command displays all the values that you provided and does not wait for you to confirm them. It then verifies that it can access your cluster.

2.1. Verify that you provided the correct values for your OpenShift cluster to the **`lab-configure`** command.

```
You entered:
 · OpenShift API URL:                  https://api.ocp4.example.com:6443
 · Wildcard Domain:                    apps.ocp4.example.com
 . Web Console URL:                    https://console-openshift-
console.apps.ocp4.example.com:6443
 . Developer User Name:                developer
 . Developer User Password:            developer
 . Cluster Administrator User Name:    admin
 · Cluster Administrator User Password: redhat
 . Quay.io Account User Name:          quayuser
```

2.2. Wait until the **`lab-configure`** command verifies that it can connect to your OpenShift cluster and saves your configuration to the **`/usr/local/etc/ocp4.config`** file.

```
Verifying your OpenShift API URL...

Verifying your OpenShift developer user credentials...

Verifying your OpenShift admin user credentials...

Verifying your Quay.io account user name...

Verifying your cluster configuration...

Saving your lab configuration file...

All fine, lab config saved. You can now proceed with your exercises.
```

2.3. If the **`lab-configure`** command finds any issues then it displays an error message and exits. You must verify the information you entered and run the **`lab-configure`** command again. The following listing shows an example of a verification error:

```
...output omitted...

Verifying your OpenShift API URL...

ERROR:
Cannot connect to an OpenShift 4 API using your URL.
Please verify your network connectivity and that the URL does not point to an
 OpenShift 3.x nor to a non-OpenShift Kubernetes API.
No changes made to your lab configuration.
```

Note

You may see this error when you run the **`lab-configure`** script immediately after cluster provisioning. Wait for a few minutes for the cluster to be fully started and the OpenShift API to be available, and re-run the **`lab-configure`** script.

If your configuration saved without errors, then you are ready to start any of the exercises for this course. If there were any errors, then do not start any exercise until you can execute the **`lab-configure`** command successfully.

Finish

This exercise has no command to finish it. You should now be set up to perform any exercise in this course.

This concludes the guided exercise.

Summary

In this section, you learned:

- OpenShift Service Mesh addresses challenges in microservice architectures, like tracing, authentication, elasticity, traceability or resiliency. A service mesh works at the network level, that is, it intercepts and alters network communication between services to include the desired features.
- Red Hat OpenShift Service Mesh uses many open source projects, such as Istio, Kiali, Jaeger, ElasticSearch, and Grafana.
- Istio (the core of OpenShift Service Mesh technology) defines the control plane, for controlling service mesh behavior, and the data plane, for enabling features at the service level.
- The data plane injects Envoy proxy sidecars to microservices to enable service mesh features. Control plane components includes Pilot for service discovery, Citadel for authentication, and Galley for configuration.

Chapter 2

Installing Red Hat OpenShift Service Mesh

Goal	Deploy Red Hat OpenShift Service Mesh on OpenShift Container Platform.
Objectives	Install Red Hat OpenShift Service Mesh on Red Hat OpenShift Container Platform.
Sections	Installing Red Hat OpenShift Service Mesh (and Guided Exercise)

Installing Red Hat OpenShift Service Mesh

Objectives

After completing this section, you should be able to install Red Hat OpenShift Service Mesh on Red Hat OpenShift Container Platform.

Describing Custom Resource Definitions

Red Hat OpenShift is a distribution of Kubernetes focused on the developer experience. Red Hat OpenShift is easily adapted to various projects using available Kubernetes components.

A *resource* is a Kubernetes API endpoint that stores objects of the same kind.

A *Custom Resource Definition* (CRD) describes a custom resource used to extend the Kubernetes API. This feature supports custom object definitions, using them as native Kubernetes objects.

Custom Resource Definitions are used to install software in Kubernetes and Red Hat OpenShift.

The following example shows a CRD definition:

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: crontabs.stable.example.com (1)
spec:
  group: stable.example.com (2)
  version: v1
  scope: Namespaced (3)
  names:
    plural: crontabs (4)
    singular: crontab (5)
    kind: CronTab (6)
    shortNames: (7)
    - ct
```

A crontab is a list of commands to be executed at a specified time. The preceding example implements the crontab concept using Custom Resource Definitions.

1. Custom Resource Definition name. Must be in the form **`plural.group`**.
2. Name to use in the REST API.
3. Defines the scope of the CRD.
4. Plural name of the CRD. Used in the REST API and to form the CRD name.
5. Singular name of the CRD. Used as an alias in the command-line interface (CLI) and for display.
6. Type of objects managed by the CRD.
7. Short string alias to use in the CLI.

After a CRD is created, Kubernetes enables a new RESTful API endpoint to manage it. The endpoint created for the preceded example is:

```
/apis/stable.example.com/v1/namespaces/*/crontabs/...
```

The API URL associated with the CRD supports creating and managing custom objects.

Defining Kubernetes Operators

A Kubernetes operator packages a Kubernetes application to automate installation, updates, and management. Operators rely on Custom Resource Definitions to extend the Kubernetes API.

Operators run on a pod, and monitor the application to ensure it performs as expected. If the application fails to properly execute, then the operator automatically acts to correct it.

There are two types of operators available to choose in Red Hat OpenShift.

Certified Operators
Operators verified on Red Hat OpenShift by Red Hat or its partners.

Community Operators
Operators not vetted or verified by Red Hat, so their stability is unknown.

Installing Red Hat OpenShift Service Mesh

OpenShift Service Mesh is installed using the Web Console, or CLI, and a Kubernetes operator. The installation process requires first installing the required operators, then deploying the Control Plane, and finally creating a Service Mesh Member Roll.

Installing the OpenShift Service Mesh Operator

OpenShift Service Mesh relies on the following operators:

Jaeger
Provides tracing features to monitor and troubleshoot your distributed application.

Elasticsearch
Stores traces and logs generated by Jaeger.

Kiali
Provides observability to the service mesh through a web user interface (UI).

You can find all the required operators and the Red Hat OpenShift Service Mesh operator in the **`OperatorHub`** page.

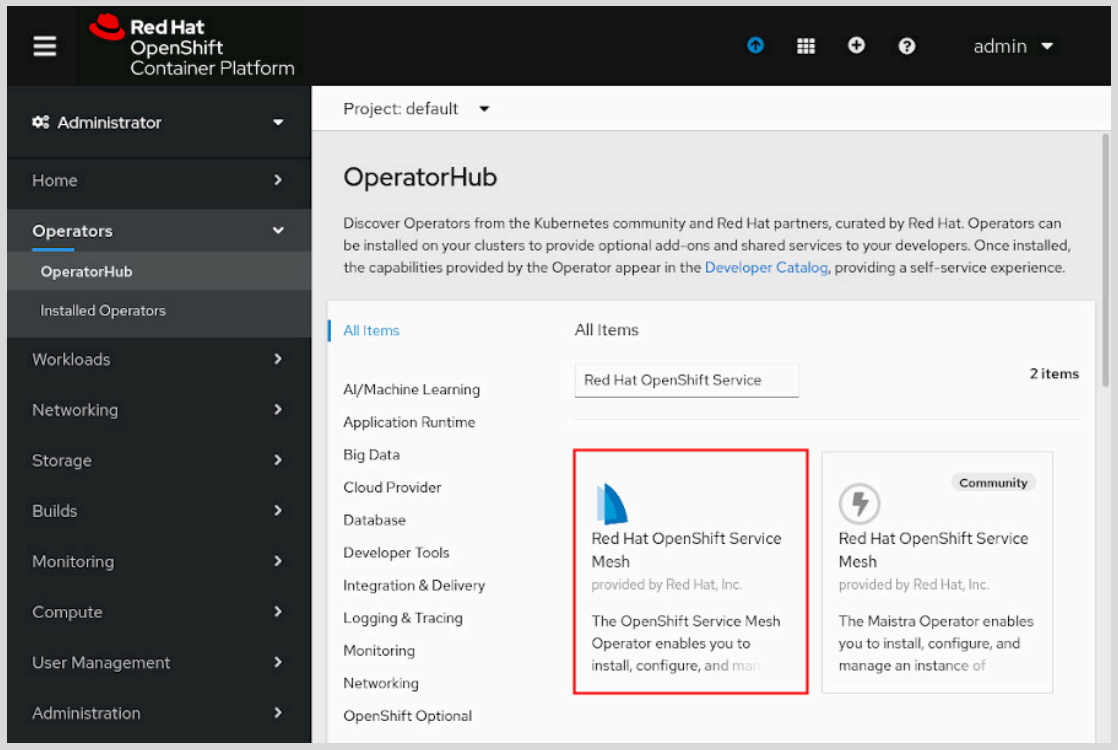


Figure 2.1: Result page in the OperatorHub for Red Hat OpenShift Service Mesh

The installation process of the operators requires first finding the operator in the OperatorHub page, then reviewing and configuring installation parameters, and finally subscribing the operator to an updates channel.

Note
Red Hat recommends to install certified operators.

Deploying the OpenShift Service Mesh Control Plane

The control plane manages the configuration and policies for the service mesh. The OpenShift Service Mesh Operator installation makes the operator available in all namespaces, so you can install the control plane in any project.

To deploy a control plane in a project with the web UI, first navigate to the **`Installed Operators`** page, then to the **`Istio Service Mesh Control Plane`** page, and finally review and configure deployment parameters.

Note
Red Hat recommends to deploy the control plane in a separate project.

Creating a Service Mesh Member Roll

The **`ServiceMeshMemberRoll`** custom resource defines the projects belonging to a control plane. Any number of projects can be added to a **`ServiceMeshMemberRoll`**, however a project can be added only to one control plane.

To create or edit a Service Mesh Member Roll, first navigate to the project where Red Hat OpenShift Service Mesh is installed, then navigate to the **`Istio Service Mesh Member Roll`** page, and finally review and configure installation parameters.

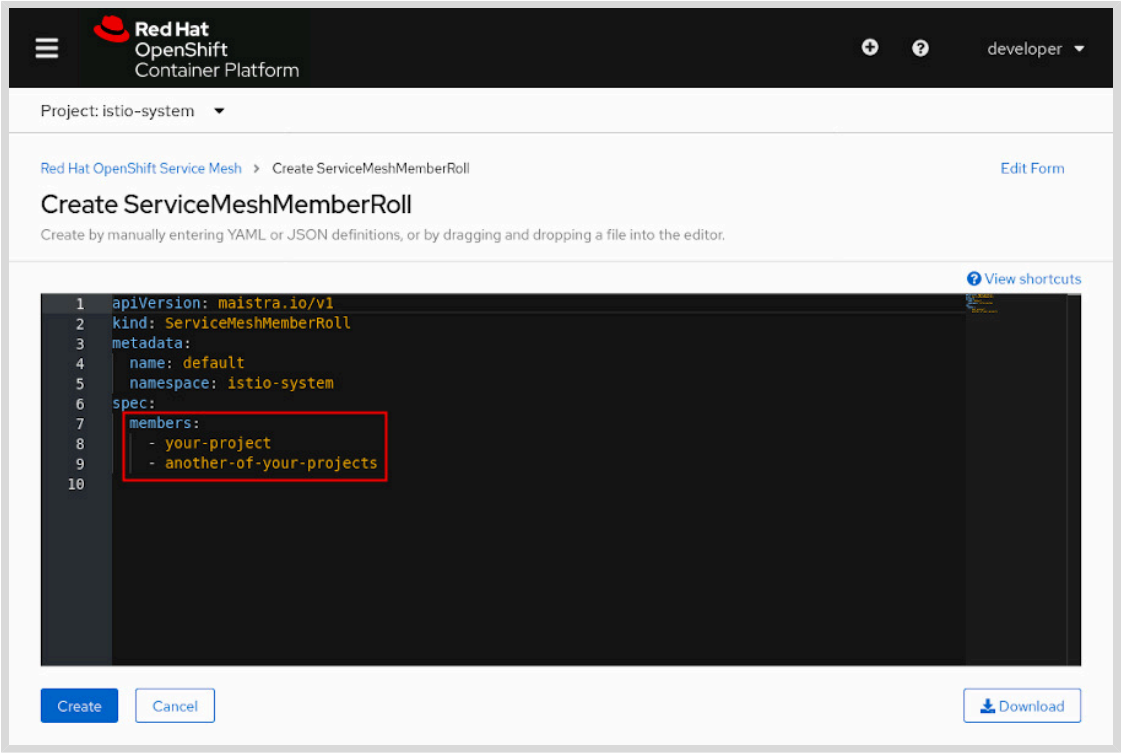


Figure 2.2: List of projects belonging to the control plane

Only projects listed in the **`ServiceMeshMemberRoll`** are managed by the Service Mesh.

Installing OpenShift Service Mesh using CLI

You can install OpenShift Service Mesh using CLI instead of the OpenShift Web Console. The CLI installation method is useful for, for example, automated installation and management of OpenShift Service Mesh.

For more information about installing OpenShift Service Mesh using CLI, refer to *Installing Red Hat OpenShift Service Mesh with the CLI*.

Upgrading Red Hat OpenShift Service Mesh

If you selected the automatic update stream during installation, then OpenShift Service Mesh is going to be updated automatically, so any extra steps are not required.

If you choose to update manually during installation, the Operator Lifecycle Manager (OLM) controls the upgrade.

For more information, refer to the *Operator Lifecycle Manager* documentation at https://access.redhat.com/documentation/en-us/openshift_container_platform/4.4/html-single/operators/#olm-overview_olm-understanding-olm.

Removing Red Hat OpenShift Service Mesh

To remove OpenShift Service Mesh from an existing Red Hat OpenShift Container Platform instance, complete the following tasks:

1. Remove the OpenShift Service Mesh control plane.
2. Remove the installed operators.
   a. Red Hat OpenShift Service Mesh Operator
   b. Jaeger Operator
   c. Kiali Operator
   d. Elasticsearch Operator
3. Clean up operator resources.

For more information, refer to the *Removing Red Hat OpenShift Service Mesh* section in the *OpenShift Container Platform Service Mesh* documentation at https://access.redhat.com/documentation/en-us/openshift_container_platform/4.4/html-single/service_mesh/index.

References

For more information, refer to the *Service Mesh Installation* section in the *OpenShift Container Platform Service Mesh* documentation at
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.4/html-single/service_mesh/index

Operators in Red Hat OpenShift
https://www.openshift.com/learn/topics/operators

Operator Hub
https://operatorhub.io/

Guided Exercise

Install OpenShift Service Mesh

In this exercise, you will deploy Red Hat OpenShift Service Mesh on Red Hat OpenShift.

Outcomes

You should be able to deploy OpenShift Service Mesh on Red Hat OpenShift.

Before You Begin

To perform this exercise, ensure you have access to:

- A configured and running OpenShift cluster.

As the **`student`** user on the **`workstation`** machine, use the **`lab`** command to validate the prerequisites for this exercise:

```
[student@workstation ~]$ lab install-mesh start
```

1. Log in as the **`admin`** user in the OpenShift web console.

   1.1. Navigate to **`https://console-openshift-console.apps.ocp4.example.com`** to access the OpenShift web console. If certificate errors appear, then accept the self-signed certificates.

```
[student@workstation ~]$ firefox https://console-openshift-
console.apps.ocp4.example.com
```

   1.2. Use the following credentials:

   - **Username**: **`admin`**
   - **Password**: **`redhat`**

   Then, click **Log in**.

2. Install the **`Elasticsearch`** operator.

   2.1. In the Administrator panel, click **Operators** → **OperatorHub**.

   2.2. Type **`Elasticsearch`** into the filter box. Click **Elasticsearch Operator**, and then click **Install**.

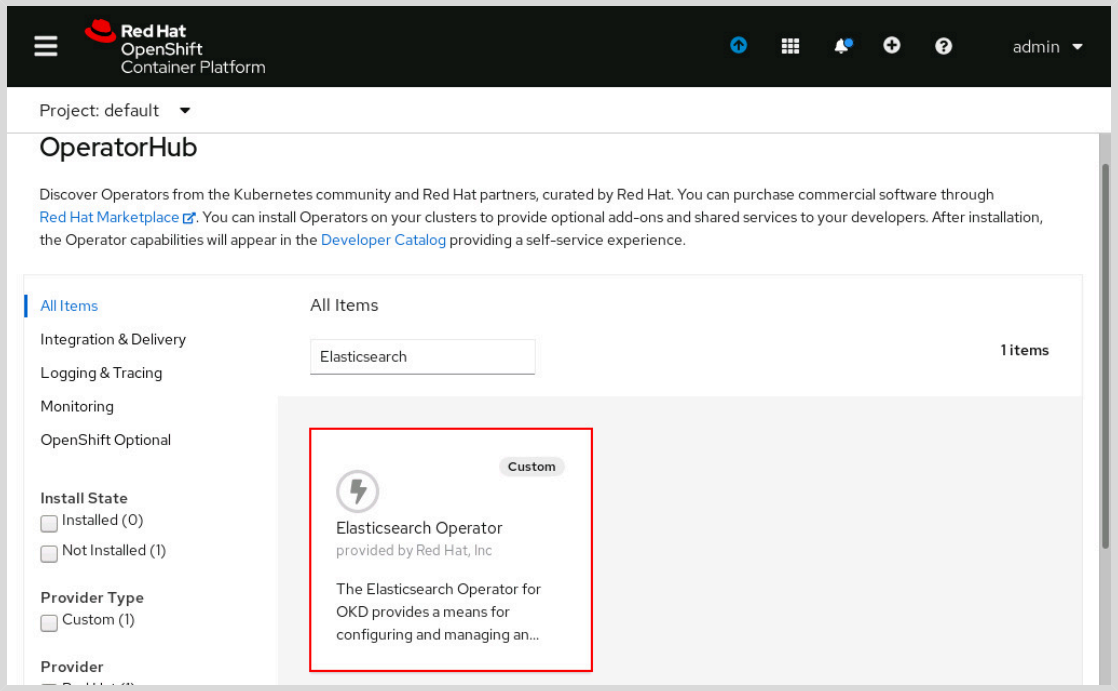


Figure 2.3: Elasticsearch operator

Note
All the required operators for this course are packaged in a custom OperatorHub. For that reason, all the operators included in the custom OperatorHub display a **`Custom`** tag.

2.3. In the **Create Operator Subscription** page, examine the default settings. Then, click **Subscribe**.

3. Install the **`Jaeger`** operator.

3.1. In the Administrator panel, click **Operators** → **OperatorHub**.

3.2. Type **`Jaeger`** into the filter box. Click **Red Hat OpenShift Jaeger**, and then click **Install**.

3.3. The **Create Operator Subscription** page displays. Examine the default settings, and select **1.17-stable** as the **Update Channel**. Then, click **Subscribe**.

4. Install the **`Kiali`** operator.

4.1. In the Administrator pane, click **Operators** → **OperatorHub**.

4.2. Type **`Kiali`** into the filter box. Click **Kiali Operator**, and then click **Install**.

4.3. The **Create Operator Subscription** page displays. Examine the default settings, and then click **Subscribe**.

5. Install the **`Red Hat OpenShift Service Mesh`** operator.

5.1. In the Administrator pane, click **Operators** → **OperatorHub**.

5.2. Type **`OpenShift Service Mesh`** into the filter box. Click **Red Hat OpenShift Service Mesh**, and then click **Install**.

5.3. In the **Create Operator Subscription** page, examine the default settings and select **1.0** as the **Update Channel**. Then, click **Subscribe**.

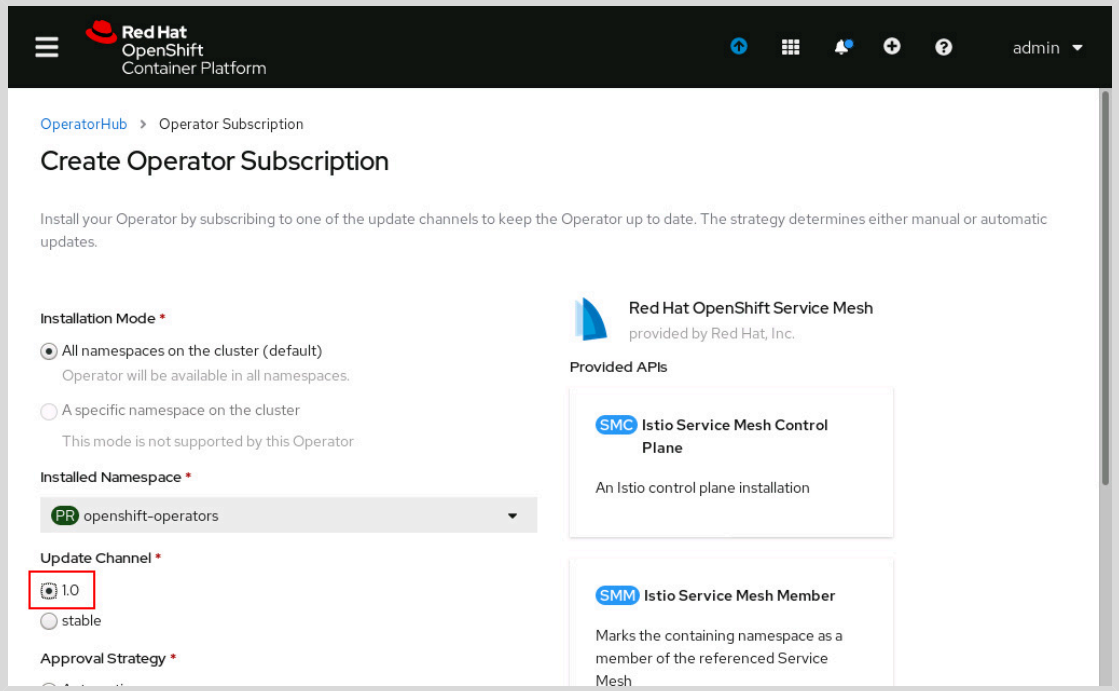


Figure 2.4: OpenShift Service Mesh subscription options

5.4. In the **Installed Operators** page, wait until you see **Succeeded** in the **Status** column of all the operators.

6. Log in as the **`Developer`** user. Then, create the **`istio-system`** project.

6.1. In the OpenShift web console, click **admin** → **Log out**. Then, log in as the **`developer`** user.

6.2. In the OpenShift web console, click **Home** → **Projects**, and then click **Create Project**.

6.3. Type **`istio-system`** into the **Name** field, and then click **Create**.

7. Deploy the OpenShift Service Mesh control plane.

7.1. On the **Administrator** pane, click **Operators** → **Installed Operators**.

7.2. Ensure that the project **`istio-system`** is selected in the **Project** list. Then, click **Red Hat OpenShift Service Mesh**.

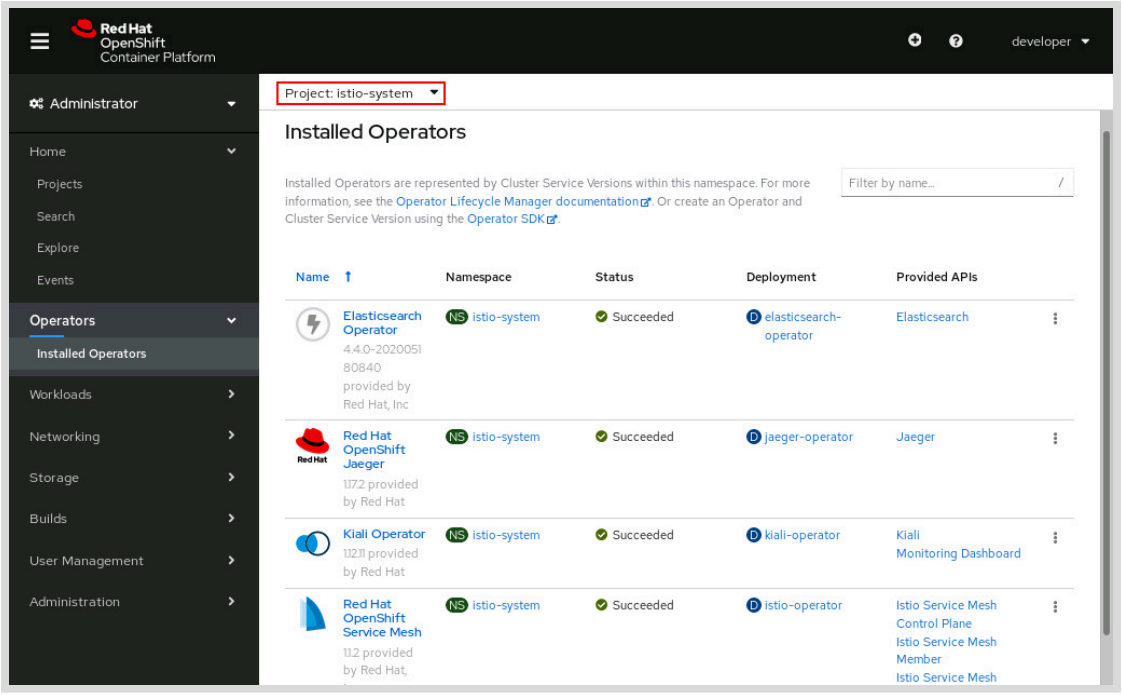


Figure 2.5: Project selection in OperatorHub

> **Note**
> If **Red Hat OpenShift Service Mesh Operator** does not display, the operator installation into the **`istio-system`** project is still in progress.
>
> The operator displays after a few seconds.

7.3. Click the **Istio Service Mesh Control Plane** tab. Then, click **Create ServiceMeshControlPlane**.

7.4. Examine the default options of the Service Mesh Control Plane installation. Then, click **Create**.

7.5. Click the **Istio Service Mesh Member Roll** tab. Then, click **Create ServiceMeshMemberRoll**.

7.6. Examine the default options of the ServiceMeshMemberRoll custom resource and remove the sample members. Then, click **Create**.

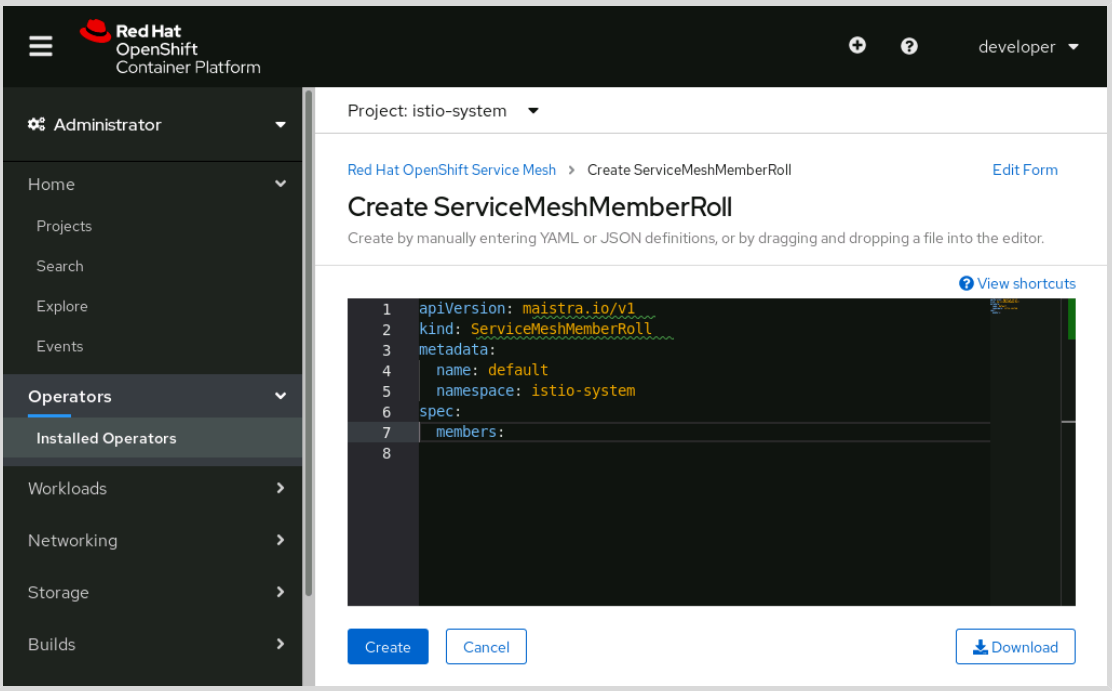


Figure 2.6: Project selection in OperatorHub

8. Verify that the Service Mesh Control Plane installation was successful.

   8.1. Click the **Istio Service Mesh Control Plane** tab.

   8.2. Click **basic-install** to see the details of your Service Mesh Control Plane installation.

   8.3. Scroll down to see the **Conditions** panel of the page.

   8.4. Wait until the **Ready** row status changes to **true**.

   8.5. As the **student** user on the **workstation** machine, use the **lab** command to verify the successful installation:

```
[student@workstation ~]$ lab install-mesh grade
```

Note

To remove OpenShift Service Mesh from your OpenShift cluster, execute **lab uninstall-mesh start**.

Finish

On the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab install-mesh finish
```

This concludes the guided exercise.

Summary

In this chapter, you learned:

- The role of the **`ServiceMeshMemberRoll`** and **`ServiceMeshControlPlane`** resources in Red Hat OpenShift Service Mesh.
- Installing Red Hat OpenShift Service Mesh using the OpenShift web console.
- Installing Red Hat OpenShift Service Mesh using the OpenShift command-line client.
- Configure Red Hat OpenShift Service Mesh to manage projects with the **`ServiceMeshMemberRoll`** resource.

Chapter 3

Observing a Service Mesh

Goal

Trace and visualize an OpenShift Service Mesh with Jaeger and Kiali.

Objectives

- Configure distributed tracing to track service traffic.
- Collect and inspect critical metrics with Prometheus and Grafana.
- Monitor and visualize service interactions with Kiali.

Sections

- Tracing Services with Jaeger (and Guided Exercise).
- Collecting Service Metrics (and Guided Exercise).
- Observing Service Interactions with Kiali (and Guided Exercise).

Lab

Observing an OpenShift Service Mesh.

Tracing Services with Jaeger

Objectives

After completing this section, you should be able to configure distributed tracing to track service traffic.

Describing Distributed Tracing

Distributed Tracing is the process of tracking the performance of individual services in an application by tracing the path of the service calls in the application. Each time a user takes action in an application, a request is executed that might require many services to interact to produce a response. The path of this request is called a *distributed transaction*.

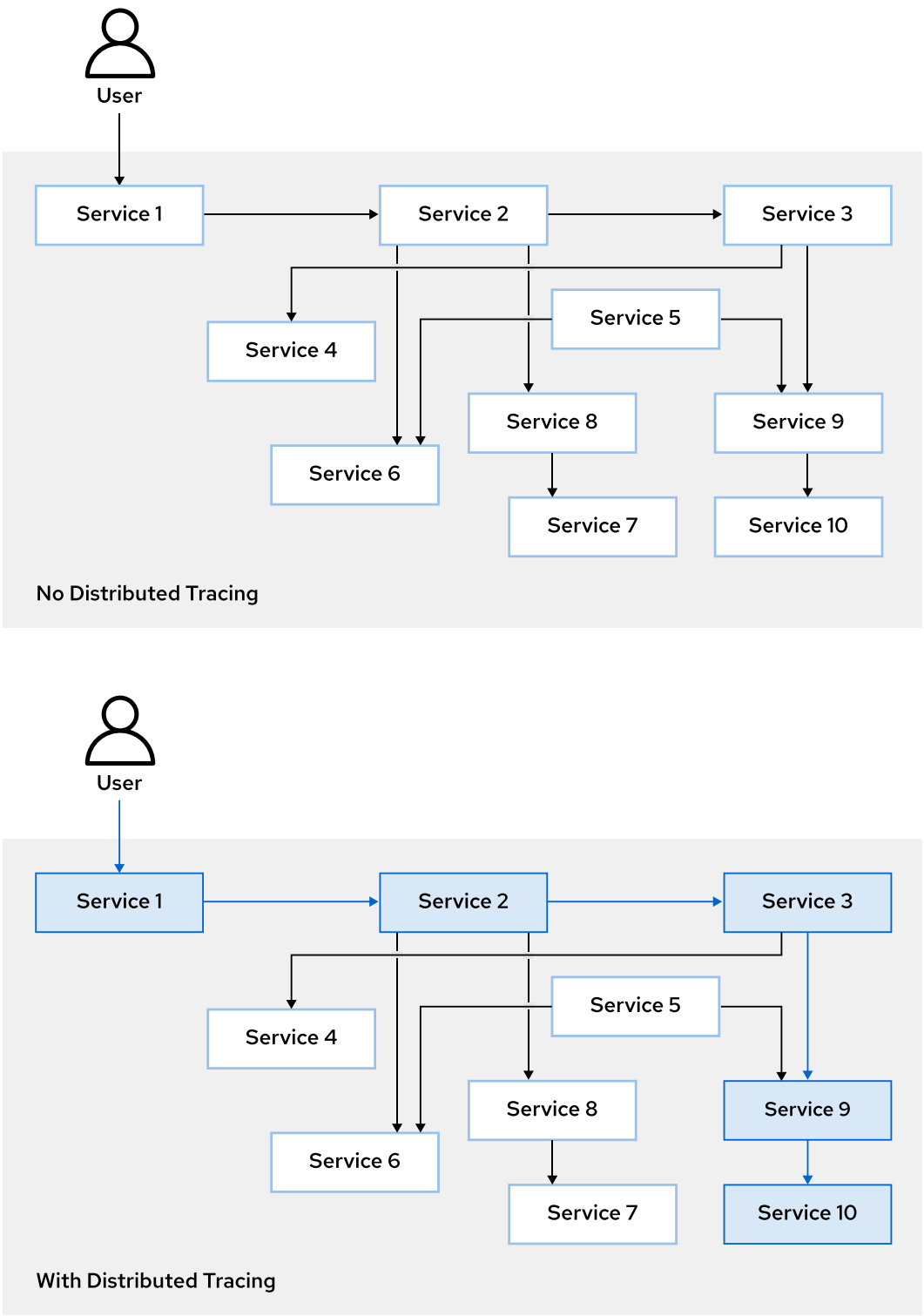


Figure 3.1: Distributed Tracing

In a microservices enabled architecture with a large number of individual microservices, a single client request that achieves a certain business requirement might involve calling multiple individual microservices in a particular sequence. An important aspect of maintaining and developing a distributed system is troubleshooting performance issues. Because a single client call can interact

with multiple services, analyzing the debugging logs of an individual service might not help troubleshooting performance issues.

Distributed tracing allows developers to visualize call flows in a microservices application. Understanding the sequence of calls (how many calls occur in a serial fashion versus how many occur in parallel), and sources of latency is useful when maintaining a distributed system.

For example, if a request takes too long, causing performance issues, then identify the service or services causing the slowdown and examine the network latency between service calls.

Distributed tracing is useful for monitoring, network profiling, and troubleshooting the interaction between services in modern, cloud-native, microservices-based applications.

Traces and Spans in Distributed Tracing

In the context of distributed tracing, it is important to understand two terms:

Span
: A *Span* represents a logical unit of work, which has a unique name, a start time, and the duration of execution. To model the service call flow in a service mesh, spans are nested and executed in a particular order.

Trace
: A *Trace* is an execution path of services in the service mesh. A trace is comprised of one or more spans.

Consider an application with the following microservices:

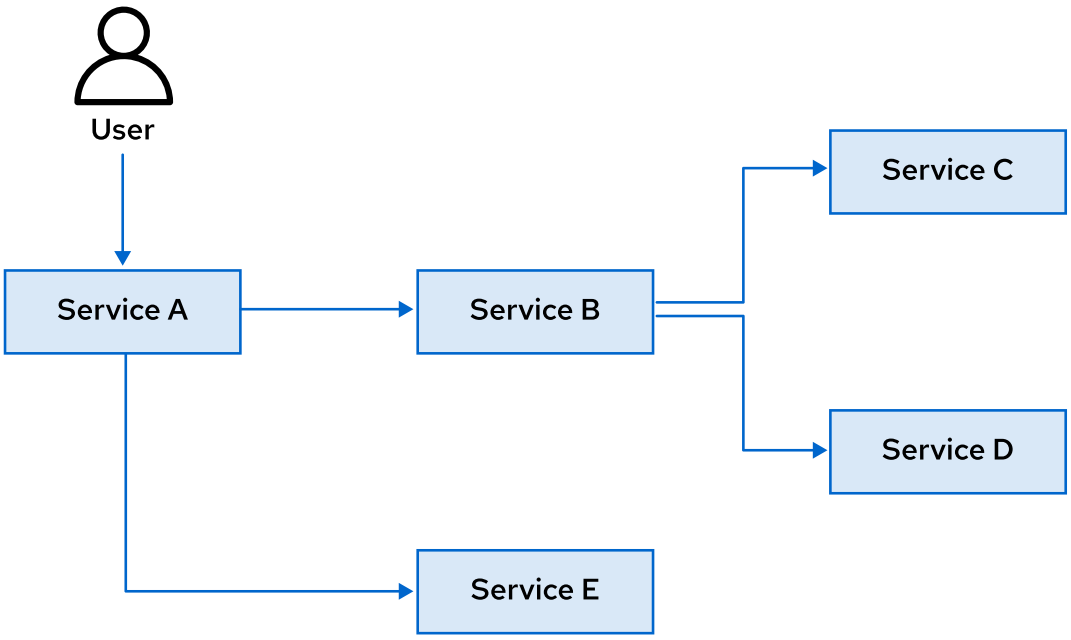


Figure 3.2: Request Call Path

In this example:

- **`Service A`** is the request entry point for the application.

- Because Service A is the entry point for the application, it is called the **`parent span`**. As shown in *Figure 3.2*, **`Service A`** makes two service calls: one to **`Service B`** and one to **`Service E`**. Thus, **`Service B`** and **`Service E`** are **`child spans`** of **`Service A`**.

- **`Service B`** in turn calls **`Service C`** and **`Service D`** before returning the response to **`Service A`**. **`Service B`** is a parent span. **`Service C`** and **`Service D`** are child spans.

The following is a line graph representation of a single trace and its constituent spans:

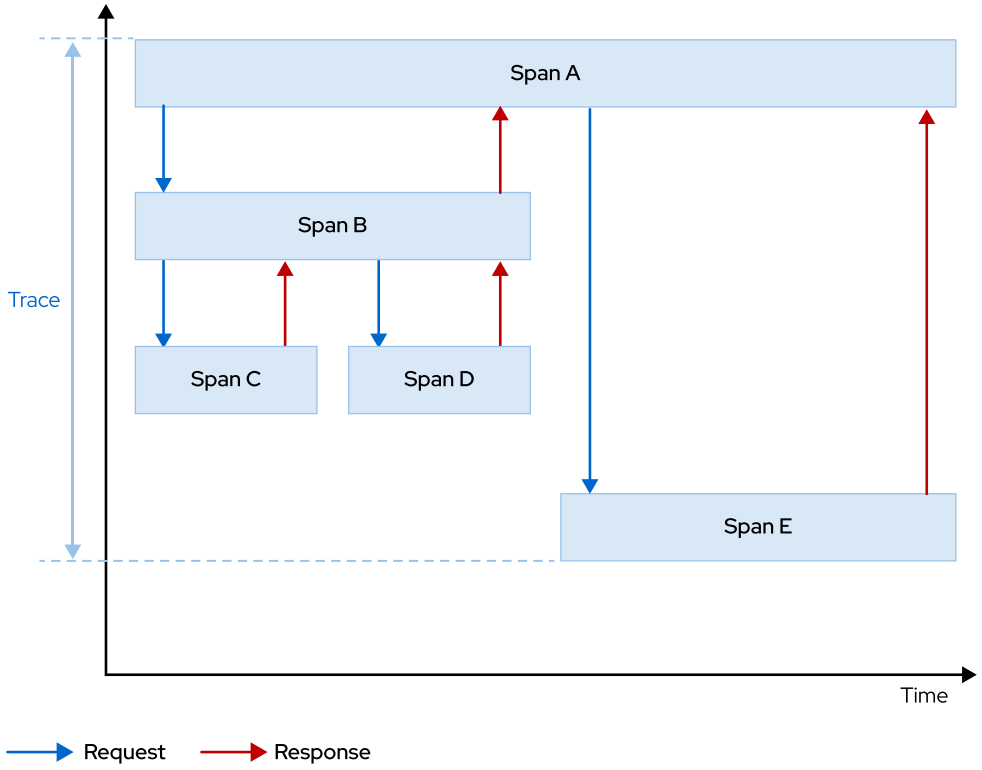


Figure 3.3: Traces and Spans

Introducing Jaeger

Jaeger is a distributed tracing platform. Jaeger allows developers to configure their services to enable gathering runtime statistics about their performance.

Jaeger is installed by default as part of Red Hat OpenShift Service Mesh.

Jaeger is made up of several components that work together to collect, store, and display tracing data.

Jaeger Components

Jaeger Client

Jaeger clients are language specific implementations of the OpenTracing API. They are used to configure applications for distributed tracing. The OpenTracing API defines standard APIs for instrumentation and distributed tracing of microservices applications.

Jaeger Agent

The Jaeger agent is a network daemon that listens for span data sent over User Datagram Protocol (UDP), which it batches and sends to the collector. The agent is meant to be placed

on the same host as the application being traced. This is accomplished by the Envoy proxy sidecar in container environments like OpenShift.

Jaeger Collector
The Collector receives runtime statistics from the agent and places them in an internal queue for processing. This allows the collector to return a response immediately to the agent.

Storage
Collectors require a persistent storage back end. Jaeger has a pluggable mechanism for storage. For Red Hat OpenShift Service Mesh, the only supported storage is **`Elasticsearch`**, which is a distributed search and analytics engine for all types of data.

Query
Query is a service that retrieves runtime statistics from storage.

Jaeger Console
Jaeger provides a web based console that lets you visualize your distributed tracing data. The Jaeger web console is tightly integrated with the OpenShift web console. Using the console, you can trace the path of requests as they flow through the services in a mesh.

Trace Context Propagation

A system with a large number of microservices interact in numerous ways and cannot be planned upfront, these services typically receive and send multiple requests concurrently. *Trace context propagation* is a process which tracks unique requests throughout the call paths in the service mesh.

A new span is generated for each logical service call in the request. This span contains the same request id, a new span id, and the parent span id (which points to the span id of the parent span). Spans are placed on a timeline, and visualized using graphical representations, based on timestamps and durations.

Red Hat OpenShift Service Mesh uses a standard set of HTTP headers for trace context propagation. The Envoy proxy sidecar tracks these headers and forwards them to Jaeger for storage and analysis. A service needs to collect and propagate the following headers from the incoming request to any outgoing requests:

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

Enabling Distributed Tracing in Quarkus Applications

Envoy proxies are configured by default to propagate tracing related headers for traffic flowing into the service mesh. You must explicitly enable tracing in your applications to generate traces and spans and to propagate context information.

Enabling tracing for Quarkus based applications is very simple. Quarkus supports tracing with minimum source code changes.

For example, to enable tracing in applications based on the Quarkus framework, the following steps are required:

- Include the **`quarkus-smallrye-opentracing`** dependency in the project Maven **`pom.xml`** file. This dependency implements the OpenTracing API, an open standard based on Jaeger for distributed tracing.

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-smallrye-opentracing</artifactId>
</dependency>
```

- Enable tracing related properties in the Quarkus **`application.properties`** file. This file is used to externalize configuration for Quarkus applications.

```
quarkus.jaeger.service-name=myservice (1)
quarkus.jaeger.sampler-type=const (2)
quarkus.jaeger.sampler-param=1 (3)
quarkus.jaeger.endpoint=http://jaeger-collector.istio-system.svc:14268/api/
traces (4)
quarkus.jaeger.propagation=b3 (5)
quarkus.jaeger.reporter-log-spans=true (6)
```

(1) A unique service name to identify traces and spans sent to Jaeger. You will be able to identify traces and spans using this name in the Jaeger web console.

(2) Indicates the rate at which traces and spans should be collected from this application. Options include; **`const`** (collect all samples), **`probabilistic`** (random sampling), **`ratelimiting`** (collect samples at a configurable rate per second), and **`remote`** (control sampling from a central Jaeger backend). Refer to https://www.jaegertracing.io/docs/1.17/sampling/ for more details.

(3) Indicates the percentage of requests for which spans should be collected. It should value between 0 and 1. For **`const`** sampling type 0 indicates no sample collection, while a 1 indicates collecting all samples (100%). This value should not be set to 1 in production environments with a large number of requests. This is usually set to 1 in development and QA environments to debug and troubleshoot inter-service latency and performance issues.

(4) The URL of the Jaeger collector. In Red Hat OpenShift Service Mesh, the Jaeger collector URL is **`http://jaeger-collector.istio-system.svc:14268/api/traces`**.

(5) Indicates Jaeger to propagate all **`x-b3-*`** related headers. This is required to construct the parent-child relationships in the call graph. Failure to set this will not show the parent-child relationships between spans in the Jaeger web console.

(6) Log span information for all incoming and outgoing traffic from the application.

- The first two steps are enough to enable tracing for all classes and methods in the application. However, for more complex applications you can control which classes and methods must be enabled for tracing.

  Annotate classes with **`@Traced`** annotation to enable tracing for the entire class. You can also add this annotation at a method level to enable tracing for specific methods and exclude the rest.

Note

The above steps are enough to enable distributed tracing for Quarkus applications. All incoming and outgoing traffic is traced and the context propagation of relevant headers is taken care of automatically.

To customize the tracing (for example, add more contextual information to traces and spans using tags and other metadata), you can use the OpenTracing API. Refer to the OpenTracing documentation at https://opentracing.io/guides/java/ for more details.

Enabling Distributed Tracing in Node.js Applications

To enable distributed tracing for Node.js based applications, you should use the **`jaeger-client`** and **`opentracing`** NPM packages.

You must add code to your application to start traces and spans. You must manually control the context propagation by manipulating the HTTP headers of incoming and outgoing traffic in an application. Parent child relationships are explicitly controlled by the developer.

To enable distributed tracing for Node.js applications, do the following:

- Install the **`jaeger-client`** and **`opentracing`** NPM packages.

```
[user@demo ~]$ npm install --save \
> jaeger-client opentracing
```

- Import the Jaeger client library.

```
const initJaegerTracer = require("jaeger-client").initTracer;
```

- Configure and initialize the Jaeger tracer

```
const config = {
  serviceName: myservice,
  sampler: {
    type: "const",
    param: 1,
  },
  reporter: {
    collectorEndpoint: 'http://jaeger-collector.istio-system.svc:14268/api/
traces',
    logSpans: true
  },
};
const options = { } ;
const tracer = initJaegerTracer(config, options);
```

Note

The configuration options are similar to the ones outlined in the previous section about configuring Quarkus applications using the **`application.properties`** file. Refer to the call out list in the previous section for details about the configuration parameters.

- Start a new span, for example at the start of a function. Pass in a suitable string identifier as argument. The Jaeger web console displays spans and will show the corresponding identifier for each span.

```
const span = tracer.startSpan("mymethod");
```

  You can add contextual information to this span using the **`setTag()`** method. You can pass any relevant object as an argument, which will help you troubleshoot issues as the request flows across services in the mesh. You can add multiple tags to a span.

```
span.setTag("mymethod", "some-message");
```

- Once the function finishes executing, invoke the **`span.finish()`** function to end the span.

```
span.finish();
```

Context Propagation and Child Spans

When you create a new span using the **`startSpan()`** method, it creates a new root span by default. A request can call multiple services in a certain order. You must link these different calls using parent child relationships so that the Jaeger web console can display the appropriate service call graph, and help you trace the call flow.

To declare a span as a child of another span, add the **`childOf`** property with a value of the parent span to the **`startSpan()`** method.

```
const childSpan = tracer.startSpan("another-method", { childOf: span });
```

For any non-trivial Node.js application, with code organized in modules spread across multiple files, a side effect of the above code is the need to keep passing the span object around. This is required to keep building the call flow and maintain parent child relationships between spans.

A better approach is to create a context object and encapsulate the span inside it. You can then pass the context object around, and use it to store other application related data as well.

```
const ctx = { span };
ctx = {
  span: tracer.startSpan("mymethod", { childOf: ctx.span }),
};
```

Note that context propagation as outlined previously will work only between method calls running in the same Node.js runtime process.

Passing contextual information between completely isolated microservices connected by a network brings more challenges. The OpenTracing API provides some helper methods to solve this problem.

You can inject the contextual information using HTTP headers to outgoing traffic, and then extract contextual information from requests coming into the application. The following code illustrates this:

```
const { Tags, FORMAT_HTTP_HEADERS } = require('opentracing');
...code omitted...

const method = 'GET';
const headers = {};
const url = "some-remote-URL"

span.setTag(Tags.HTTP_URL, url);
span.setTag(Tags.HTTP_METHOD, method);
span.setTag(Tags.SPAN_KIND, Tags.SPAN_KIND_RPC_CLIENT);

tracer.inject(span, FORMAT_HTTP_HEADERS, headers);
```

You can then invoke other remote services, for example, using the NPM **`request-promise`** module and pass on the HTTP headers.

```
request({url, method, headers})
            .then( data => {
                span.finish();
                return data;
            }, e => {
                span.finish();
                throw e;
            });
```

For incoming traffic, you can extract the contextual information using the **`tracer.extract()`** method as per the following.

```
const { Tags, FORMAT_HTTP_HEADERS } = require('opentracing');
...code omitted...

const parentSpanContext = tracer.extract(FORMAT_HTTP_HEADERS, req.headers);
const span = tracer.startSpan('another-method', {
        childOf: parentSpanContext,
        tags: {[Tags.SPAN_KIND]: Tags.SPAN_KIND_RPC_SERVER}
    });
```

With this approach, trace, span and context information is maintained across service calls in a service mesh.

Note

You can also configure Jaeger using environment variables. Support for configuration through environment variables differs based on the client implementation. See https://www.jaegertracing.io/docs/1.17/client-features/ for more details.

You can declare these environment variables in Dockerfiles, but it is recommended to use configuration maps or secrets to inject these variables to comply with the 12-factor methodology.

Viewing Traces and Spans Using the Jaeger Web Console

The Jaeger web console is installed by default with Red Hat OpenShift Service Mesh and is tightly integrated with the OpenShift web console. To view details about traces and spans in the Jaeger console, do the following:

1. In the OpenShift web console, navigate to **Networking** → **Routes** and search for the **`jaeger`** route, which is the URL listed in the **`Location`** column.

2. Log in using the same user name and password that is used to access the OpenShift web console. You should see the Jaeger web console home page.

3. In the left pane of the Jaeger console, from the **`Service`** menu, select your application and click **`Find Traces`** at the bottom of the pane. A list of traces gathered for the application are displayed.

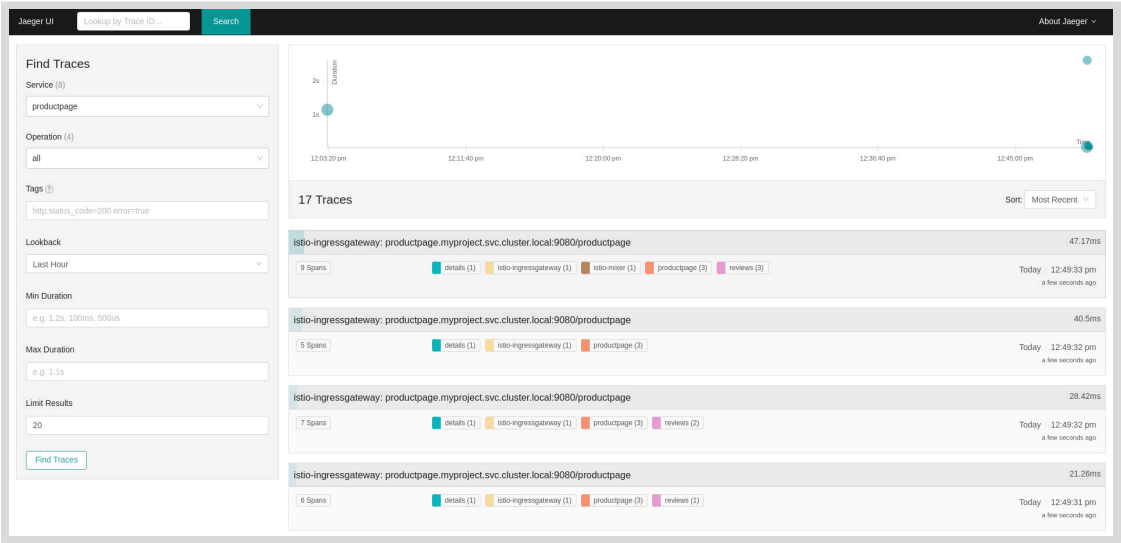


Figure 3.4: List of Traces

4. Click one of the traces in the list to open a detailed view of that trace.

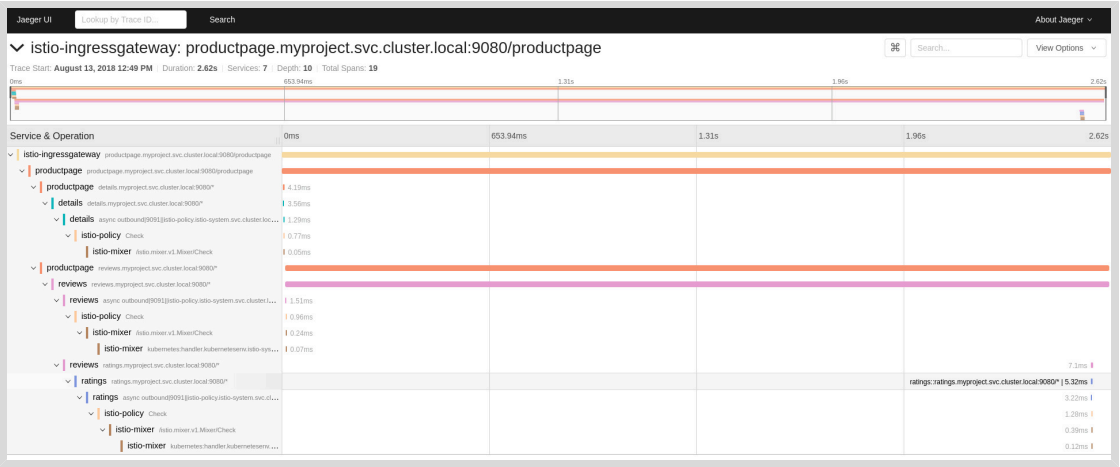


Figure 3.5: Trace Details

References

For more information, refer to the *Understanding Jaeger* section in the *Red Hat Service Mesh Guide* at
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.4/html-single/service_mesh/index#ossm-jaeger

Istio Distributed Tracing
https://archive.istio.io/v1.4/docs/tasks/observability/distributed-tracing/

Guided Exercise

Tracing Services with Jaeger

In this exercise, you will deploy two microservices on OpenShift Service Mesh, and trace the service communication using Jaeger.

- **`servicea`**: written in JavaScript using the Node.js runtime.
- **`serviceb`**: written in Java using the Quarkus framework.

Traffic enters the service mesh through **`servicea`**. **`servicea`** calls **`serviceb`** and returns a response.

You will do the following in this exercise:

1. Enable distributed tracing in both microservices using Jaeger.
2. Build container images locally for both microservices using **`podman`**.
3. Push the built container images to the **`Quay.io`** public container registry.
4. Deploy both microservices to the service mesh, and trace the service calls between the two microservices.

Outcomes

You should be able to deploy applications to OpenShift Service Mesh, and trace the path of the service calls for requests entering the service mesh.

Before You Begin

To perform this exercise, ensure you have access to:

- A configured and running OpenShift cluster.
- An installed and running OpenShift Service Mesh in the OpenShift cluster.
- The OpenShift CLI (**`/usr/local/bin/oc`**).
- An account with the **`Quay.io`** container registry, and **`podman`** installed locally on your workstation.

As the **`student`** user on the **`workstation`** machine, use the **`lab`** command to validate the prerequisites for this exercise:

```
[student@workstation ~]$ lab observe-jaeger start
```

1. Clone the source code for the microservices from GitHub. Inspect the source code for both microservices.

Use a text editor like **`VSCodium`**, which supports syntax highlighting for editing JavaScript and Java source files.

1.1. Open a new terminal window on your workstation. From the home directory, clone the source code for the microservices from GitHub.

```
[student@workstation ~]$ git clone https://github.com/RedHatTraining/DO328-apps
...output omitted...
Cloning into 'DO328-apps'...
...output omitted...
```

1.2. Copy the contents of the **/home/student/DO328-apps/tracing-ge** folder from your local Git repository to the **/home/student/DO328/labs/observe-jaeger** folder.

```
[student@workstation ~]$ cp -Rv ~/DO328-apps/tracing-ge/* \
> ~/DO328/labs/observe-jaeger/
```

You should now see two folders called **servicea** and **serviceb** in the **/home/student/DO328/labs/observe-jaeger/** folder.

You should also see shell scripts and service mesh related YAML files in the same folder. These were created by the lab start script.

1.3. Review the source code for **servicea** in the **/home/student/DO328/labs/observe-jaeger/servicea/index.js** file.

This microservice exposes a single HTTP GET endpoint, which calls **serviceb** and returns a response to the client.

```
...output omitted...
server.get("/", async (request) => {
    const { rootSpan } = request;
    const msg = await serviceb.callServiceB(rootSpan);

    return 'Hello from ServiceA!.\nResponse from ServiceB => ' + msg + '\n';
});
...output omitted...
```

1.4. Review the source code for **serviceb** in the **/home/student/DO328/labs/observe-jaeger/serviceb/src/main/java/com/redhat/training/serviceb/ServiceB.java** file.

This microservice contains a single HTTP GET endpoint that returns a string.

```
...output omitted...
String message = "Hello from ServiceB!";

@GET
@Produces(MediaType.TEXT_PLAIN)
public String sayHello() {
  return message;
}
...output omitted...
```

2. Enable tracing for **servicea** using the **Jaeger** and **OpenTracing** libraries.

2.1. Navigate to the **/home/student/DO328/labs/observe-jaeger/servicea** folder on the command line terminal. All references to file paths in this step are relative to this folder.

```
[student@workstation ~]$ cd ~/DO328/labs/observe-jaeger/servicea
```

2.2. Inspect the **package.json** file, which declares all the NPM packages required for running this microservice.

Install the packages declared in **package.json**, followed by installing the **opentracing** and **jaeger-client** dependencies.

```
[student@workstation servicea]$ npm install
[student@workstation servicea]$ npm install --save \
> jaeger-client@3.17.2 opentracing@0.14.4
...output omitted...
+ opentracing@0.14.4
+ jaeger-client@3.17.2
...output omitted...
```

2.3. Edit the **index.js** file.

Add the default value for the **TRACE_COLLECTOR_URL** variable. This should be the URL of the Jaeger collector that is running in the service mesh.

```
const TRACE_COLLECTOR_URL = ...output omitted... || "http://jaeger-
collector.istio-system.svc:14268/api/traces";
```

Note how the collector URL is used to initialize the tracer.

```
const tracer = Tracer.create("servicea", TRACE_COLLECTOR_URL, logger);
```

The service name **servicea** is also passed to the **Tracer.create()** method, which indicates the starting point of the trace.

2.4. Edit the **Tracer.js** file, which configures the tracer properties for this microservice.

Add the parameters for the **config** and **reporter** properties in the **create** function as follows:

```
  const config = {
    serviceName: serviceName,
    sampler: {
      type: "const",
      param: 1,
    },
    reporter: {
      logSpans: true,
      collectorEndpoint
    }
  };
```

You can also copy the code from the solution file **/home/student/DO328/solutions/observe-jaeger/servicea/Tracer.js**.

Note the use of HTTP headers to propagate the trace context using the **tracer.registerInjector()** and **tracer.registerExtractor()** methods.

```
tracer.registerInjector(FORMAT_HTTP_HEADERS, codec);
tracer.registerExtractor(FORMAT_HTTP_HEADERS, codec);
```

2.5. Edit the **`HttpServer.js`** file. Note the callback methods, **`traceRequest`** and **`traceResponse`** registered as hooks to the server. These methods are called after every HTTP request to the microservice, and before sending the response to the client respectively.

Edit the **`traceRequest`** method and add code to create a new root span with a unique string id. Add Opentracing tags to the span to identify the original URL of the request and the HTTP method (GET, POST, PUT, DELETE and more).

```
const span = tracer.startSpan(`${method}:servicea`);
span.setTag(Opentracing.Tags.HTTP_URL, originalUrl);
span.setTag(Opentracing.Tags.HTTP_METHOD, method);
```

You can also copy the code from the solution file **`/home/student/DO328/solutions/observe-jaeger/servicea/HttpServer.js`**.

2.6. Briefly review the **`Services/ServiceB.js`** file. Do not make any changes to this file. The **`get()`** method is invoked on every request to the root URL of the microservice ("/").

The implementation is inherited from the parent **`RestClient`** class. Edit the **`Services/RestClient.js`** file.

Create a new span for the **`get()`** method, which is a child span of the root span. Add Tags to indicate which endpoint is being called, as well as the caller URL, the HTTP method, and the type of span.

```
const span = this.tracer.startSpan(spanName, { childOf: rootSpan.context() });
span.setTag(Tags.PEER_HOSTNAME, this.baseURL);
span.setTag(Tags.HTTP_URL, url);
span.setTag(Tags.HTTP_METHOD, "GET");
span.setTag(Tags.SPAN_KIND, Tags.SPAN_KIND_RPC_CLIENT);
```

Inject the span data as HTTP headers to propagate the span context. Add the following code to the **`_buildAxiosRequestConfig()`** method.

```
const headers = {};
this.tracer.inject(span, FORMAT_HTTP_HEADERS, headers);
return { headers };
```

You can also copy the code from the solution file **`/home/student/DO328/solutions/observe-jaeger/servicea/Services/RestClient.js`**.

2.7. Save your changes. To ensure that there are no syntax errors, run **`npm start`** and ensure that the microservice starts without any errors.

If there are errors, then compare your changes with the solution files in the **`/home/student/DO328/solutions/observe-jaeger/servicea`** folder.

```
[student@workstation servicea]$ npm start

> servicea@1.0.0 start ...output omitted...
> node index.js

...output omitted...: "Initializing Jaeger Tracer ...output omitted...
...output omitted...: "Server listening at http://0.0.0.0:8080"}
```

Press **Ctrl**+**C** to stop the server.

3. Enable tracing for **serviceb** using the **quarkus-smallrye-opentracing** library.

3.1. Change to the **/home/student/DO328/labs/observe-jaeger/serviceb** folder. All references to file paths in this step are relative to this folder.

```
[student@workstation ~]$ cd ~/DO328/labs/observe-jaeger/serviceb
```

3.2. Inspect the maven **pom.xml** file, which declares the dependencies for this microservice.

Include the **quarkus-smallrye-opentracing** dependency for enabling tracing.

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-smallrye-opentracing</artifactId>
</dependency>
```

You can also copy the code from the solution file **/home/student/DO328/solutions/observe-jaeger/serviceb/pom.xml**.

3.3. Briefly review the code in the **src/main/java/com/redhat/training/serviceb/ServiceB.java** file. Do not make any changes to this file.

Tracing is automatically enabled by adding the **quarkus-smallrye-opentracing** maven dependency, and then enabling some jaeger related environment variables in the Quarkus **application.properties** file.

3.4. Edit the **src/main/resources/application.properties** file and add the jaeger related properties.

```
quarkus.jaeger.service-name=serviceb
quarkus.jaeger.sampler-type=const
quarkus.jaeger.sampler-param=1
quarkus.log.console.format=%d{HH:mm:ss} %-5p traceId=%X{traceId}, spanId=
%X{spanId}, sampled=%X{sampled} [%c{2.}] (%t) %s%e%n
quarkus.jaeger.endpoint=http://jaeger-collector.istio-system.svc:14268/api/traces
quarkus.jaeger.propagation=b3
quarkus.jaeger.reporter-log-spans=true
```

You can also copy the code from the solution file **/home/student/DO328/solutions/observe-jaeger/serviceb/src/main/resources/application.properties**.

3.5. Save your changes. To ensure that there are no syntax errors, run **mvn clean package** and ensure that a fat JAR is created in the **target** folder.

```
[student@workstation serviceb]$ mvn clean package
...output omitted...
[INFO] Building serviceb 1.0.0
...output omitted...
...output omitted... Building fat jar: observe-jaeger/serviceb/target/
serviceb-1.0.0-runner.jar
[INFO] [io.quarkus.deployment.QuarkusAugmentor] Quarkus augmentation completed in
 2440ms
...output omitted...
[INFO] BUILD SUCCESS
...output omitted...
```

3.6. Start the microservice and verify that there are no errors.

```
[student@workstation serviceb]$ java -jar target/serviceb-1.0.0-runner.jar
...output omitted... (powered by Quarkus 1.3.2.Final) started in 0.747s.
Listening on: http://0.0.0.0:8080
...output omitted...
```

Verify that there are no errors. If there are errors, then compare your changes with the solution files in the **/home/student/DO328/solutions/observe-jaeger/serviceb** folder. Press **Ctrl**+**C** to stop the server.

4. Build container images for both microservices using **podman**.

4.1. Load your classroom environment configuration.

Run the following command to load the environment variables:

```
[student@workstation serviceb]$ source /usr/local/etc/ocp4.config
```

4.2. Review the **Dockerfile** for **servicea**. Use Red Hat Universal Base Images (UBI) as the base for building your container image. Do not make any changes to the Dockerfile.

4.3. Build the container image for **servicea** using **podman**.

```
[student@workstation serviceb]$ cd ~/DO328/labs/observe-jaeger/servicea
[student@workstation servicea]$ podman build -t \
> quay.io/${RHT_OCP4_QUAY_USER}/ossm-tracing-servicea:1.0 .
STEP 1: FROM registry.access.redhat.com/ubi8/nodejs-12
...output omitted...
STEP 13: COMMIT quay.io/youruser/ossm-tracing-servicea:1.0
```

4.4. Similarly, build the container for **serviceb** after reviewing the **Dockerfile**.

```
[student@workstation servicea]$ cd ~/DO328/labs/observe-jaeger/serviceb
[student@workstation serviceb]$ podman build -t \
> quay.io/${RHT_OCP4_QUAY_USER}/ossm-tracing-serviceb:1.0 .
STEP 1: FROM registry.access.redhat.com/ubi8:8.1
...output omitted...
STEP 15: COMMIT quay.io/youruser/ossm-tracing-serviceb:1.0
```

4.5. Verify that the container images for **servicea** and **serviceb** are built successfully.

```
[student@workstation serverb]$ podman images
REPOSITORY                                    TAG ...output omitted...
quay.io/youruser/ossm-tracing-serviceb        1.0 ...output omitted...
quay.io/youruser/ossm-tracing-servicea        1.0 ...output omitted...
...output omitted...
```

5. Create new public image repositories in Quay.io to store the newly built container images. Push the container images to **Quay.io**.

5.1. Create two new public container image repositories called **ossm-tracing-servicea**, and **ossm-tracing-serviceb** in Quay.io. Refer to the instructions in *Creating a Quay Account* for creating public container image repositories.

Warning

If you skip this step and push the container images without creating public repositories, then the **podman push** commands create private container image repositories by default.

5.2. Login to your Quay.io account using **podman**.

```
[student@workstation serverb]$ podman login -u ${RHT_OCP4_QUAY_USER} quay.io
```

You will be prompted for your Quay.io password.

5.3. Push the container image for **servicea** to Quay.io.

```
[student@workstation serverb]$ podman push \
> quay.io/${RHT_OCP4_QUAY_USER}/ossm-tracing-servicea:1.0
...output omitted...
Writing manifest to image destination
Storing signatures
```

5.4. Push the container image for **serviceb** to Quay.io.

```
[student@workstation serverb]$ podman push \
> quay.io/${RHT_OCP4_QUAY_USER}/ossm-tracing-serviceb:1.0
...output omitted...
Writing manifest to image destination
Storing signatures
```

6. Create the **tracing** project and and then add it to the **ServiceMeshMemberRoll** resource.

6.1. Log in to OpenShift as the **developer** user.

```
[student@workstation serverb]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_API}
Login successful.
...output omitted...
```

6.2. Create the **tracing** project.

```
[student@workstation serverb]$ oc new-project tracing
Now using project "tracing" on server
"https://api.cluster.domain.example.com:6443".
...output omitted...
```

6.3. Add the **tracing** project to the list of members in the **ServiceMeshMemberRoll** resource. Edit the **default** ServiceMeshMemberRoll resource in the OpenShift web console and add the **tracing** project to the member list.

```
apiVersion: maistra.io/v1
kind: ServiceMeshMemberRoll
metadata:
...output omitted...
spec:
  members:
  - tracing
...output omitted...
```

You can also run the **add-project-to-smmr.sh** script in the **/home/student/DO328/labs/observe-jaeger** folder to add the **metrics** project to the list of members in the **ServiceMeshMemberRoll** resource.

```
[student@workstation serverb]$ cd /home/student/DO328/labs/observe-jaeger
[student@workstation observe-jaeger]$ oc patch servicemeshmemberroll/default \
> -n istio-system --type=merge \
> -p '{"spec": {"members": ["tracing"]}}'
servicemeshmemberroll.maistra.io/default patched
```

Note

You can also use the **oc edit smmr default -n istio-system** command and add the **tracing** project to the member list.

7. Deploy the microservices to OpenShift service mesh.

7.1. Edit the ***-deploy.yaml** files in the **/home/student/DO328/labs/observe-jaeger** folder, which describes the necessary resources to deploy both applications.

The deployment files have the **sidecar.istio.io/inject: "true"** annotation included to inject the Envoy proxy after deployment.

7.2. Edit the **/home/student/DO328/labs/observe-jaeger/servicea-deploy.yaml** file. Edit the **spec.template.spec.containers.image** attribute and add the Quay.io URL of the newly built container image.

```
...output omitted...
spec:
  containers:
  - name: servicea
    image: quay.io/youruser/ossm-tracing-servicea:1.0
    imagePullPolicy: IfNotPresent
...output omitted...
```

7.3. Edit the **/home/student/DO328/labs/observe-jaeger/serviceb-deploy.yaml** file. Add the Quay.io URL of the container image for **serverb**.

```
...output omitted...
spec:
  containers:
  - name: serviceb
    image: quay.io/youruser/ossm-tracing-serviceb:1.0
    imagePullPolicy: IfNotPresent
...output omitted...
```

7.4. Run the **oc create** command to deploy the applications.

```
[student@workstation observe-jaeger]$ oc create -f servicea-deploy.yaml
deployment.apps/servicea created
service/servicea created
[student@workstation observe-jaeger]$ oc create -f serviceb-deploy.yaml
deployment.apps/serviceb created
service/serviceb created
```

7.5. Run the **oc get pods** command and verify that both microservices are deployed and in **Running** state.

```
[student@workstation observe-jaeger]$ oc get pods
NAME                        READY   STATUS    RESTARTS   AGE
servicea-6c9fffcc58-v699r   2/2     Running   0          10m
serviceb-78489f94bf-z4vdh   2/2     Running   0          10m
```

7.6. Inspect the **/home/student/DO328/labs/observe-jaeger/gateway.yaml** file, which describes the ingress gateway for traffic entering the mesh.

Use the **oc create** command to create the ingress gateway.

```
[student@workstation observe-jaeger]$ oc create -f gateway.yaml
gateway.networking.istio.io/observe-jaeger-gateway created
```

7.7. Create a **VirtualService** to redirect the ingress traffic to **servicea**, which acts as an entry point into the mesh.

Examine the **virtual-service.yaml** file, which routes the ingress traffic to **servicea**.

Use the **oc create** command to create the virtual service.

```
[student@workstation observe-jaeger]$ oc create -f virtual-service.yaml
virtualservice.networking.istio.io/observe-jaeger-vs created
```

8. Test the microservices.

8.1. Run the **`oc get route`** command to get the URL of the Istio gateway.

You can also cut and paste the full command from the **`get-ingress-gateway-url.sh`** file.

Export the ingress gateway URL to an environment variable called **`GATEWAY_URL`**.

```
[student@workstation observe-jaeger]$ export  \
> GATEWAY_URL=$(oc get route istio-ingressgateway -n istio-system \
> -o template --template '{{ "http://" }}{{ .spec.host }}')
```

8.2. Execute the **`curl`** command in combination with the **`GATEWAY_URL`** variable to access the application.

```
[student@workstation observe-jaeger]$ curl ${GATEWAY_URL}/trace
Hello from ServiceA!.
Response from ServiceB => Hello from ServiceB!
```

9. Visualize traces generated by the microservices using the Jaeger web console.

9.1. Run the **`oc get route`** command to gather the Jaeger web console URL. You can also copy the commands from the **`get-jaeger-url.sh`** file.

```
[student@workstation observe-jaeger]$ export  \
> JAEGER_URL=$(oc get route jaeger -n istio-system \
> -o template --template '{{ "https://" }}{{ .spec.host }}')
```

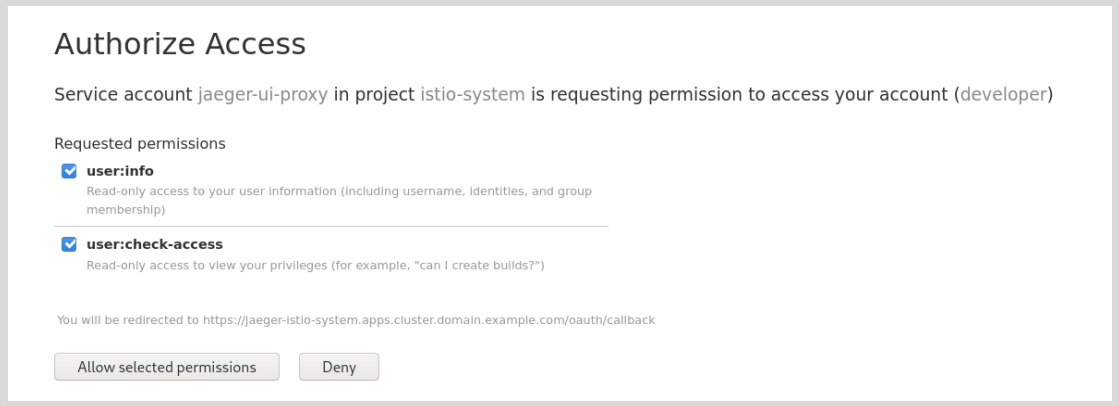
9.2. Use Firefox web browser to access the Jaeger web console.

```
[student@workstation observe-jaeger]$ firefox ${JAEGER_URL} &
```

9.3. Click **Log in with OpenShift**.

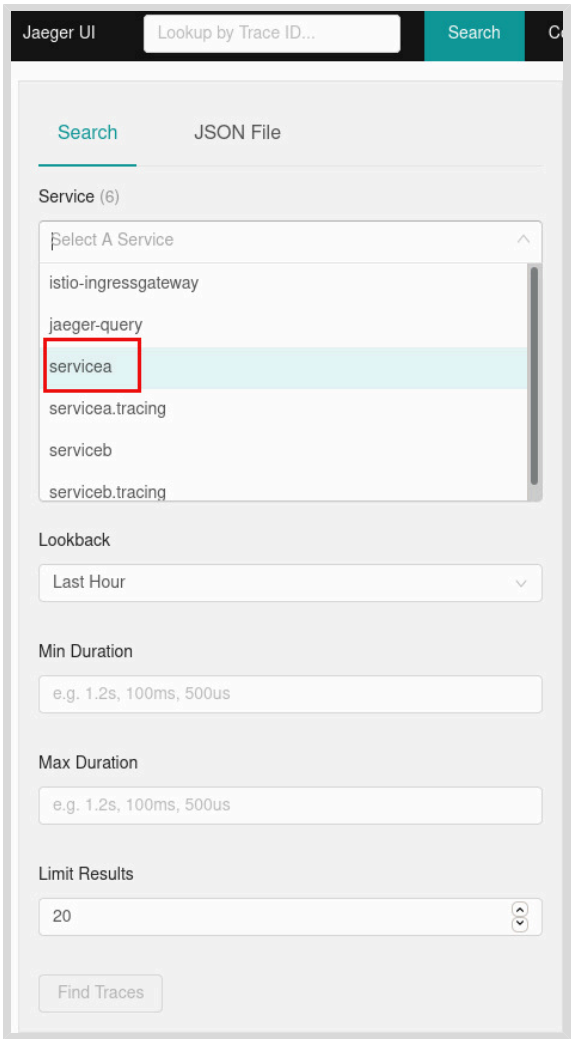
Log in using your developer user account. Your user name is the **`RHT_OCP4_DEV_USER`** variable in the **`/usr/local/etc/ocp4.config`** classroom configuration file. Your password is the **`RHT_OCP4_DEV_PASSWORD`** variable in the same file.

If you are accessing the Jaeger web console for the first time, then you will be prompted with a page asking you to authorize service account access to your account. Click **Allow selected permissions** to bring up the Jaeger web console.



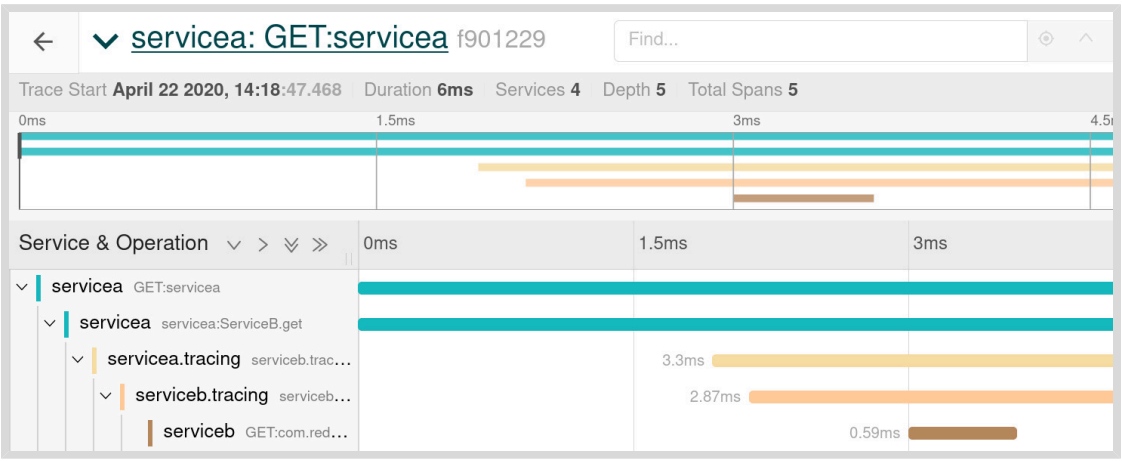
9.4. Execute the **`curl`** command against the URL **`${GATEWAY_URL}/trace`** multiple times to generate some load and allow the microservices to send traces and span information to Jaeger.

9.5. In the Jaeger web console, refresh the page and then select the **`servicea`** in the left panel. Click **Find Traces** to retrieve the traces.

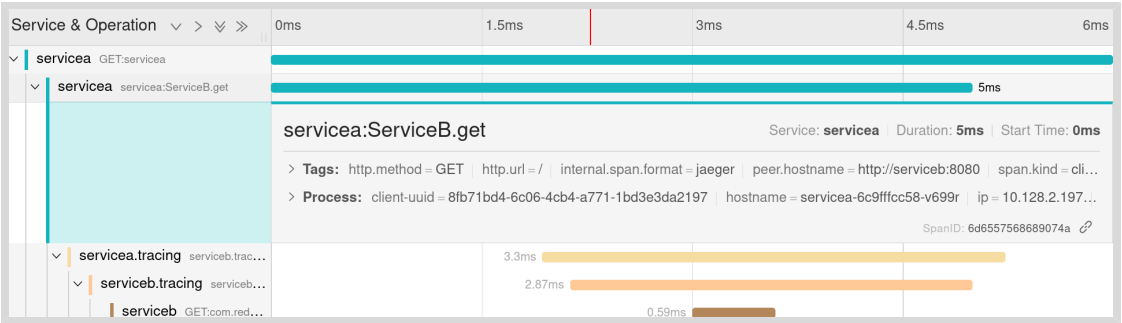


9.6. You should see a number of traces displayed, corresponding to the number of requests that you made to the service mesh.

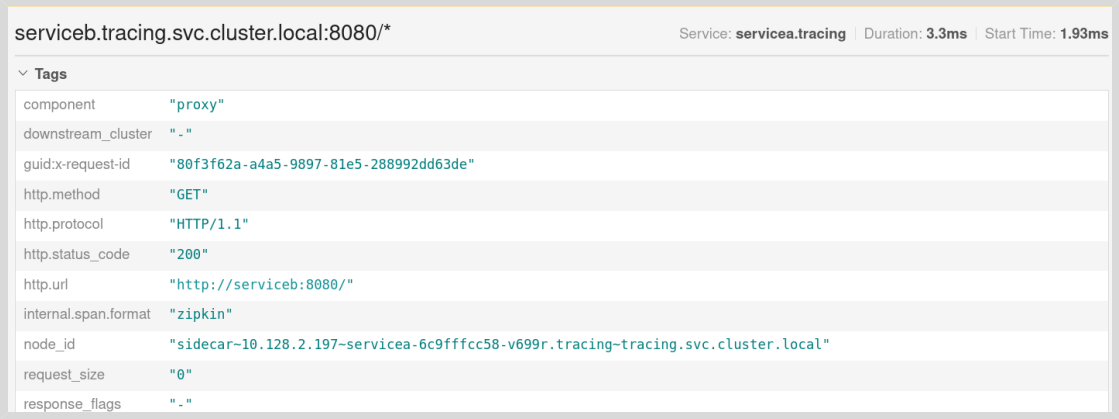
Click any one trace and observe the spans reported by both microservices. Note how the spans from **`serviceb`** are shown as child spans of **`servicea`**.



You can click each of the spans and note the runtime values, which have been propagated through HTTP headers as the traffic flows from **`servicea`** to **`serviceb`**.



Expand **Tags** for the **`servicea.tracing`** span to see contextual span information, which was propagated to services.



Note

You will see extra **`*.tracing`** spans in the Jaeger console. These are propagated by the Envoy proxy as it intercepts traffic bound for the services in the mesh.

10. Return to the home directory.

```
[student@workstation observe-jaeger]$ cd ~
[student@workstation ~]$
```

Finish

On the **`workstation`** machine, use the **`lab`** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab observe-jaeger finish
```

This concludes the guided exercise.

Collecting Service Metrics

Objectives

After completing this section, you should be able to collect and inspect critical metrics with Prometheus and Grafana.

Metrics and Service Level Objectives

In any system that has a large number of services, it is important to understand the different types of metrics to gather, and then decide on a process to measure and evaluate their performance. Each service, or a set of services with common business functionality can define their own set of metrics that should be gathered, measured and analyzed.

A common practice is to define a set of *service levels* that act as a sort of contract, or agreement between a service provider and a service consumer. Service levels can be broadly classified into three categories:

Service Level Indicators (SLI)
An SLI is a carefully defined quantitative measure of some aspect of the level of service that is provided. For example, a common SLI is the **`response time`**, that is, the time taken by a service to provide a response. Other examples of SLI include **`error rate`** (percentage of responses that were invalid), and **`availablity`** (percentage of time that the service was in a correctly functional state).

Service Level Agreements (SLA)
An SLA is an explicit or implicit contract with your users which includes consequences of meeting (or missing) the service level objectives (SLO) for a service. Selecting and publishing an SLA to users sets expectations about how a service will perform. For example, a credit card payment service for an e-commerce website will have an SLA that declares that "All payment requests will be processed in less than 5 seconds."

Service Level Objectives (SLO)
An SLO is a threshold value, or range of values that is measured by an SLI. For example, an E-commerce website can have an SLO that tries to render a product catalog page in 3 to 5 seconds. Another scenario could be for handling a large number of users, for example "The payment service should be capable of handling 30,000 to 35,000 users concurrently on weekends."

Selecting Metrics to Measure

You should carefully select a set of metrics for a service that you want to include in your SLI. Selecting too many metrics wastes monitoring cycles and clutters up your dashboards, while choosing too few, or the wrong metrics will impede your analysis and reduce the effectiveness of your response to issues in the field.

Examples of types of systems, and the metrics that are relevant for measuring their performance are as follows:

- **`End user facing systems/HTTP web APIs:`** Good candidates for SLIs are system uptime (availability), response time (latency), and number of requests served per second (throughput).
- **`Big Data/Machine Learning:`** SLIs such as data throughput (how many items were processed) and end-to-end latency (how much time to process all the data) are the preferred starting points for measurement.
- **`Database/Storage Systems:`** These kinds of systems are concerned with latency (how quickly were items written to disk), availability, and durability (no data corruption).

These are good starting points and not a definitive list of indicators. You must analyze your service and determine what metrics to gather, ideally a small set of core metrics are preferred over monitoring a large number of items. Use an iterative approach to add new metrics and learn from incidents in production.

Service Mesh Metrics (Telemetry)

Red Hat OpenShift Service Mesh gathers detailed metrics (telemetry) for all services within a service mesh. These metrics allow developers to observe, troubleshoot, and optimize the behavior of their applications under load. Developers are able to understand how the services communicate and interact with each other, as well as with the service mesh control plane.

A default installation of Red Hat OpenShift Service Mesh gathers a number service metrics related to error rates, rate of traffic, HTTP status codes of the response, and more. The service mesh also gathers detailed metrics for its control plane. A default set of monitoring dashboards using these metrics is automatically created and provided to developers.

Metrics from Envoy Proxies

The Envoy proxies provide a rich set of metrics about traffic passing through the proxy, both incoming (ingress) and outgoing (egress). The proxies also provide detailed statistics about the functioning of the proxy itself (health status and configuration).

You can customize the set of Envoy proxy metrics that will be collected for a given service mesh. By default, only a small subset of the Envoy proxy metrics are collected. This avoids overloading the system and reduces the CPU overhead associated with metrics collection.

The *References* section has a number of links that detail the metrics that are available for monitoring the Envoy proxies.

Metrics from Application Services

Red Hat OpenShift Service Mesh provides a set of application service metrics for observing the performance and state of incoming and outgoing service traffic.

A default installation of Red Hat OpenShift Service Mesh gathers the following service metrics:

- Request Count : The total number of requests sent to a service.
- Request Duration : The time taken for the service to provide a response.
- Request Size : The size of the body in the HTTP request.
- Response Size : The size of the body in the HTTP response.

The *References* section has a number of links that detail the full list of metrics that are available for monitoring the application services.

Metrics from the Service Mesh Control Plane

Each of the service mesh components (Pilot, Citadel, and Galley) also provide a set of metrics about their health, configuration, and performance.

The *References* section has a number of links that detail the full list of metrics that are available for monitoring the components of the service mesh control plane.

Introducing Prometheus and Grafana

Prometheus is an open-source systems monitoring and alerting toolkit which includes a time-series database for storing metrics. It provides a powerful web based user interface to query and analyze performance trends from the data it collects.

Red Hat OpenShift Service Mesh provides a default Prometheus server instance which gathers metrics data from the Envoy proxies, the services in the service mesh, and the components in the control plane.

Grafana is an open-source graphical visualization tool used for creating operational dashboards for software systems.

Red Hat OpenShift Service Mesh provides a default Grafana instance with ready made dashboards for viewing data from the Envoy proxies, the services in the service mesh, and the components in the control plane. The Grafana instance is tightly integrated with the default Prometheus instance in the service mesh installation and uses the data stored in Prometheus to render the graphs in the dashboard.

Collecting Custom Application Metrics

A default Red Hat OpenShift Service Mesh instance automatically collects several useful metrics for your application. These metrics are useful to get a high-level understanding of the performance of the service mesh and the applications deployed on it.

However, you might sometimes need to gather custom application specific metrics and display them in a Grafana dashboard. For example, in an E-commerce application, you might wish to track how many items of a specific category are sold over a weekend or special holiday sale. Another example is a financial services application that tracks the number of successful and failed transactions of a specific type.

You must include the Prometheus client libraries in your application, and then create these custom metrics. Each metric is categorized into a specific Prometheus metric type. The client libraries will collect all the custom metrics and then send it to Prometheus for storage. The data stored in Prometheus can be visualized using custom Grafana dashboards.

Prometheus Metric Types

Prometheus supports four different types of metrics:

Counter
: A counter is a cumulative metric that represents a single variable whose value can only increase, or be reset to zero on restart. For example, you can use a counter to represent the number of errors, requests served and tasks completed.

Gauge
A gauge is a metric that represents a single numerical value, which can be incremented or decremented. For example, you can use a gauge to represent the number of processes, or the number of concurrent users.

Histogram
A histogram samples values and counts them in configurable buckets. It also provides a sum of all values. Histograms track the number and the sum of the values , allowing you to calculate the average of the values.

For example, you can categorize response times in buckets (range of values) of 200 milliseconds with an upper limit of 1000 milliseconds. Prometheus collects the values and categorizes the values in each of the buckets.

Summary
Similar to a histogram, a summary samples values. However a summary also calculates configurable values over a sliding time window.

Histograms buckets are categorized on the Prometheus server, while summaries are calculated on the client side (that is, the service exposing the metrics).

Note
For a more detailed explanation about the histogram and summary metric type, refer to https://blog.pvincent.io/2017/12/prometheus-blog-series-part-2-metric-types/.

Creating Custom Metrics for Quarkus Applications

You can enable custom metrics for your Quarkus application by adding the **`quarkus-smallrye-metrics`** extension.

The following are the steps to enable custom metrics for Quarkus applications:

1. Include the **`quarkus-smallrye-metrics`** dependency in your application Maven **`pom.xml`** file for enabling metrics collection.

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-smallrye-metrics</artifactId>
</dependency>
```

2. Import the required metrics classes in your application from the **`org.eclipse.microprofile.metrics.*`** package. This package has all the standard Prometheus metric types that you can use to instrument your custom metrics.

3. Add relevant metrics for your application.

   Counters can be created by adding the **`@Counted`** annotation to methods as follows:

```
@Counted(name = "card_transactions", description = "count of credit card
 transactions")
public void processCreditCardTransaction() {
...output omitted...
```

Gauges can be created by adding the **`@Gauge`** annotation to methods:

```
@Gauge(name = "concurrent_users", description = "count of active users")
public void listActiveUsers() {
...output omitted...
```

You can add different types of metrics to your application. For the complete list of options, refer to the Microprofile Metrics standard API reference at https://github.com/eclipse/microprofile-metrics/blob/master/spec/src/main/asciidoc/app-programming-model.adoc.

4. By default, Quarkus makes the metrics available at the **`/metrics`** endpoint. This endpoint combines your custom metrics along with other standard metrics for the application, such as garbage collection information, heap size, and other system level metrics

   Quarkus makes only the custom metrics available at the **`/metrics/application`** endpoint. To see only the system level metrics, access the **`/metrics/base`** endpoint.

Creating Custom Metrics for Node.js Applications

You can add custom metrics to your Node.js application by importing the **`prom-client`** package. The steps to add custom metrics to your Node.js applications are:

1. Install the **`prom-client`** package.

```
[user@demo product]$ npm install --save prom-client
```

2. Import the **`prometheus-client`** package and initialize it. Add a unique prefix to easily identify the standard Node.js runtime metrics for this application. This is to help narrow down the search in the Prometheus web console.

```
var prometheus = require('prom-client');
const prefix = 'myapp_';
prometheus.collectDefaultMetrics({ prefix });
```

Note

You can comment out the **`prometheus.collectDefaultMetrics()`** method if you do not want Node.js runtime system metrics, such as memory usage, garbage collection, CPU usage, and more.

3. Add code to create new Prometheus metric types for your application. Use a suitable prefix (usually application name), and description of the custom metric to easily identify this custom metric in the Prometheus web console.

   Create a new counter as follows:

```
const transactions = new prometheus.Counter({
    name: 'myapp:card_transactions',
    help: 'count of credit card transactions'
});
```

You can then increment the counter (for example at the start of a function):

```
transactions.inc();
```

Similarly, to add a gauge:

```
const concurrent_users = new prometheus.Gauge({
      name: 'myapp:concurrent_users',
      help: 'No of concurrent users'
});
```

Set the value of the gauge:

```
concurrent_users.set(some_value);
```

4. Add code to create a new HTTP GET endpoint called **`/metrics`** which will be used by Prometheus to collect metrics for this microservice. This endpoint will then return the metrics by calling the **`register`** object in the Prometheus client library as follows:

```
app.get('/metrics', function (req, res) {
    res.set('Content-Type', prometheus.register.contentType);
    res.send(prometheus.register.metrics());
})
```

Prometheus client libraries exist for other programming languages. Refer to the Prometheus documentation for more details.

Enabling Prometheus Metrics Scraping

A default installation of Red Hat OpenShift Service Mesh does not collect metrics from applications unless it is explicitly enabled in the application deployment resource file.

To enable Prometheus to scrape (collect) metrics from your application, add the following annotations to the **`spec.template.metadata.annotations`** section in the deployment YAML resource file:

```
...output omitted...
labels:
  app: myapp
annotations:
  sidecar.istio.io/inject: "true"
  prometheus.io/scrape: "true"
  prometheus.io/port: "8080"
  prometheus.io/scheme: "http"
...output omitted...
```

Make sure that your **`prometheus.io/port`** value matches the port number where the **`/metrics`** endpoint is running.

Querying Service Mesh Metrics using Prometheus

To view metrics collected from the service mesh using the Prometheus web console, do the following:

1. In the OpenShift web console, navigate to **Networking** → **Routes** and search for the **`prometheus`** route, which is the URL listed in the **`Location`** column.

2. Log in using the same user name and password that you used to access the OpenShift web console. You should see the Prometheus web console home page.

3. In the expression editor, type **`istio`** and observe the list of metrics that are available (the UI provides auto-completion and prompts you for a list of available metrics). You can query custom metrics by typing the metric name that you declared in your application source code when instrumenting the service for metrics. Once you have selected a metric, click **`Execute`** to display the metrics collected.

4. Select **`insert metric at Cursor`**, and observe the full list of available metrics available for querying.

   Istio service metrics can be queried using the **`istio_*`** entries.

   Envoy proxy metrics can be queried using the **`envoy_*`** entries.

   Control plane metrics can be queried using the **`pilot_*`**, **`citadel_*`**, and **`galley_*`** entries.

Visualizing Service Mesh Metrics using Grafana

To view the metrics dashboard for a service mesh using the Grafana web console, do the following:

1. In the OpenShift web console, navigate to **Networking** → **Routes** and search for the **`grafana`** route, which is the URL listed in the **`Location`** column.

2. Log in using the same user name and password that you used to access the OpenShift web console. You should see the Grafana web console home page.

3. Select **`Home`** in the top left corner and then expand the **`istio`** folder to see a list of available dashboards.

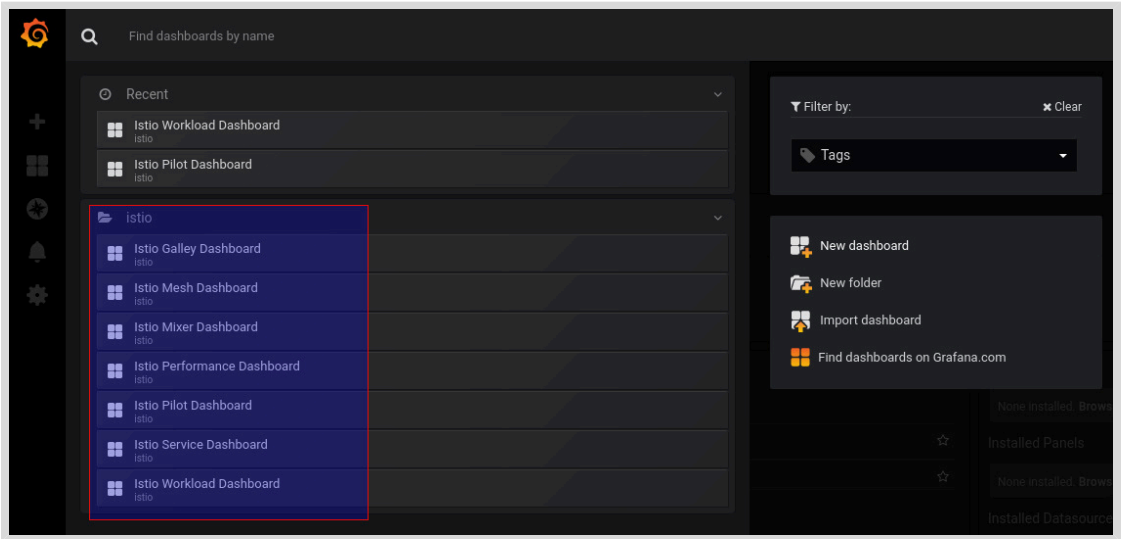


Figure 3.12: List of Grafana Dashboards

4. Click **`Istio Service Dashboard`** to view details about your application services.

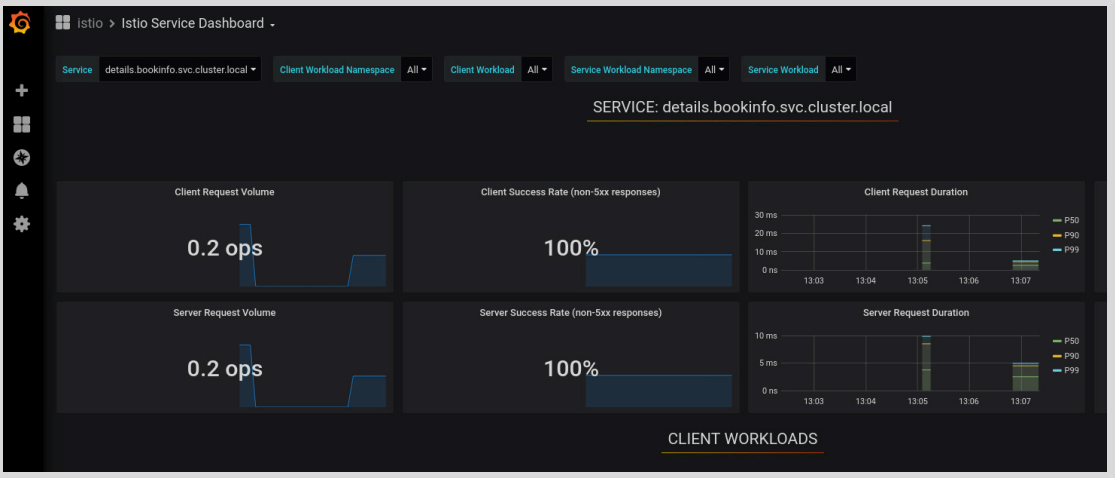


Figure 3.13: Application Service Dashboard

5. Click **`Istio Mesh Dashboard`** to view the state of the overall service mesh.

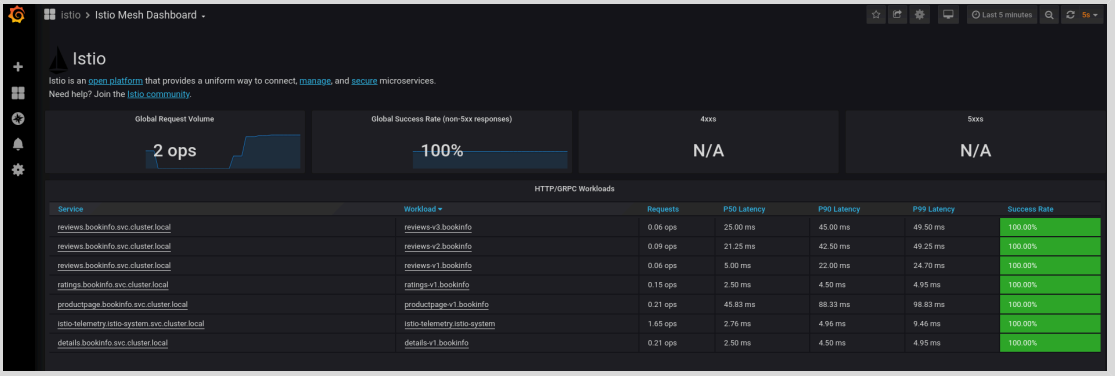


Figure 3.14: Overall Service Mesh Dashboard

Creating Custom Grafana Dashboards

You can create your own custom Grafana dashboards and visualize custom metrics from your application. To create a custom dashboard, do the following:

1. Click the plus (+) icon in the left navigation panel in Grafana, and click **Dashboard** to create a new dashboard.

2. A dashboard consists of one or more panels. A panel can contain one or more metrics of different graph types (Line Graph, Metered Gauge, Bar Graph, Table, Heatmap, and more). Click **Add Query**.

3. You can search for metrics in the query expression editor (Grafana will provide you with auto-completion options) and select an option.

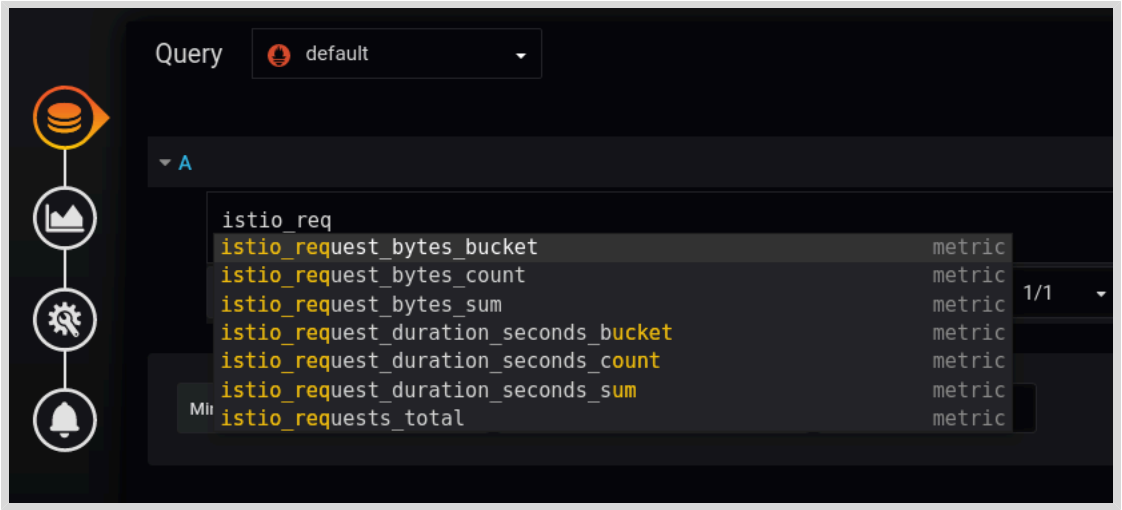


Figure 3.15: Grafana add query

You can click **Add Query** to add more metrics to the panel.

4. Click the graph icon in the left navigation panel to select different types of visual representation for the metrics. Click **Visualization** to select a graph type.

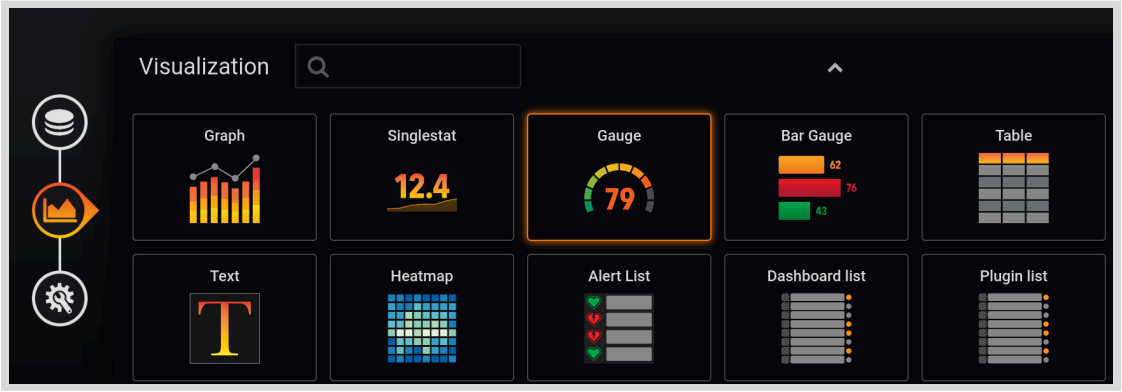


Figure 3.16: Grafana select graph type

Click the gear icon in the left navigation panel to open the **General** page, and provide a suitable name for the panel in the **Title** field.

5. Click the left arrow icon in the top left corner (next to **New dashboard**) to go back to the dashboard page. You should see the panels you added to the dashboard with the selected graph types.

6. Click **Save dashboard** (floppy disk icon) in the top navigation panel to save the dashboard. Provide a suitable name for your dashboard and click **Save**.

References

Service Level Objectives
https://landing.google.com/sre/sre-book/chapters/service-level-objectives/

Prometheus
https://prometheus.io

Grafana
https://grafana.com/

Service Mesh Metrics for Envoy proxy, services and Control Plane
https://archive.istio.io/v1.4/docs/concepts/observability/#metrics

Querying Service Mesh Metrics using Prometheus
https://maistra.io/docs/monitoring_and_tracing/prometheus/

Visualizing Service Mesh Metrics using Grafana
https://maistra.io/docs/monitoring_and_tracing/grafana/

Prometheus client library for Node.js
https://www.npmjs.com/package/prom-client

Prometheus client library for Node.js
https://www.npmjs.com/package/prom-client

Using OpenTracing in Quarkus
https://quarkus.io/guides/opentracing

Types of Prometheus metrics
https://tomgregory.com/the-four-types-of-prometheus-metrics/

Guided Exercise

Collecting Service Metrics

In this exercise, you will deploy two microservices on OpenShift Service Mesh, and collect and inspect critical metrics with Prometheus and Grafana.

You will deploy two microservices in this exercise for a fictional online shopping store:

- **`product`**: written in JavaScript using the Node.js runtime. This service renders product details available for sale in the store.
- **`order`**: written in Java using the Quarkus framework. This service handles orders placed by customers.

You will do the following in this exercise:

1. Enable standard and custom metrics collection in both microservices. Metrics data is sent to the Prometheus instance running in the service mesh.
2. Build container images locally for both microservices using **`podman`**.
3. Push the built container images to the **`Quay.io`** public container registry.
4. Deploy both microservices to the service mesh, and view metrics data using the Prometheus and Grafana web console.

Outcomes

You should be able to deploy applications to OpenShift Service Mesh, and view custom metrics data using Prometheus and Grafana.

Before You Begin

To perform this exercise, ensure you have access to:

- A configured and running OpenShift cluster.
- An installed and running OpenShift Service Mesh in the OpenShift cluster.
- The OpenShift CLI (**`/usr/local/bin/oc`**).
- An account with the **`Quay.io`** container registry, and **`podman`** installed locally on your workstation.

As the **`student`** user on the **`workstation`** machine, use the **`lab`** command to validate the prerequisites for this exercise:

```
[student@workstation ~]$ lab observe-metrics start
```

1. If you have not cloned the source code from the **`DO328-apps`** GitHub repository in a previous exercise, do so now using the **`git clone`** command. Inspect the source code for both microservices in the **`metrics-ge`** folder.

Use a text editor like **VSCodium** that supports syntax highlighting for editing JavaScript and Java source files.

1.1. Open a new terminal window on your workstation. From the home directory, clone the source code for the microservices from GitHub.

```
[student@workstation ~]$ git clone https://github.com/RedHatTraining/DO328-apps
...output omitted...
Cloning into 'DO328-apps'...
...output omitted...
```

1.2. Copy the contents of the **/home/student/DO328-apps/metrics-ge** folder from your local Git repository to the **/home/student/DO328/labs/observe-metrics** folder.

```
[student@workstation ~]$ cp -Rv ~/DO328-apps/metrics-ge/* \
> ~/DO328/labs/observe-metrics/
```

You should now see two folders called **order** and **product** in the **/home/student/DO328/labs/observe-metrics/** folder.

You should also see shell scripts and service mesh related YAML files in the same folder. These were created by the lab start script.

1.3. Review the source code for the **product** microservice in the **/home/student/DO328/labs/observe-metrics/product/server.js** file.

This microservice exposes an HTTP GET endpoint called **/spl50** that renders a 50% discount offer page to customers. For the sake of simplicity, this just renders a simple string that enables us to test invoking the endpoint.

1.4. Review the source code for the **order** in the **/home/student/DO328/labs/observe-metrics/order/src/main/java/com/redhat/training/order/OrderService.java** file.

This microservice has two HTTP GET endpoints. The **/order** endpoint handles orders that are placed after a customer views the special offer page (rendered by the **product** microservice). A randomly generated order id is sent to the client as response.

The **/rating** endpoint simulates customers submitting feedback about the order process. For the sake of simplicity, the rating is generated randomly (a number between 1 and 5).

Note

Code for communication between the **product** and **order** service is not implemented to keep the code simple. The focus for this exercise is to learn how to generate custom metrics for the microservices.

2. Add code to the **product** microservice to enable the collection of custom metrics.

You have been asked to collect the following metrics for this microservice:

1. Response time to render the response for the **/spl50** endpoint. Consistently low response times for a large majority of customers is very important for the store. The operations team wants to track the average response time over a set period of time.

2. Page view count for the **`/spl50`** endpoint. The sales team would like to see data about the number of people who viewed the offer page, and then went on to place the order.

2.1. Navigate to the **`/home/student/DO328/labs/observe-metrics/product`** folder on the command line terminal. All references to file paths in this step are relative to this folder.

```
[student@workstation ~]$ cd ~/DO328/labs/observe-metrics/product
```

2.2. Inspect the **`package.json`** file which declares all the NPM packages required for running this microservice.

Install the NPM **`prom-client`** dependency which provides the Prometheus client library.

```
[student@workstation product]$ npm install --save \
> prom-client@12.0.0
...output omitted...
+ prom-client@12.0.0
...output omitted...
```

2.3. Edit the **`server.js`** file. The complete source code changes for this file can be copied from the **`/home/student/DO328/solutions/observe-metrics/product/server.js`** file.

Import the **`prometheus-client`** package and initialize it. Add a unique prefix called **`product_svc`** to easily identify the standard Node.js runtime metrics for this application in the Prometheus web console.

```
...output omitted...
const express = require('express');
var prometheus = require('prom-client');
const prefix = 'product_svc_';
prometheus.collectDefaultMetrics({ prefix });
```

2.4. Add code to create a new Prometheus gauge type for tracking response time. Note the use of the service name as a prefix to easily identify this custom metric in the Prometheus web console.

```
...output omitted...
app.listen(8080, function () {
    console.log('product-svc started on port 8080');
})

const responseTime = new prometheus.Gauge({
    name: 'product_svc:spl50_response_time',
    help: 'Time take in seconds to render the 50% special offer page'
});
...output omitted...
```

2.5. Add code to create a new Prometheus counter type for tracking the page view count.

```
...output omitted...
  app.listen(8080, function () {
      console.log('product-svc started on port 8080');
  })

  const responseTime = new prometheus.Gauge({
      name: 'product_svc:spl50_response_time',
      help: 'Time take in seconds to render the 50% special offer page'
  });

  const page_views = new prometheus.Counter({
    name: 'product_svc:spl50_page_view_count',
    help: 'No of page views for the 50% special offer page'
  });
  ...output omitted...
```

2.6. Edit the code for the **`/spl50`** route handler.

Start the timer which measures the response time as soon as the function begins to execute. Increment the page view counter.

```
...output omitted...
app.get('/spl50', async function (req, res) {
    responseTime.setToCurrentTime();
    const end = responseTime.startTimer();
    page_views.inc();
...output omitted...
```

2.7. Before sending the response to the client, end the timer which you started at the beginning of the function (after the **`sleep()`** call).

```
...output omitted...
await sleep(Math.floor(Math.random() * 200) + 1);
end();
res.send(view_msg);
...output omitted...
```

2.8. Add code to create a new **`/metrics`** endpoint which will be used by Prometheus to collect metrics for this microservice.

```
...output omitted...
app.get('/metrics', function (req, res) {
    res.set('Content-Type', prometheus.register.contentType);
    res.send(prometheus.register.metrics());
})
```

2.9. Save your changes. To ensure that there are no syntax errors, run the code using the **`Node.js`** runtime, and ensure that the microservice starts without any errors.

If there are errors, then compare your changes with the solution files in the **`/home/student/DO328/solutions/observe-metrics/product`** folder.

```
[student@workstation product]$ npm install
...output omitted...
[student@workstation product]$ node server.js
product-svc started on port 8080
```

Press **Ctrl+C** to stop the server.

3. Add code to the **order** microservice to enable the collection of custom metrics.

You have been asked to collect the following metrics for this microservice:

1. Response time to process an order.
2. Count of orders placed after viewing the special offer page. The sales team would like to see data about the number of people who placed an order.
3. The rate at which orders can be processed. The operations team would like to understand the average rate of orders that can be processed for certain time intervals. This data will be used for planning the scalability and capacity estimation for the system.
4. Satisfaction rating. The sales team would like to collect the satisfaction rating data provided by customers after they place an order.

3.1. Change to the **/home/student/DO328/labs/observe-metrics/order** folder. All references to file paths in this step are relative to this folder.

```
[student@workstation product]$ cd ~/DO328/labs/observe-metrics/order
```

3.2. Inspect the maven **pom.xml** file which declares the dependencies for this microservice.

Include the **quarkus-smallrye-metrics** dependency for enabling metrics collection. Add this dependency below the **quarkus-smallrye-health** dependency.

You can also copy the code from the solution file at **/home/student/DO328/solutions/observe-metrics/order/pom.xml**.

```
...output omitted...
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-smallrye-metrics</artifactId>
</dependency>
...output omitted...
```

3.3. Edit the **src/main/java/com/redhat/training/order/OrderService.java** file. The complete source code changes for this file can be copied from the **/home/student/DO328/solutions/observe-metrics/order/src/main/java/com/redhat/training/order/OrderService.java** file.

Edit the annotations for the **processOrder()** method. Add a counter that tracks the number of orders placed.

```
...output omitted...
@GET
@Path("/order")
@Counted(name = "order_svc:spl50_orders_placed",
  description = "count of spl50 orders placed")
...output omitted...
public String processOrder() {
...output omitted...
```

3.4. Add a timer to track the response time for processing an order.

```
...output omitted...
@Counted(name = "order_svc:spl50_orders_placed", description = "count of spl50
 orders placed")
@SimplyTimed(name = "order_svc:spl50_order_process_time",
           description = "A measure of how long it takes to process an order",
           unit = MetricUnits.MILLISECONDS)
...output omitted...
```

3.5. Add a **`Metered`** metric to track the rate of order processing.

```
...output omitted...
@Metered(name = "order_svc:orders_processed_rate",
         unit = MetricUnits.MINUTES,
         description = "Rate at which orders are placed",
        absolute = true)
...output omitted...
```

The final **`processOrder()`** function should look like the following:

```
...output omitted...
@GET
@Path("/order")
@Counted(name = "order_svc:spl50_orders_placed",
 description = "count of spl50 orders placed")
@SimplyTimed(name = "order_svc:spl50_order_process_time",
 description = "A measure of how long it takes to process an order",
 unit = MetricUnits.MILLISECONDS)
@Metered(name = "order_svc:orders_processed_rate",
   unit = MetricUnits.MINUTES,
   description = "Rate at which orders are placed",
   absolute = true)
@Produces(MediaType.TEXT_PLAIN)
public String processOrder() {
...output omitted...
```

3.6. Add a gauge type metric to the **`generateRandomRating()`** method to capture the rating for the order.

```
...output omitted...
  @Gauge(name = "order_svc:spl50_order_process_rating",
  unit = MetricUnits.NONE,
  description = "Overall customer rating for the order process")
  private Integer generateRandomRating() {
...output omitted...
```

3.7. Save your changes. To ensure that there are no syntax errors, run **`mvn clean package`** and ensure that a fat JAR is created in the **`target`** folder.

```
[student@workstation order]$ mvn clean package
...output omitted...
[INFO] Building order 1.0.0
...output omitted...
...output omitted... Building fat jar: observe-metrics/order/target/order-1.0.0-
runner.jar
[INFO] [io.quarkus.deployment.QuarkusAugmentor] Quarkus augmentation completed in
 2440ms
...output omitted...
[INFO] BUILD SUCCESS
...output omitted...
```

3.8. Start the microservice and verify that there are no errors.

```
[student@workstation order]$ java -jar target/order-1.0.0-runner.jar
...output omitted... (powered by Quarkus 1.3.2.Final) started in 0.747s.
Listening on: http://0.0.0.0:8080
...output omitted...
```

Verify that there are no errors. If there are errors, compare your changes with the solution files in **`/home/student/DO328/solutions/observe-metrics/order`** folder. Press **`Ctrl`**+**`C`** to stop the server.

4. Build container images for both microservices using **`podman`**.

4.1. Load your classroom environment configuration.

Run the following command to load the environment variables:

```
[student@workstation order]$ source /usr/local/etc/ocp4.config
```

4.2. Briefly review the **`Dockerfile`** for the **`product`** microservice. You will use Red Hat Universal Base Images (UBI) as the base for building your container image. Build the container image using **`podman`**.

```
[student@workstation order]$ cd ~/DO328/labs/observe-metrics/product
[student@workstation product]$ podman build -t \
> quay.io/${RHT_OCP4_QUAY_USER}/ossm-metrics-product:1.0 .
STEP 1: FROM registry.access.redhat.com/ubi8/nodejs-12:latest
...output omitted...
STEP 13: COMMIT quay.io/youruser/ossm-metrics-product:1.0
```

4.3. Build the container for the **order** microservice after reviewing the **Dockerfile**.

```
[student@workstation product]$ cd ~/DO328/labs/observe-metrics/order
[student@workstation order]$ podman build -t \
> quay.io/${RHT_OCP4_QUAY_USER}/ossm-metrics-order:1.0 .
STEP 1: registry.access.redhat.com/ubi8:8.1
...output omitted...
STEP 15: COMMIT quay.io/youruser/ossm-metrics-order:1.0
```

4.4. Verify that the container images for **product** and **order** are built successfully.

```
[student@workstation order]$ podman images
REPOSITORY                              TAG ...output omitted...
quay.io/youruser/ossm-metrics-product   1.0 ...output omitted...
quay.io/youruser/ossm-metrics-order     1.0 ...output omitted...
...output omitted...
```

5. Create new public image repositories in Quay.io to store the newly built container images. Push the container images to **Quay.io**.

5.1. Create two new public container image repositories called **ossm-metrics-product**, and **ossm-metrics-order** in Quay.io. Refer to the instructions in *Creating a Quay Account* for creating public container image repositories.

Warning

If you skip this step and push the container images without creating public repositories, then the **podman push** commands will create private container image repositories by default.

5.2. Login to your Quay.io account using **podman**.

```
[student@workstation order]$ podman login -u ${RHT_OCP4_QUAY_USER} quay.io
```

You will be prompted for your Quay.io password.

5.3. Push the container image for the **product** microservice to Quay.io.

```
[student@workstation order]$ podman push \
> quay.io/${RHT_OCP4_QUAY_USER}/ossm-metrics-product:1.0
...output omitted...
Writing manifest to image destination
Storing signatures
```

5.4. Push the container image for the **order** microservice to Quay.io.

```
[student@workstation order]$ podman push \
> quay.io/${RHT_OCP4_QUAY_USER}/ossm-metrics-order:1.0
...output omitted...
Writing manifest to image destination
Storing signatures
```

6. Create the **`metrics`** project and then add it to the **`ServiceMeshMemberRoll`** resource.

6.1. Log in to OpenShift as the **`developer`** user.

```
[student@workstation order]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_API}
Login successful.
...output omitted...
```

6.2. Create the **`metrics`** project.

```
[student@workstation order]$ oc new-project metrics
Now using project "metrics" on server
"https://api.cluster.domain.example.com:6443".
...output omitted...
```

6.3. Add the **`metrics`** project to the list of members in the **`ServiceMeshMemberRoll`** resource.

You can also run the **`add-project-to-smmr.sh`** script in the **`/home/student/DO328/labs/observe-metrics`** folder to add the **`metrics`** project to the list of members in the **`ServiceMeshMemberRoll`** resource.

```
[student@workstation order]$ cd /home/student/DO328/labs/observe-metrics
[student@workstation observe-metrics]$ oc patch servicemeshmemberroll/default \
> -n istio-system --type=merge \
> -p '{"spec": {"members": ["metrics"]}}'
servicemeshmemberroll.maistra.io/default patched
```

7. Deploy the microservices to OpenShift service mesh.

7.1. Edit the **`*-deploy.yaml`** files in the **`/home/student/DO328/labs/observe-metrics`** folder which describes the necessary resources to deploy both applications.

The deployment files have the **`sidecar.istio.io/inject: "true"`** annotation included to inject the Envoy proxy after deployment.

7.2. Edit the **`/home/student/DO328/labs/observe-metrics/product-deploy.yaml`** file. Edit the **`spec.template.spec.containers.image`** attribute and add the Quay.io URL of the container image you created in a previous step.

```
...output omitted...
spec:
  containers:
  - name: product
    image: quay.io/youruser/ossm-metrics-product:1.0
    imagePullPolicy: IfNotPresent
...output omitted...
```

Add annotations to allow the Prometheus instance to collect metrics from this microservice. Add the following annotations below the **`sidecar.istio.io/inject: "true"`** annotation.

```
...output omitted...
labels:
  app: product
  version: v1
annotations:
  sidecar.istio.io/inject: "true"
  prometheus.io/scrape: "true"
  prometheus.io/port: "8080"
  prometheus.io/scheme: "http"
...output omitted...
```

7.3. Edit the **/home/student/DO328/labs/observe-metrics/order-deploy.yaml** file. Add the Quay.io URL of the container image for **order**.

```
...output omitted...
spec:
  containers:
  - name: order
    image: quay.io/youruser/ossm-metrics-order:1.0
    imagePullPolicy: IfNotPresent
...output omitted...
```

Add annotations to allow the Prometheus instance to collect metrics from this microservice. Add the following annotations below the **sidecar.istio.io/inject: "true"** annotation.

```
...output omitted...
labels:
  app: order
  version: v1
annotations:
  sidecar.istio.io/inject: "true"
  prometheus.io/scrape: "true"
  prometheus.io/port: "8080"
  prometheus.io/scheme: "http"
...output omitted...
```

7.4. Run the **oc create** command to deploy the applications.

```
[student@workstation observe-metrics]$ oc create -f product-deploy.yaml
deployment.apps/product created
service/product created
[student@workstation observe-metrics]$ oc create -f order-deploy.yaml
deployment.apps/order created
service/order created
```

7.5. Run the **oc get pods** command and verify that both microservices are deployed and in **Running** state.

```
[student@workstation observe-metrics]$ oc get pods
NAME                       READY   STATUS    RESTARTS   AGE
order-6c89b48d88-b5tqk     2/2     Running   2          1h
product-69dd4f647f-jxwmp   2/2     Running   2          1h
```

7.6. Inspect the **/home/student/DO328/labs/observe-metrics/gateway.yaml** file which describes the ingress gateway for traffic entering the mesh.

From the command line terminal, use the **oc create** command to create the ingress gateway.

```
[student@workstation observe-metrics]$ oc create -f gateway.yaml
gateway.networking.istio.io/observe-metrics-gateway created
```

7.7. Create a **VirtualService** to redirect the ingress traffic to the **product** or **order** service based on the URL.

Examine the **virtual-service.yaml** file which routes the ingress traffic to both services,

Use the **oc create** command to create the virtual service.

```
[student@workstation observe-metrics]$ oc create -f virtual-service.yaml
virtualservice.networking.istio.io/observe-metrics-vs created
```

8. Test the microservices.

8.1. Run the **oc get route** command to get the URL of the Istio gateway.

You can also cut and paste the full command from the **get-ingress-gateway-url.sh** file.

Export the ingress gateway URL to an environment variable called **GATEWAY_URL**.

```
[student@workstation observe-metrics]$ export  \
> GATEWAY_URL=$(oc get route istio-ingressgateway -n istio-system \
> -o template --template '{{ "http://" }}{{ .spec.host }}')
```

8.2. Execute the **curl** command in combination with the **GATEWAY_URL** variable to access the application.

View the 50% special offer page.

```
[student@workstation observe-metrics]$ curl ${GATEWAY_URL}/spl50
50% off on purchase of 100 or more items!
Hurry! Limited stocks...
```

8.3. Place an order using the **order** service.

```
[student@workstation observe-metrics]$ curl ${GATEWAY_URL}/order
Thank you for your order! Your order id is 4789
```

The order id is randomly generated and may be different in your case.

8.4. Invoke the **/rating** endpoint for the **order** service.

```
[student@workstation observe-metrics]$ curl ${GATEWAY_URL}/rating
You rated the order process 3 stars. Thank you for your feedback!
```

The rating score is randomly generated and may be different in your case.

9. Query metrics generated by the microservices using the Prometheus web console.

9.1. Run the **`oc get route`** command to gather the Prometheus web console URL. You can also copy the commands from the **`get-prometheus-url.sh`** file.

```
[student@workstation observe-metrics]$ export  \
> PROM_URL=$(oc get route prometheus -n istio-system \
> -o template --template '{{ "https://" }}{{ .spec.host }}')
```

9.2. USe the Firefox web browser to access the Prometheus web console.

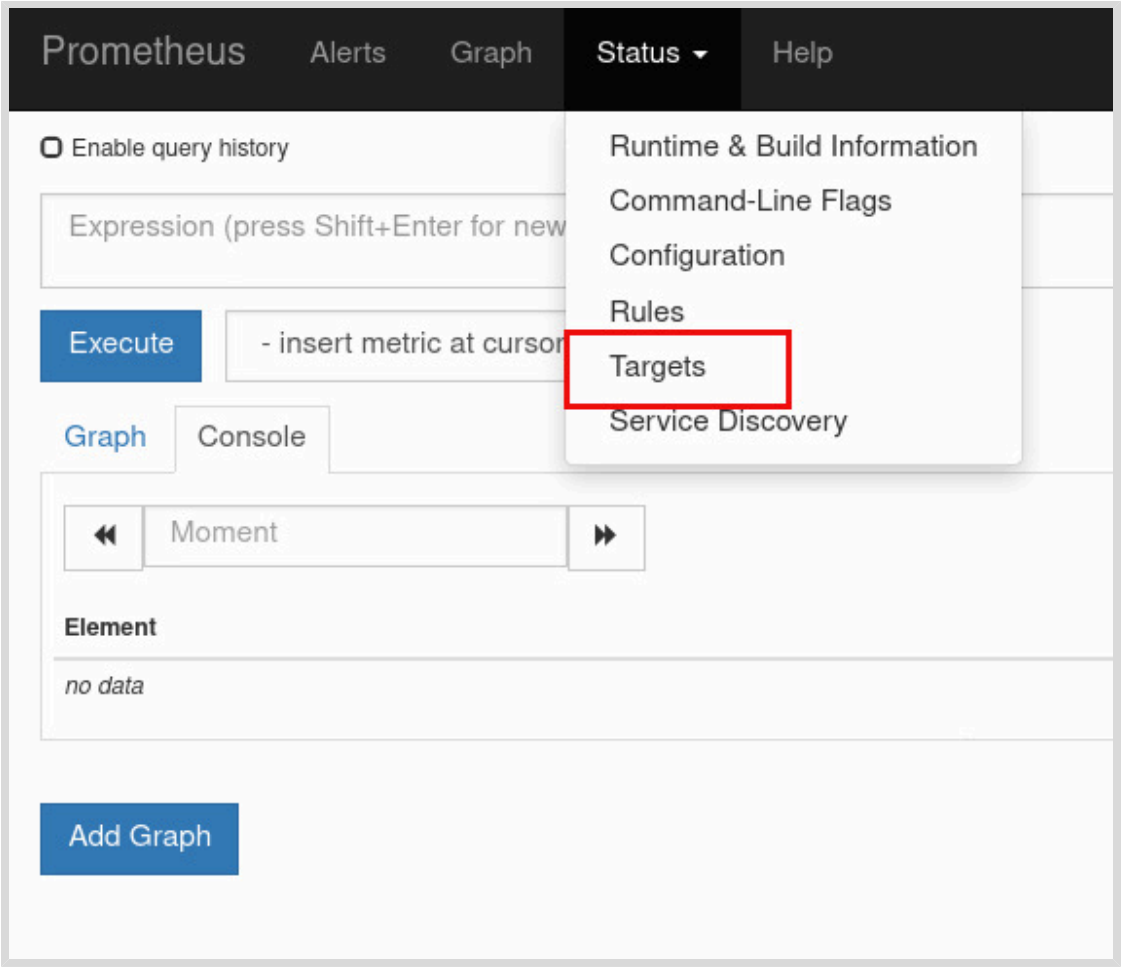
```
[student@workstation observe-metrics]$ firefox ${PROM_URL} &
```

9.3. Click **Log in with OpenShift**.

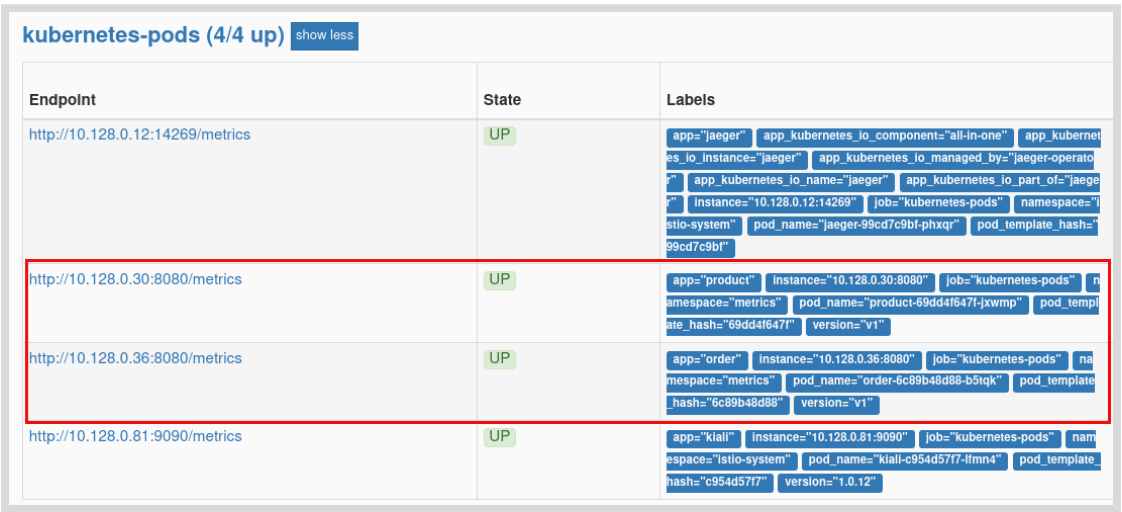
Log in using your developer user account. Your user name is the **`RHT_OCP4_DEV_USER`** variable in the **`/usr/local/etc/ocp4.config`** classroom configuration file. Your password is the **`RHT_OCP4_DEV_PASSWORD`** variable in the same file.

If you are prompted with a page asking you to authorize service account access to your account. Click **Allow selected permissions** to bring up the Prometheus web console.

9.4. Click **Status** → **Targets** to confirm that metrics are being collected from the **`order`** and **`product`** services.



You should see the entry for both services in the **kubernetes-pods** section. You can identify the services by their labels in the **Labels** column.

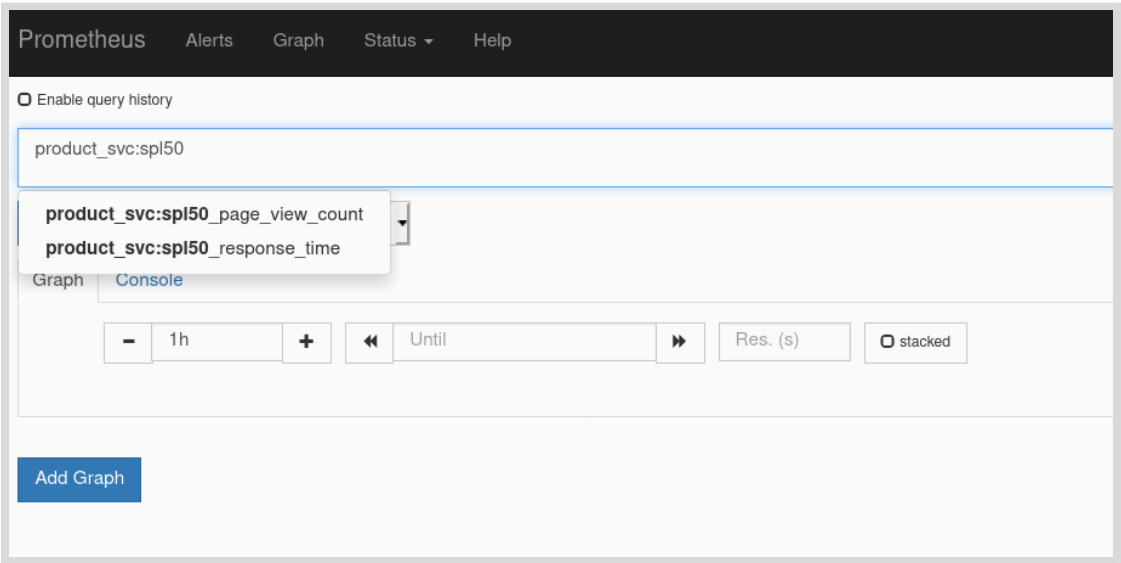


Note

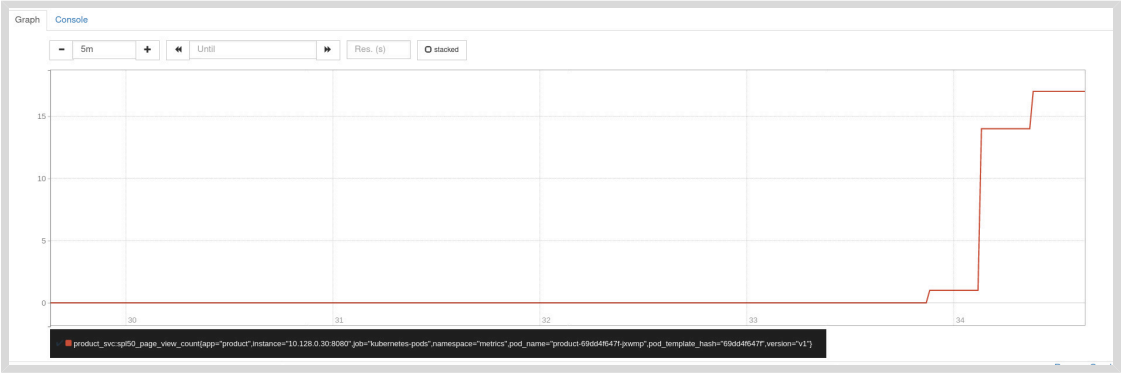
If you cannot see your services in the **Targets** page, then ensure that you added **`prometheus.io/*`** annotations to the deployment YAML files for the services.

9.5. Click **Graph** in the top navigation bar in the Prometheus web console. Click the **Graph** tab. An expression editor is available at the top of the page for queries using the PromQL query language.

9.6. Start by typing **`product_svc:spl50`** to see the metrics gathered for the **`product`** service. Recall that you had provided this string as a prefix for all metrics created for the **`product`** service. The expression editor should provide autocompleted options based on the metrics it has gathered.

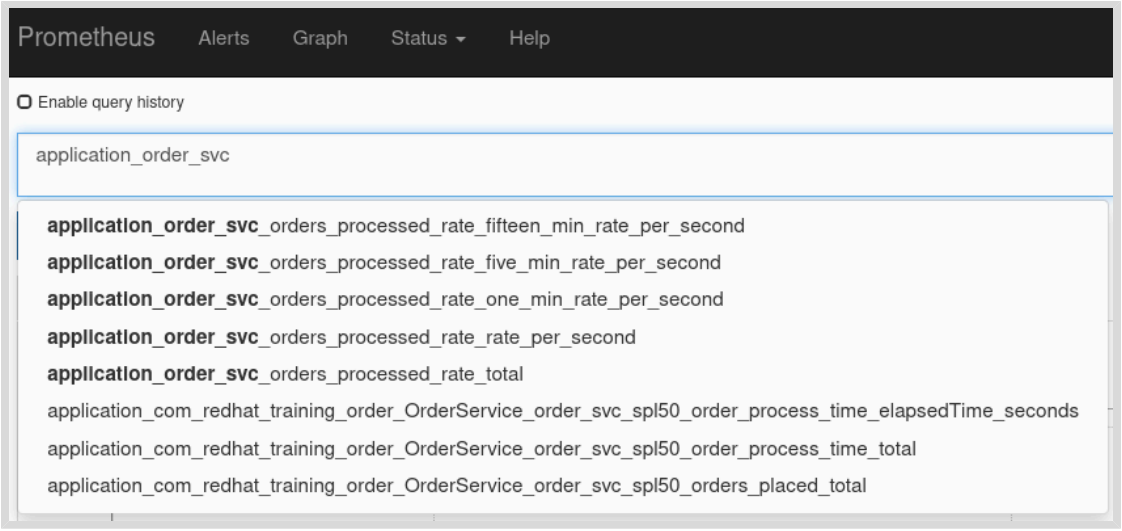


Select **`product_svc:spl50_page_view_count`** in the expression editor, and click **Execute**. You should see a graph for the page view count metric.



Next, select **`product_svc:spl50_response_time`** in the expression editor, and click **Execute**. You should see a graph for the response time metric.

9.7. Start by typing **`application_order_svc`** in the expression editor to view the metrics for the **`order`** service. You should be presented with a list of custom, and other standard metrics that were collected from the service.



Select a metric in the expression editor, and click **Execute** to see a graph for the corresponding metric.

Note

If you do not see any metric data, then use the **`curl`** command from previous steps to invoke the **`/spl50`**, **`/order`**, and **`/rating`** endpoints a few times. You may have to wait for a few seconds until Prometheus scrapes the metrics from the service and displays the data in the graph.

10. Visualize default metrics for the service mesh using the Grafana web console.

10.1. Run the **`oc get route`** command to gather the Grafana web console URL. You can also copy the commands from the **`get-grafana-url.sh`** file.

```
[student@workstation observe-metrics]$ export  \
> GRAFANA_URL=$(oc get route grafana -n istio-system \
> -o template --template '{{ "https://" }}{{ .spec.host }}')
```

10.2. Use the Firefox web browser to access the Grafana web console.

```
[student@workstation observe-metrics]$ firefox ${GRAFANA_URL} &
```

10.3. Click **Log in with OpenShift**.

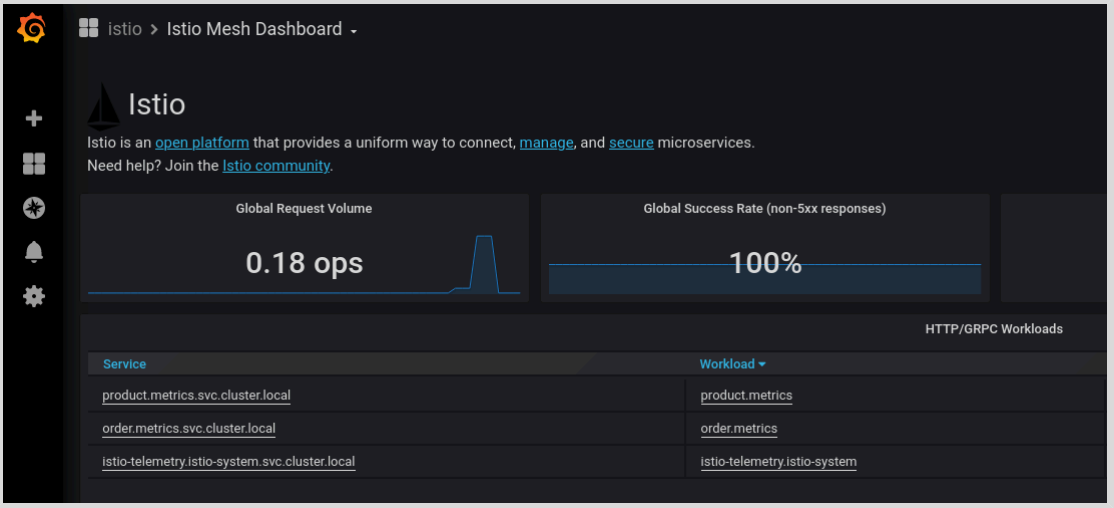
Log in using your developer user account. Your user name is the **`RHT_OCP4_DEV_USER`** variable in the **`/usr/local/etc/ocp4.config`** classroom configuration file. Your password is the **`RHT_OCP4_DEV_PASSWORD`** variable in the same file.

If you are prompted with a page asking you to authorize service account access to your account, then you must click **Allow selected permissions** to bring up the Grafana web console.

10.4. Grafana is already configured to use Prometheus as a data source. The envoy proxy sidecars are configured to automatically send metrics to Prometheus. This data is

used to populate a set of pre-created dashboards related to various aspects of the service mesh.

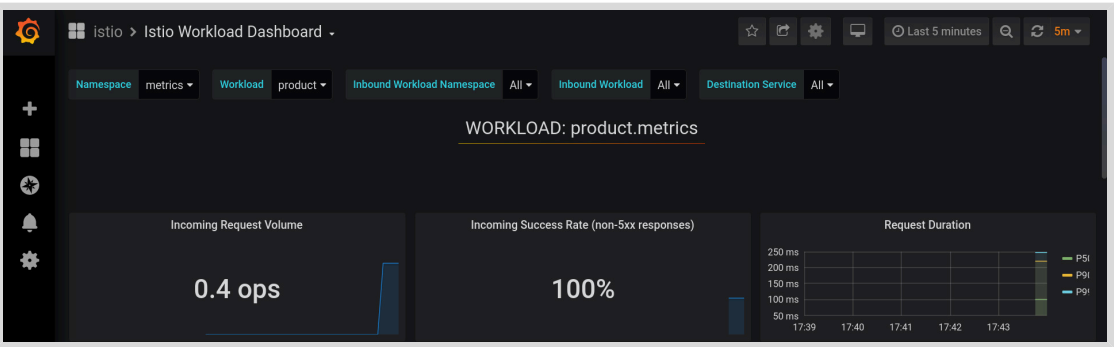
Click **Home** in the top left corner of the Grafana web console. The service mesh dashboards are grouped together under the **`istio`** folder. Expand the **`istio`** folder, and select the **Istio Mesh Dashboard** to view high-level statistics about the overall service mesh.



Note

If you do not see any metric data, then use the **`curl`** command from previous steps to invoke endpoints from the **`product`**, and **`order`** services a few times. You might have to wait for a few seconds until Prometheus scrapes the metrics from the service and displays the data in the graph. Adjust the refresh interval (refresh icon in the top right corner) to a higher value like **`5m`**.

10.5. Click **Istio Workload Dashboard** to view details about the **`product`** and **`order`** services. You are provided with options to filter data by namespace and workload. Select the **metrics** namespace, and then select **product** or **order** to view the statistics for the corresponding service.

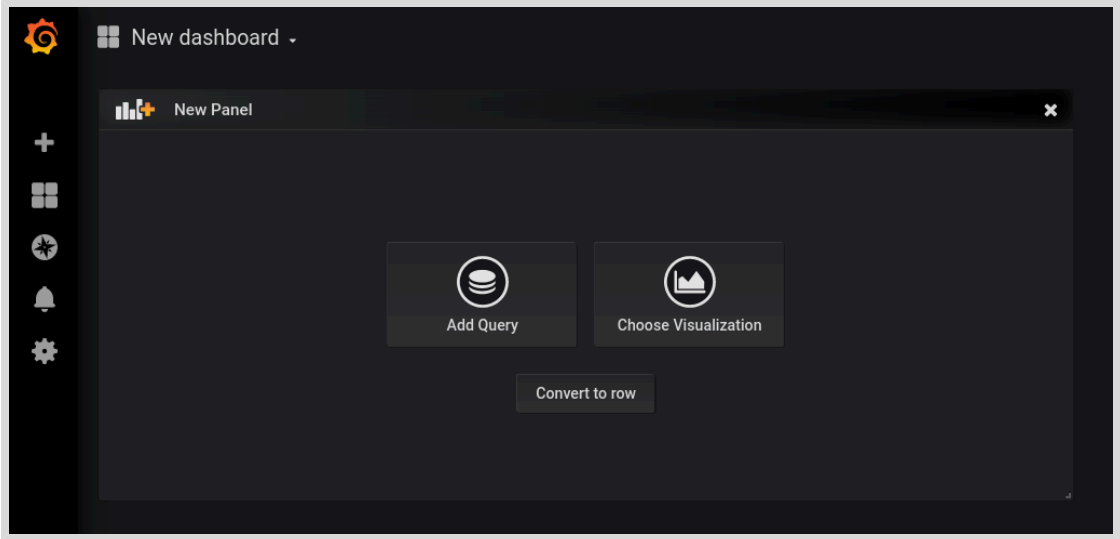


Optional: Briefly select and view other provided dashboards under the **istio** folder.

11. Create a custom dashboard for the shopping store. Populate it with the custom metrics gathered from the **`product`** and **`order`** services.

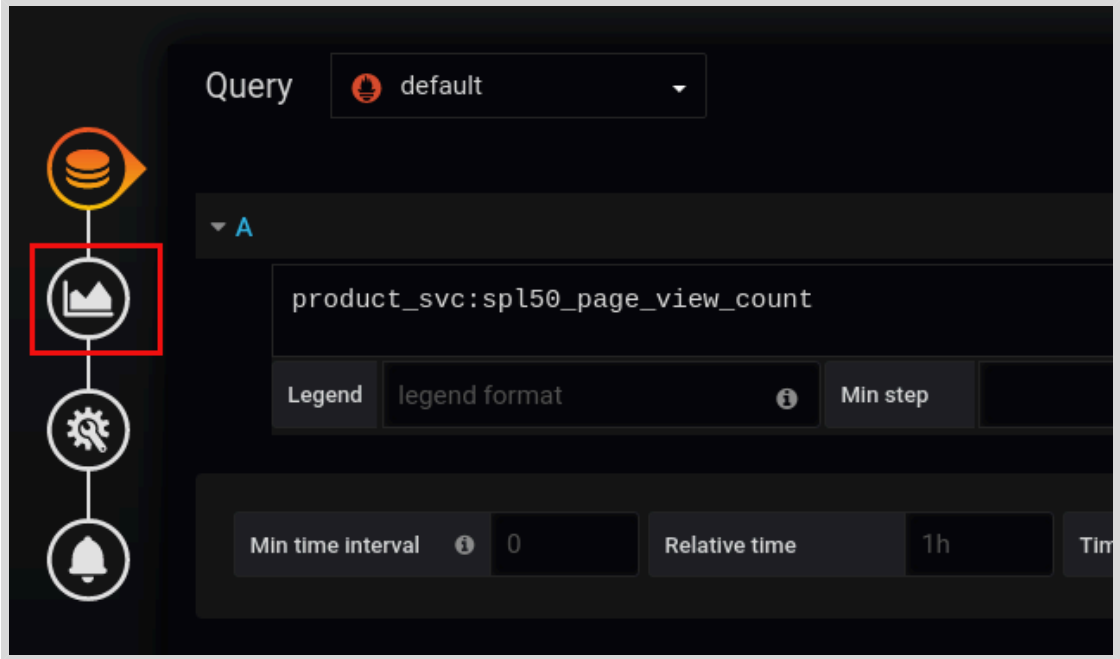
11.1. Click the plus (+) icon in the left navigation menu of Grafana web console to open the **Create** menu. Select **Dashboard** to create a new dashboard.

You will see a panel added to the dashboard called **New Panel**.

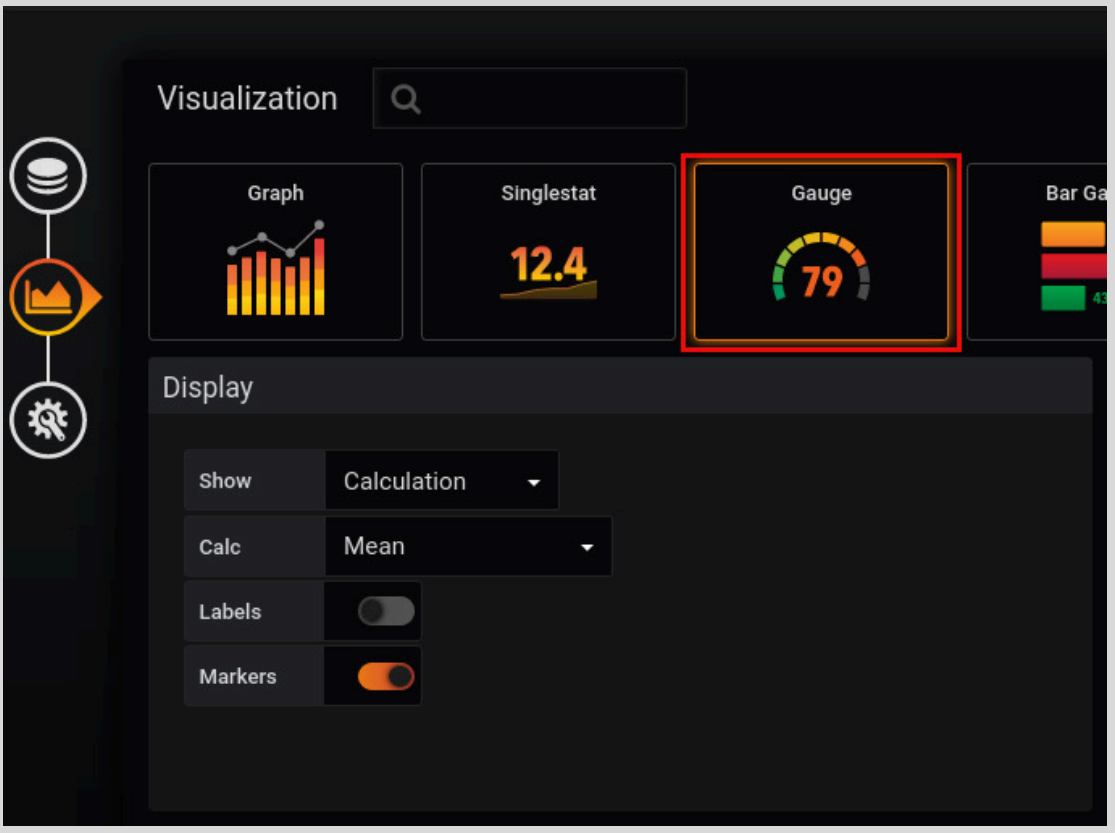


11.2. Click **Add Query** in the **New Panel** panel.

In the query expression editor named **A** (next to the **Metrics** label), type **`product_svc:spl50_page_view_count`**. Click the graph icon in the left menu (to the left of the **Query** window) to select how you want to visualize the metric.



11.3. In the **Visualization** panel, expand the **Visualization** selection, and select **Gauge**.



11.4. Click **General** (gear icon below **Visualize** icon in the left menu).

Enter **`Product Page View Count`** as the **Title**.

Click the left arrow icon (top left corner to the left of **New dashboard**) to go back to the dashboard page. You will add a panel with a metric from the **`order`** service to this dashboard in the next step. Your dashboard should now show the page view count metric from the **`product`** service.

11.5. Click **Add panel** (graph icon with yellow plus in top right) to add another panel for displaying metrics from the **`order`** service.

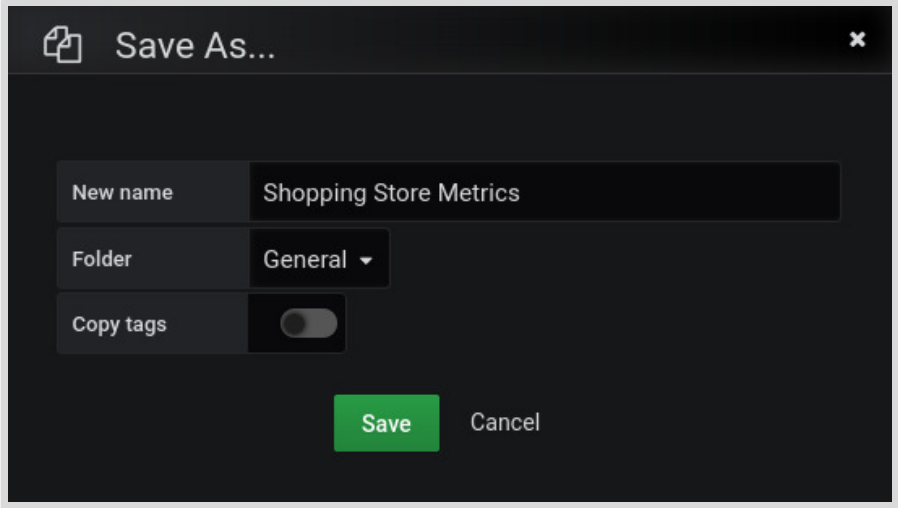
Click **Add Query** in the **New Panel**, and type **`orders_placed`**. Prometheus will auto-complete a long metric name ending with **`orders_placed_total`**. Select the option.

Click **Visualization** in the left menu, and select the **Graph** option.

Click **General** in the left panel, and change the **Title** field to **`Total Orders Placed`**. Click the left arrow icon in the top left corner to go back to the **New dashboard** page.

11.6. Your new dashboard should now show one metric each from both services. You can add more panels and add custom metrics from your services using the same steps outlined previously.

Click the **Save dashboard** icon to save your new dashboard. Enter **`Shopping Store Metrics`** in the **Save As** dialog.



Your final Grafana dashboard should look like the following.

12. Return to the home directory.

```
[student@workstation observe-metrics]$ cd ~
[student@workstation ~]$
```

Finish

On the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab observe-metrics finish
```

This concludes the guided exercise.

Observing Service Interactions with Kiali

Objectives

After completing this section, you should be able to monitor and visualize service interactions with Kiali.

Introducing Kiali

Kiali is a web based console for viewing the topology of a service mesh. In a microservices architecture, a large number of discrete services will be interacting with each other in various complex ways to achieve business goals. Kiali helps you to understand the structure of your service mesh and how traffic flows between services in the mesh. Kiali also provides intuitive dashboards with dynamic animation to understand the end-to-end flow of requests as it traverses the service mesh.

Kiali provides an interactive graphical view of the services in your service mesh in real time. It provides visibility into features like circuit breakers, request rates, latency, and traffic flows. Kiali also provides the ability to validate your service mesh configuration. You can configure gateways, destination rules, virtual services, mesh policies and visually verify the impact of these changes using Kiali.

A default installation of Red Hat OpenShift Service Mesh includes a fully configured ready to use instance of Kiali.

Viewing Service Mesh Interactions with Kiali

The Kiali console has different views that provide insights into service mesh components from various perspectives, such as applications, services, versions, configuration health status, and more. The following are the steps to use Kiali:

1. In the OpenShift web console, navigate to **Networking** → **Routes** and search for the **`kiali`** route, which is the URL listed in the **`Location`** column.

2. Log in using the same user name and password that you used to access the OpenShift web console. You should see the Kiali web console home page, which shows the OpenShift projects that are managed by the service mesh, and the count and overall health status of the services in them.

3. Click a project to see the list of services and their overall health status.

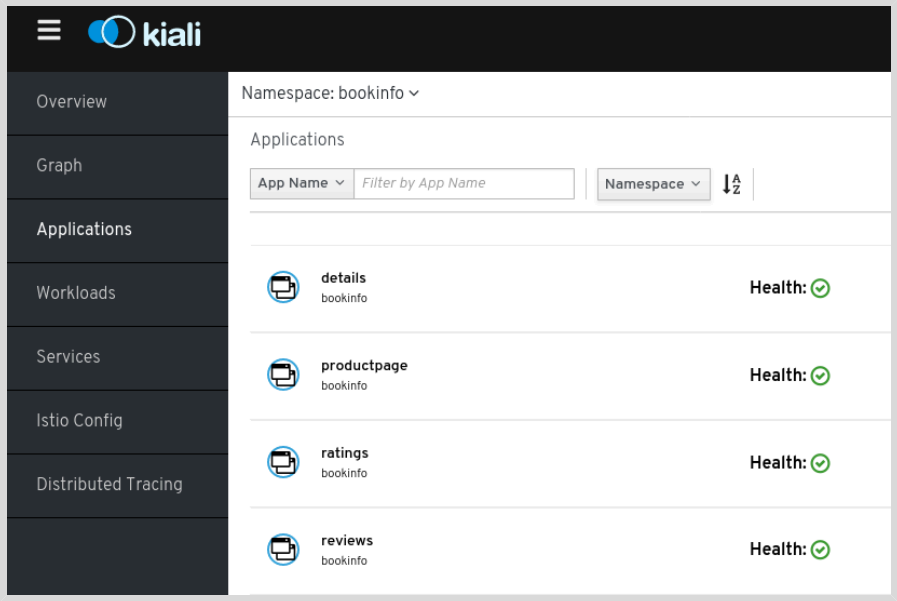


Figure 3.29: Service Health Status in a Project

4. Click **`Graph`** in the left menu to see a dynamic, real-time, animated representation of your service mesh. In the **`Graph`** page, you can view the topology of your service mesh in terms of services, workloads, and versions.

   Select **`App graph`** (below the **`Namespace`**) to view the topology in terms of the services and the application containers to which traffic is being sent. This view aggregates all versions of an application into a single node on the graph. This view does not show the different versions of an application container that are deployed to the mesh.

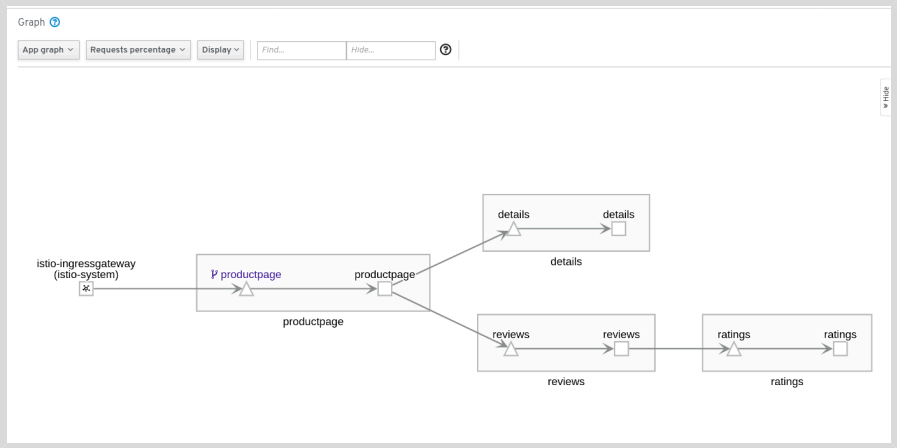


Figure 3.30: App Graph View

5. Select **`Service graph`** to view the topology in terms of only the services in the mesh. This view shows a node for each service in your mesh, but excludes all applications and their versions from the graph.

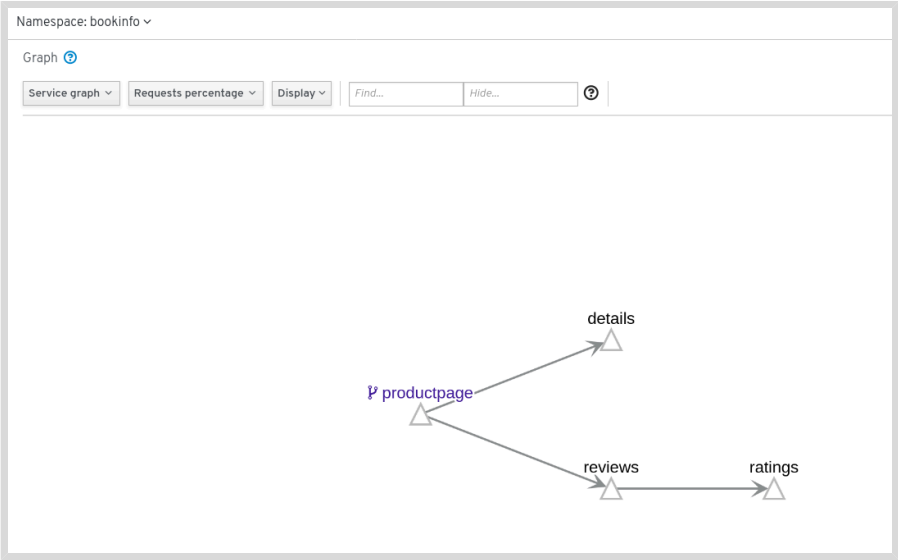


Figure 3.31: Service Graph View

6. Select **`Versioned app graph`** to view the topology in terms of the services and the different versions of application containers to which traffic is being sent. This view shows a node in the graph for each version of an application, but all versions of a particular application are grouped together. Using this view, you can easily identify the amount of traffic that is being sent to specific versions of your application. This is useful to verify dark launches, A/B testing and deployments involving multiple versions of applications.

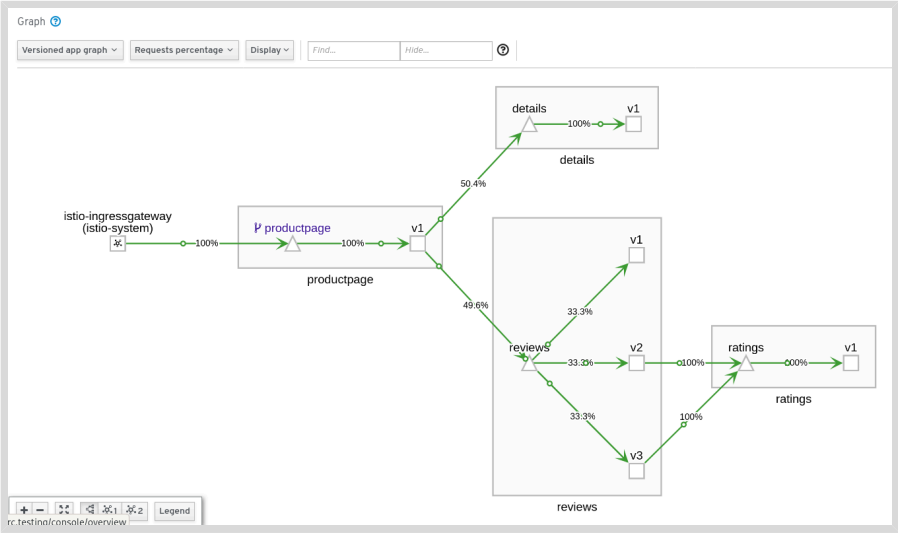


Figure 3.32: Versioned App Graph View

7. Select **`Workload graph`** to view a simplified view of the service mesh topology without any grouping.

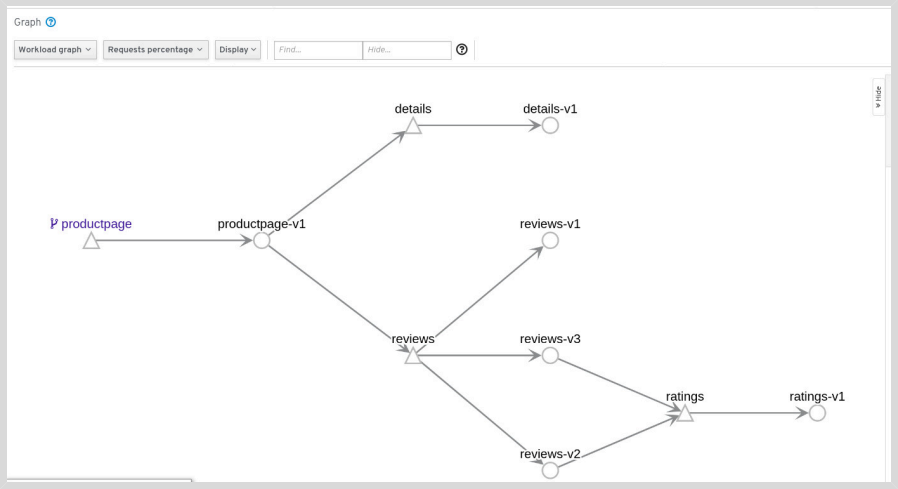


Figure 3.33: Workload Graph View

Note

Click the blue question mark icon next to the **Graph** page title to understand the function of the various drop-down boxes,buttons and options available in the **Graph** page. You can also click the **Legend** button at the bottom of the **Graph** page to get details about the various colored icons in the graphs.

References

For more information about Kiali, refer to the *Kiali* chapter in the *Red Hat OpenShift Service Mesh Guide* at
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.4/html-single/service_mesh/index#ossm-kiali

Kiali
https://kiali.io/

Visualizing your Service Mesh
https://archive.istio.io/v1.4/docs/tasks/observability/kiali/

Guided Exercise

Observing Service Interactions with Kiali

In this exercise, you will deploy an application consisting of four microservices and visualize the service interaction and traffic flow using Kiali.

The application consists of four microservices:

- The first three microservices are **`czech`**, **`english`** and **`spanish`**, which are simple microservices that greet the user in Czech, English and Spanish respectively.
- **`greet-api`**: An API gateway, which acts as the entry point for the application. The API gateway calls the individual language services in different ways depending on the request.

Outcomes

You can visualize traffic flow and inter-service communication using Kiali.

Before You Begin

To perform this exercise, ensure you have:

- A configured and running OpenShift cluster.
- An installed and running OpenShift Service Mesh in the OpenShift cluster.
- The OpenShift CLI (**`/usr/local/bin/oc`**).

On the **`workstation`** machine, use the **`lab`** command to prepare your system for this exercise.

```
[student@workstation ~]$ lab observe-kiali start
```

The **`lab`** command deploys the **`czech`**, **`english`**, **`spanish`**, and **`greet-api`** services into your Red Hat OpenShift cluster. The source code is in the Git repository at https://github.com/RedHatTraining/DO328-apps in the **`kiali-ge`** folder.

You can examine the full deployment file in the **`~/DO328/labs/observe-kiali/app-deployment.yaml`** file. In the **`app-deployment.yaml`** file, note that a gateway and a virtual service is created which exposes the following endpoints:

- **`/greet`**: The API gateway calls each of the individual language services in alphabetical order:

  czech -> english -> spanish.

- **`/chained`**: The API gateway calls only the **`english`** service. The **`english`** service in turn calls another service to form a chain as follows:

  english -> spanish -> czech.

1. Log in to OpenShift and verify that the four microservices are deployed.

1.1. Run the following command to load the environment variables created in the first guided exercise:

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

1.2. Log in to OpenShift:

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_API}
Login successful.
...output omitted...
```

1.3. Set the current project to **observe-kiali**:

```
[student@workstation ~]$ oc project observe-kiali
Now using project "observe-kiali" on server ...output omitted...
```

1.4. Verify that there are four pods in **Running** state:

```
[student@workstation ~]$ oc get pods
AME                          READY   STATUS    RESTARTS   AGE
czech-84c5754796-cpqfq        2/2     Running   0          49s
english-v1-684884c897-qk7j2   2/2     Running   0          49s
greet-api-7fb89fdc45-f7gs6    2/2     Running   0          49s
spanish-f8848fc89-c9slw       2/2     Running   0          49s
```

2. Log in to Kiali and verify that the four microservices are in a healthy state.

2.1. Run the **oc get route** command to gather the Kiali web console URL. You can also copy the commands from the **get-kiali-url.sh** file in the **/home/student/DO328/labs/observe-kiali** folder..

```
[student@workstation ~]$ export  \
> KIALI_URL=$(oc get route kiali -n istio-system \
> -o template --template '{{ "http://" }}{{ .spec.host }}')
```

2.2. Access the Kiali web console using the **firefox** browser on your workstation.

Warning

The lab start script updates the **ServiceMeshMemberRoll** resource of the service mesh control plane. This will cause the Kiali pod to be redeployed after some time. Check the status of the Kiali pod by running **oc get pods -n istio-system**, and proceed after you see it in **Running** state.

If the Kiali pod is restarted after you have logged into Kiali, or if you see errors in the Kiali console, verify the Kiali pod status and log in again.

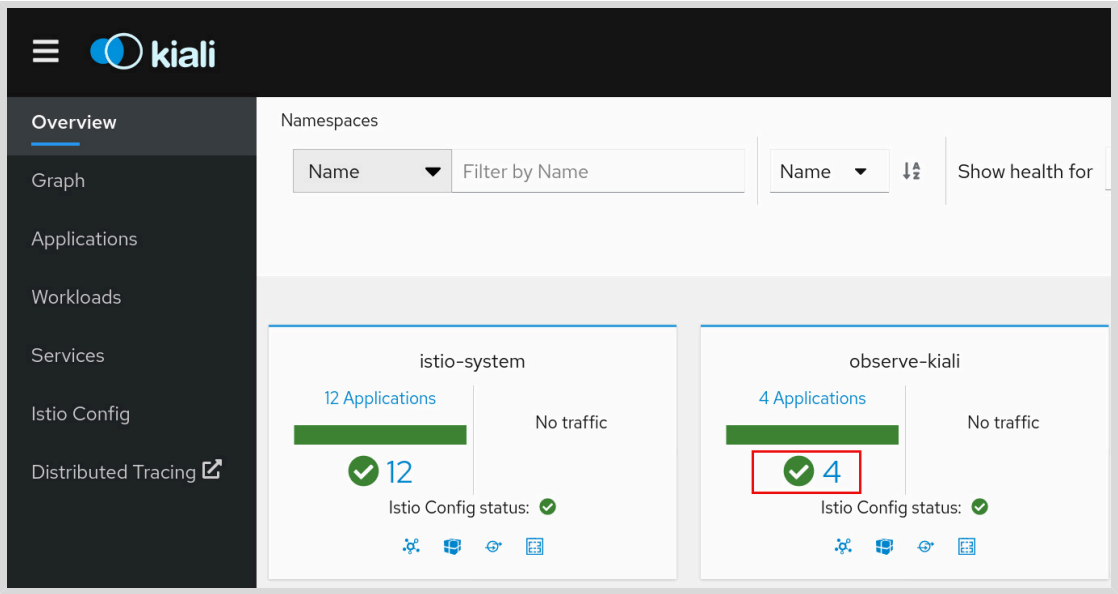
```
[student@workstation observe-metrics]$ firefox ${KIALI_URL} &
```

2.3. Click **Log in with OpenShift**.

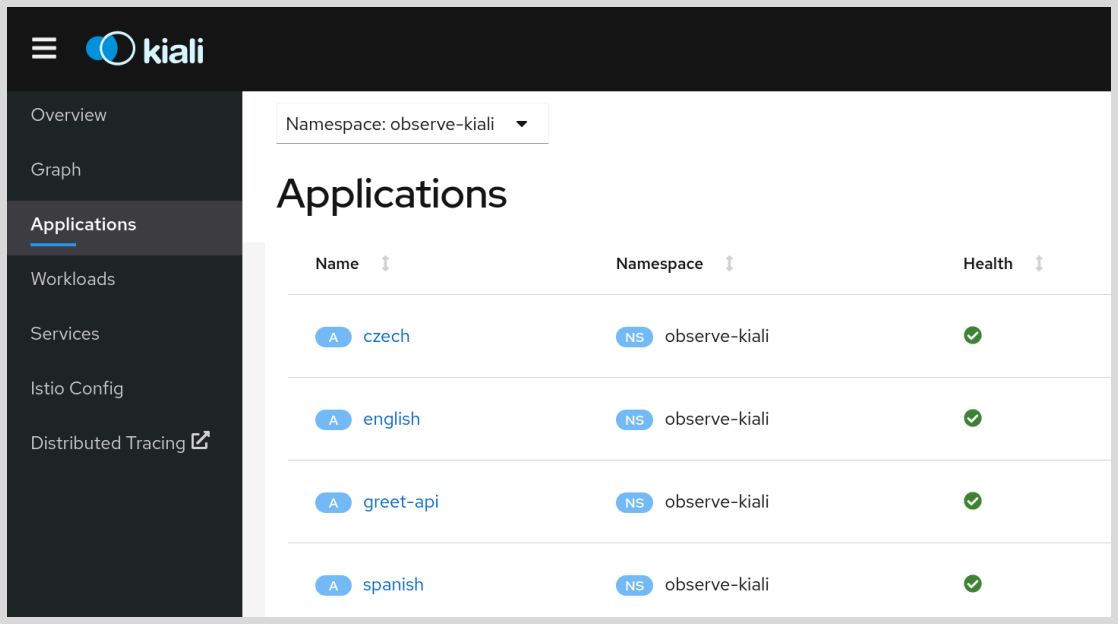
Log in using your developer user account. Your user name is the **`RHT_OCP4_DEV_USER`** variable in the **`/usr/local/etc/ocp4.config`** classroom configuration file. Your password is the **`RHT_OCP4_DEV_PASSWORD`** variable in the same file.

If you are prompted with a page asking you to authorize service account access to your account, then click **Allow selected permissions** to bring up the **Overview** page of the Kiali web console.

2.4. Click the green tick icon in the **observe-kiali** namespace to view the **Applications** page.



Verify that all four microservices are healthy.

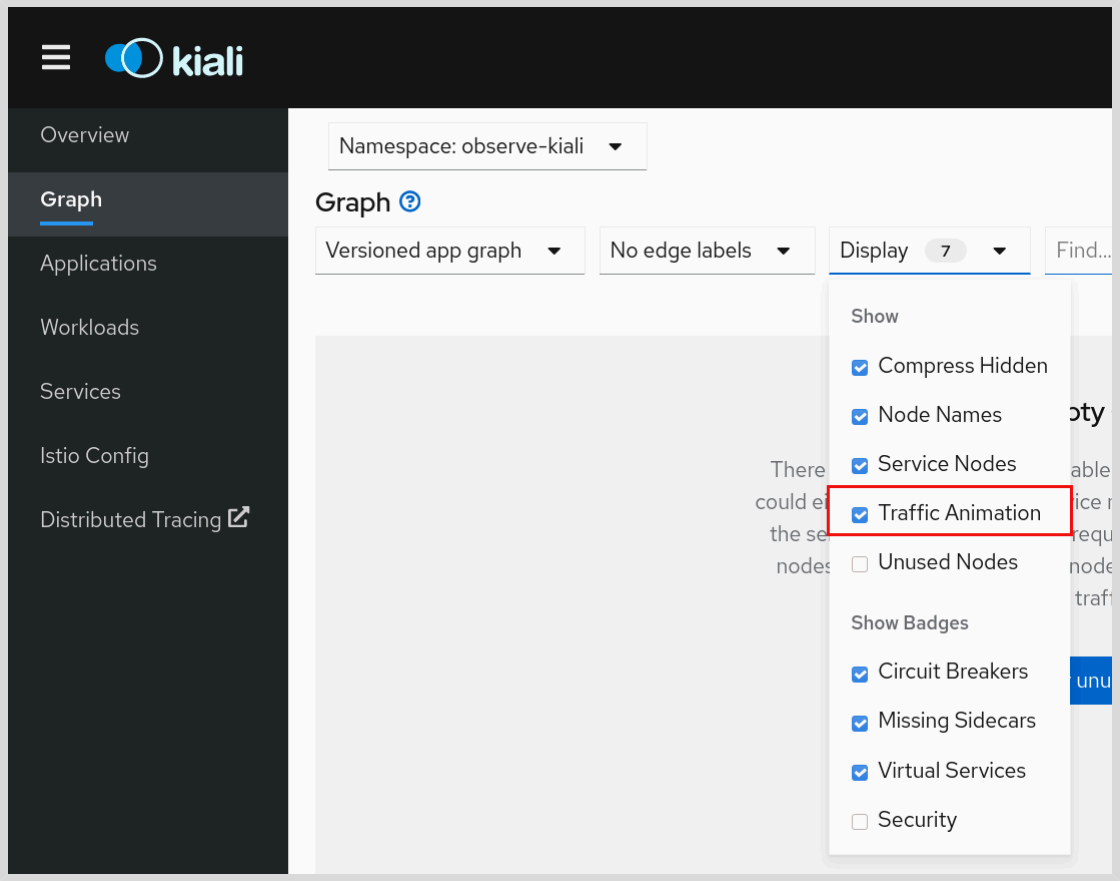


3. Invoke the **`/greet`** end point, and visualize the traffic flow in Kiali.

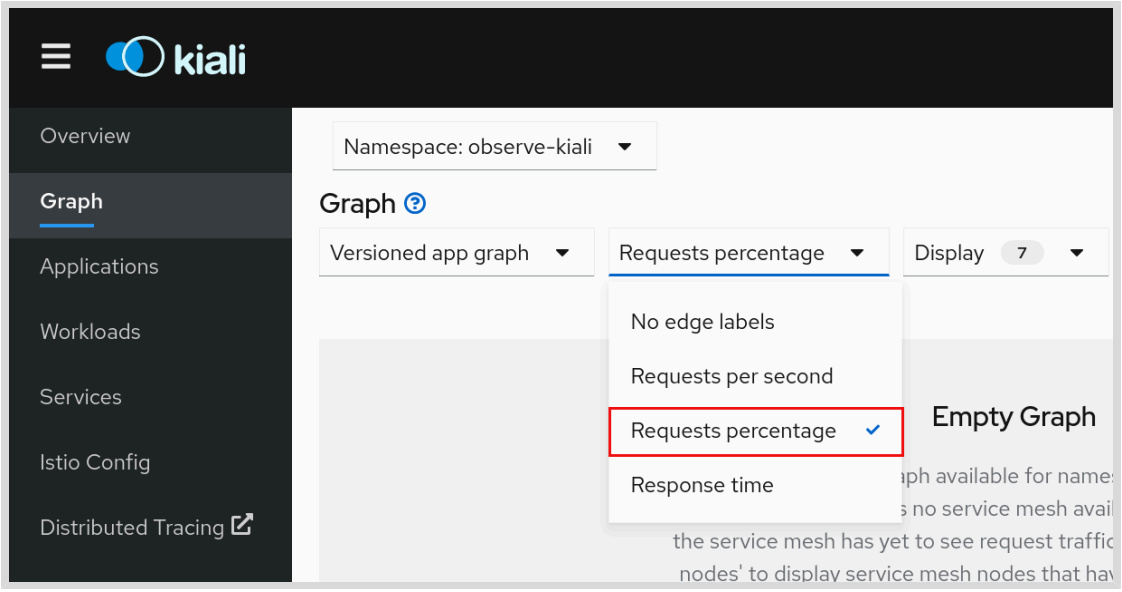
3.1. Set up Kiali for traffic visualization.

Click **Graph** in the left navigation panel. Because there is no traffic being sent to the microservices, the **`Graph`** page will be empty.

Click **Display**, and select the **Traffic Animation** option to enable Kiali to show you an animated version of the traffic flow as requests come in to the service mesh.



Click **No edge labels**, and select the **Requests percentage** option to enable Kiali to show you the percentage of requests sent to different versions of a microservice.



3.2. Run the **`oc get route`** command to get the URL of the istio gateway.

You can also cut and paste the full command from the **`get-ingress-gateway-url.sh`** file in the **`/home/student/DO328/labs/observe-kiali`** folder.

Export the ingress gateway URL to an environment variable called **`GATEWAY_URL`**.

```
[student@workstation ~]$ export  \
> GATEWAY_URL=$(oc get route istio-ingressgateway -n istio-system \
> -o template --template '{{ "http://" }}{{ .spec.host }}')
```

3.3. Use the **`curl`** command to keep sending continuous requests to the **`/greet`** endpoint. This command will not return to the prompt, and will continue to run unless you explicitly stop it with **`Ctrl`**+**`C`**.

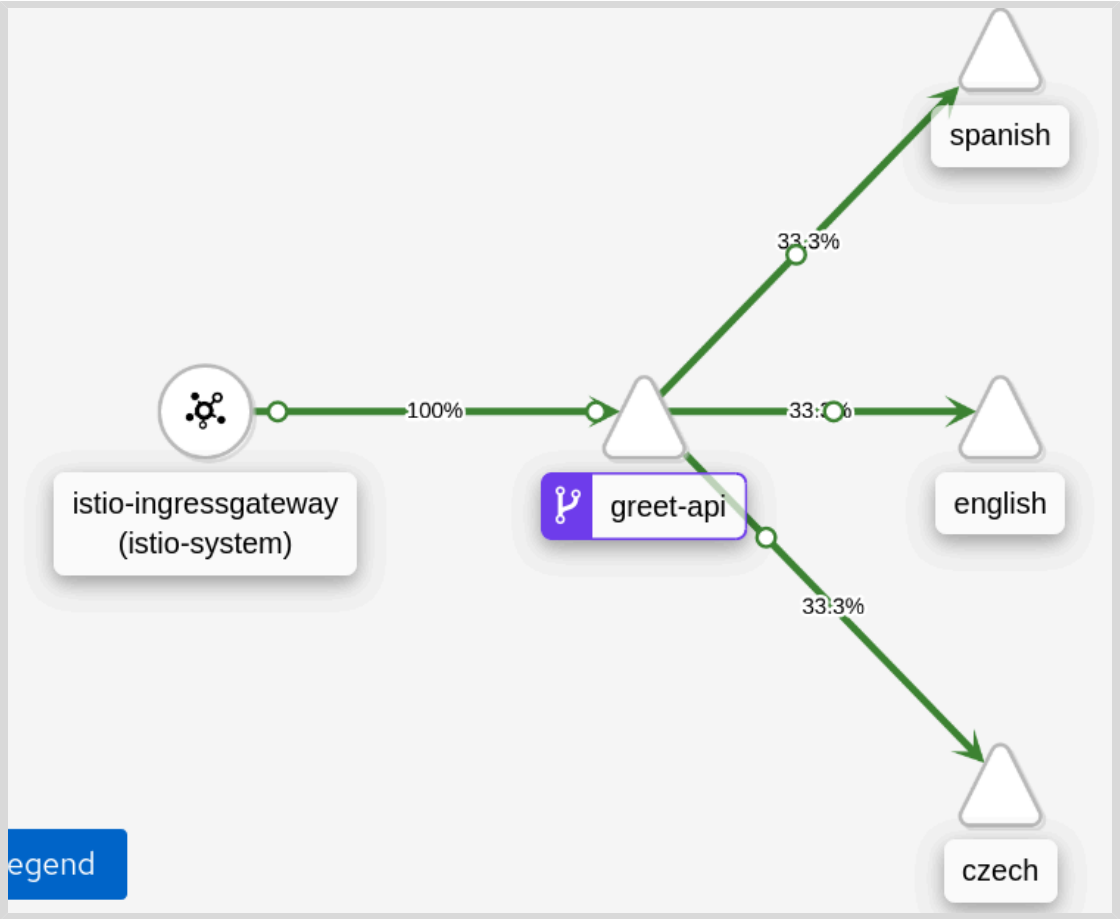
You can also run the **`invoke-greet.sh`** script in the **`/home/student/DO328/labs/observe-kiali`** folder

```
[student@workstation observe-metrics]$ while true;do curl ${GATEWAY_URL}/greet; \
> sleep 3;done
Ahoj světe! | Hello World! | Hola Mundo!
Ahoj světe! | Hello World! | Hola Mundo!
...output omitted...
```

3.4. Switch to the Kiali **Graph** page, and observe the traffic animation. You might have to wait for a few seconds while Kiali captures data from the Envoy proxies and renders the animation.

By default, Kiali displays a graph with the services and their versions (**Versioned app graph**).

Click **Versioned app graph** and select the **Service graph** option to display a more compact graph with only the services in the application.

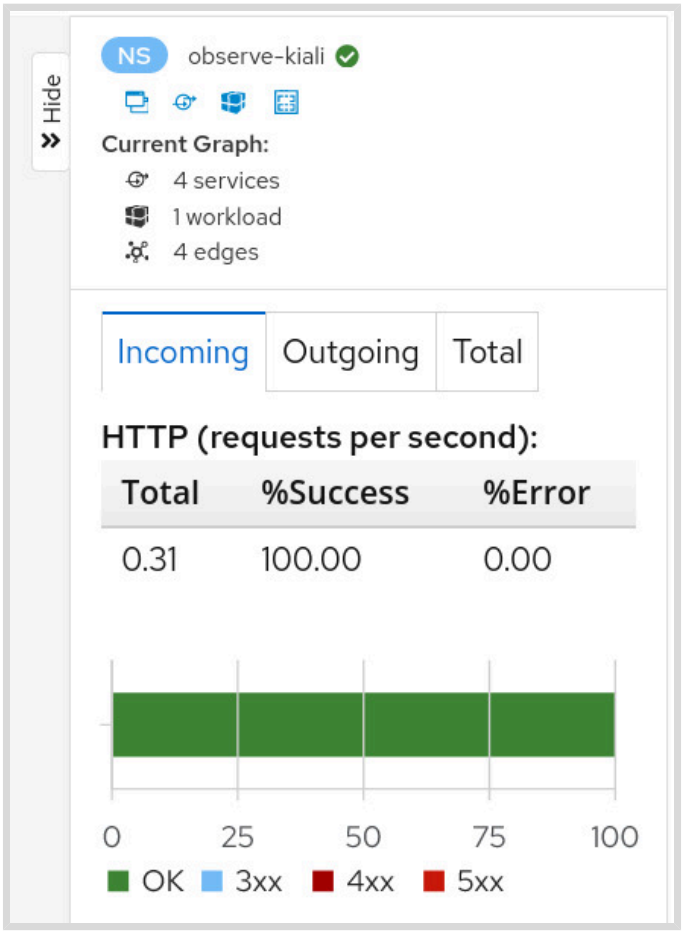


Note how the **`greet-api`** calls the individual services. You should see the percentage of responses being equally split between the three language services at this point.

A side panel to the right of the graph shows more details about the overall service mesh. You can click the services in the graph, and the side panel will show details of

the selected service. Clicking anywhere other than the displayed services switches back to the overall service mesh view.

You can click the **Hide** or **Show** button on the side panel to hide or show the side panel respectively.



3.5. Press **`Ctrl`**+**`C`** to stop the curl command in the command line terminal window where you were invoking the **`/greet`** endpoint.

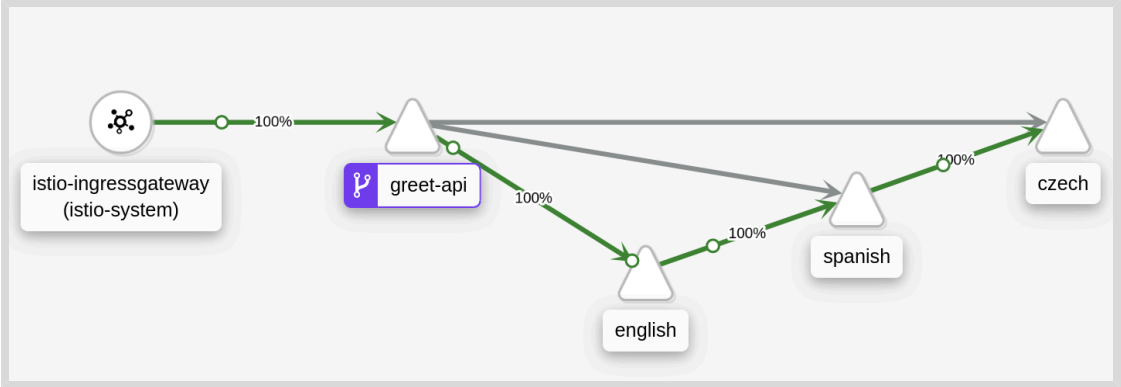
4. Invoke the **`/chained`** end point, and visualize the traffic flow in Kiali.

4.1. Use the **`curl`** command to keep sending continuous requests to the **`/chained`** endpoint. Observe that the language services are now called in a different order.

You can also run the **`invoke-chained.sh`** script in the **`/home/student/DO328/labs/observe-kiali`** folder

```
[student@workstation observe-metrics]$ while true; \
> do curl ${GATEWAY_URL}/chained; \
> sleep 3;done
Hello World! -> Hola Mundo! -> Ahoj světe!
Hello World! -> Hola Mundo! -> Ahoj světe!
...output omitted...
```

4.2. Switch to the Kiali **Graph** page, and observe the traffic animation. You might have to wait for a few seconds while Kiali captures data from the Envoy proxies and renders the animation.



Note how the **`greet-api`** only calls the **`english`** service, which calls the other languages in a chain.

Do not interrupt the command line terminal where you are sending traffic to the **`/chained`** endpoint. Leave it running. You will need this for subsequent steps in this exercise.

5. Deploy version 2 of the **`english`** microservice and view the updated traffic flow in Kiali.

5.1. From a new command line terminal, run the **`oc create`** command to deploy version 2 of the **`english`** microservice. This new version prints a more informal greeting. The deployment resource is provided in the **`english-v2-deploy.yaml`** file in the **`/home/student/DO328/labs/observe-kiali`** folder.

```
[student@workstation ~]$ cd ~/DO328/labs/observe-kiali
[student@workstation observe-kiali]$ oc create -f english-v2-deploy.yaml
deployment.apps/english-v2 created
```

5.2. Run the **`oc get pods`** command and verify that version 2 of the **`english`** microservice is deployed and in **`Running`** state.

```
[student@workstation observe-jaeger]$ oc get pods
NAME                          READY   STATUS    RESTARTS   AGE
czech-84c5754796-cpqfq        2/2     Running   0          10m
english-v1-684884c897-qk7j2   2/2     Running   0          10m
english-v2-f696b69db-s285s    2/2     Running   0          27s
greet-api-7fb89fdc45-f7gs6    2/2     Running   0          10m
spanish-f8848fc89-c9slw       2/2     Running   0          10m
```

5.3. Switch to the command line terminal window running the curl command. After a while, you should see the output change to:

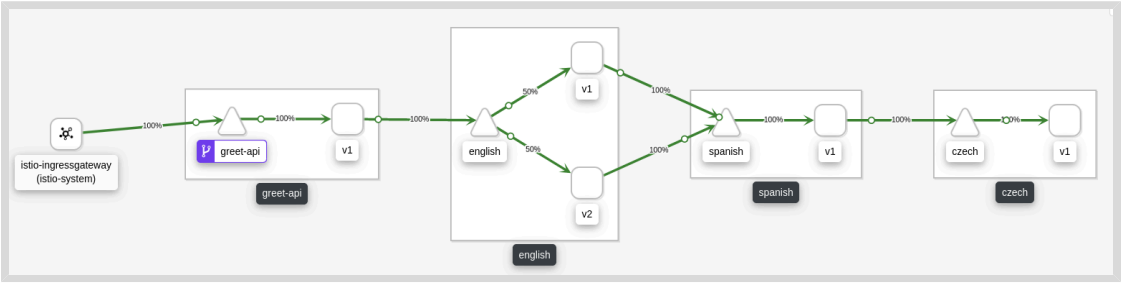
```
Hello World! -> Hola Mundo! -> Ahoj světe!
Hello World! -> Hola Mundo! -> Ahoj světe!
Hello World! -> Hola Mundo! -> Ahoj světe!
...output omitted...
Howdy! This mesh ain't big enough for both of us. -> Hola Mundo! -> Ahoj světe!
Hello World! -> Hola Mundo! -> Ahoj světe!
```

```
Howdy! This mesh ain't big enough for both of us. -> Hola Mundo! -> Ahoj světe!
Hello World! -> Hola Mundo! -> Ahoj světe!
...output omitted...
```

Traffic for the **`english`** service is split equally (load balanced) between both versions.

5.4. Switch to the Kiali **Graph** page, and observe the traffic animation. You might have to wait for a few seconds while Kiali captures data from the Envoy proxies and renders the animation.

Click **Service graph** and select **Versioned app graph**. This will change the graph to display versions of services.



Note

Your graph might not look exactly like the above. Graph nodes from previous scenarios might still be visible, but grayed out in the graph.

Note how Kiali shows the request percentage split equally between version 1 and version 2 of the **`english`** microservice.

6. Redirect all traffic bound for the **`english`** microservice to version 2 of the service. View the updated traffic flow in Kiali.

6.1. From a new command line terminal, run the **`oc create`** command to deploy version 2 of the **`english`** microservice. This new version prints a more informal greeting. The deployment resource is provided in the **`english-v2-all.yaml`** file in the **`/home/student/DO328/labs/observe-kiali`** folder.

Note

Do not worry about the details in the YAML resource file. You will learn more about traffic shaping and load balancing in subsequent chapters.

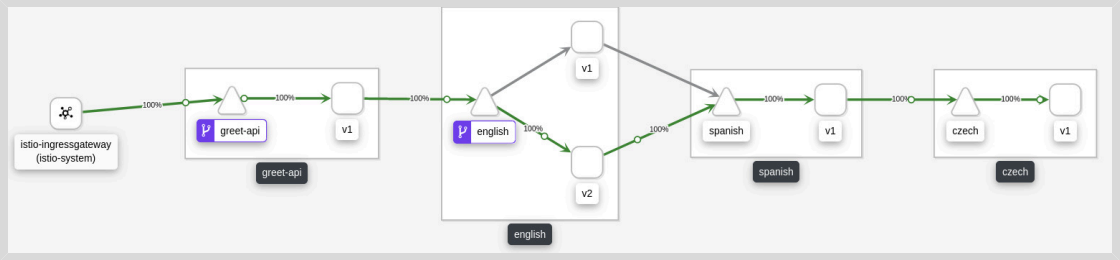
```
[student@workstation observe-kiali]$ oc create -f english-v2-all.yaml
destinationrule.networking.istio.io/english created
virtualservice.networking.istio.io/english-v2-all created
```

6.2. Switch to the command line terminal window running the curl command. After a while, you should see the output change to:

```
Hello World! -> Hola Mundo! -> Ahoj světe!
Howdy! This mesh ain't big enough for both of us. -> Hola Mundo! -> Ahoj světe!
Hello World! -> Hola Mundo! -> Ahoj světe!
...output omitted...
Howdy! This mesh ain't big enough for both of us. -> Hola Mundo! -> Ahoj světe!
Howdy! This mesh ain't big enough for both of us. -> Hola Mundo! -> Ahoj světe!
Howdy! This mesh ain't big enough for both of us. -> Hola Mundo! -> Ahoj světe!
Howdy! This mesh ain't big enough for both of us. -> Hola Mundo! -> Ahoj světe!
...output omitted...
```

All traffic for the **`english`** service is sent to version 2.

6.3. Switch to the Kiali **Graph** page, and observe the traffic animation. You might have to wait for a few seconds while Kiali captures data from the Envoy proxies and renders the animation.



Note how Kiali shows 100% of traffic being sent to version 2 of the **`english`** microservice.

6.4. Press **`Ctrl`**+**`C`** to stop the curl command in the command line terminal window where you were invoking the **`/chained`** endpoint.

7. Return to the home directory.

```
[student@workstation observe-kiali]$ cd ~
[student@workstation ~]$
```

Finish

On the **`workstation`** machine, use the **`lab`** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab observe-kiali finish
```

This concludes the guided exercise.

Lab

Observing an OpenShift Service Mesh

Performance Checklist

In this lab, you will troubleshoot performance issues with an application using distributed tracing, and enable metrics collection for the application.

Outcomes

You should be able to:

- Identify and fix performance issues with an application deployed on Red Hat OpenShift Service Mesh.
- Enable distributed tracing for a Quarkus based microservice and visualize the traces using Jaeger.
- Enable custom application metrics for a Quarkus based microservice and create a custom dashboard for visualizing the collected metrics.
- Visualize communication between services in the application using Kiali.

Before You Begin

To perform this lab, ensure you have:

- A configured and running OpenShift cluster.
- An installed and running OpenShift Service Mesh in the OpenShift cluster.
- The OpenShift client (oc), OpenJDK 1.8, and podman installed on workstation.

You will be using an application that simulates an online currency exchange for this lab. The application has four microservices:

- **`currencies`**: Written in Python using the **`Flask`** framework. It provides a REST API, which returns a list of all currencies supported by the exchange.
- **`history`**: Written in JavaScript using the **`Node.js`** runtime. It provides a REST API, which returns historical data of exchange rates between currencies.
- **`frontend`**: Written using HTML, JavaScript, and CSS using the **`React.js`** library. It provides a web user interface for users of the currency exchange application.
- **`exchange`**: Written in Java using the **`Quarkus`** framework. It acts as an API gateway and communicates with the **`currencies`** and **`history`** microservices. It acts as a single point of communication for the **`frontend`** microservice.

The source code for the four microservices are available in the **`exchange-traced`** folder in the GitHub repository at https://github.com/RedHatTraining/DO328-apps.

On the **`workstation`** machine, use the **`lab`** command to prepare your system for this exercise.

```
[student@workstation ~]$ lab observe-mesh start
```

The **`lab observe-mesh start`** command will create a new project called **`observe-mesh`** owned by the developer (the value of the **`$RHT_OCP4_DEV_USER`** environment variable in your **`/usr/local/etc/ocp4.config`** file) user. It will deploy an initial version of the four microservices in this project.

You can examine the template that deploys the microservices in the **`~/DO328/labs/observe-mesh/exchange-app-template.yaml`** file.

1. Log in to OpenShift as the developer user, and inspect the deployed applications. Verify that the four microservices are deployed and running.

   Do not forget to load the environment variables from the **`/usr/local/etc/ocp4.config`** file in your command line terminal.

2. Identify the gateway URL for the service mesh. The currency exchange application is available at the URL **`/frontend`** relative to the gateway URL.

   Explore the currency exchange application by clicking on the **Historical Data**, and **Exchange** menu options in the left navigation panel.

   You can ignore the **News** menu option because it requires communicating with an external service, which is not deployed in this lab.

   Note that fetching the historical data of currency rates in the **Historical Data** page is very slow. It takes more than 5 seconds to fetch the data.

3. Visualize the service mesh communication using the Kiali web console. View the traces and spans generated by the currency exchange application using the Jaeger web console.

   Identify the microservice that is causing the slow response time in the **Historical Data** page. From the traces and spans in the Jaeger console, you can identify the function name that is causing the slowdown.

   Although distributed tracing was enabled for the services that you originally deployed, the source code of the problematic microservice that is available in this lab does not include tracing instrumentation. Therefore, to keep distributed tracing active after fixing the slowdown problem, you will add tracing instrumentation when you make changes to the source code of the problematic microservice. You will clone the source code of this microservice from GitHub and add tracing code in a subsequent step of the lab.

Warning

The lab start script updates the **`ServiceMeshMemberRoll`** resource of the service mesh control plane. This will cause the Kiali pod to be redeployed after some time. Check the status of the Kiali pod by running **`oc get pods -n istio-system`**, and proceed after you see it in **`Running`** state.

If the Kiali pod is restarted after you have logged into Kiali, or if you see errors in the Kiali console, verify the Kiali pod status and log in again.

4. Clone the source code for the traced currency exchange application from the GitHub repository at https://github.com/RedHatTraining/DO328-apps. The application code is located in the **`exchange-traced`** directory. Correct the source code of the microservice you identified in the previous step to eliminate the performance issue.

   Use a text editor like **`VSCodium`**, which supports syntax highlighting for editing source files.

5. Enable distributed tracing for the problematic microservice.

   The distributed tracing properties for the problematic microservice must be configured as follows:

   - The service name for identifying traces and spans in Jaeger must be named **`exchange`**.
   - Enable Jaeger to collect all sample traces from all requests to the service.
   - The URL of the Jaeger collector is **`http://jaeger-collector.istio-system.svc:14268/api/traces`**.
   - Enable propagation of all **`x-b3-*`** HTTP headers, and log span information for all incoming and outgoing requests to and from this service.

6. You have been instructed by the operations team to add custom metrics to the problematic microservice, which will keep track of its performance in future deployments.

   Add the following custom metrics to the problematic microservice:

   - A timer that tracks how long (in milliseconds) the **`getHistoricalData()`** function takes to execute. Provide a unique name for this metric called **`exchange_svc:history_fetch_time`**.
   - The rate of requests (per minute) served by the **`getHistoricalData()`** function. This will provide information for capacity planning in the future. Provide a unique name for this metric called **`exchange_svc:history_fetch_rate`**.

7. Rebuild the container image for the problematic microservice. Fully complete Dockerfiles are provided to you for all four microservices in their respective folders.

   Create a new container image repository in Quay.io called **`ossm-`*`microservice`*`-traced`**. Replace **`microservice`** with the name of the problematic microservice.

   Push the newly built container image with a tag named **`1.0`** to the **`ossm-`*`microservice`*`-traced`** image repository.

8. Edit the deployment resource for the **`exchange`** service.

   Replace the container image for the problematic microservice with your newly built container image.

   Enable metrics collection for the problematic microservice by adding the correct annotations.

9. Test the application with the updated **`exchange`** microservice. Verify that the performance issues identified earlier are no longer present in the **Historical Data** page.

10. Create a custom dashboard in Grafana for the custom application metrics you added in a previous step.

11. Return to the home directory.

```
[student@workstation exchange]$ cd ~
[student@workstation ~]$
```

Evaluation

Grade your work by running the **`lab observe-mesh grade`** command from your **`workstation`** machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab observe-mesh grade
```

Finish

On the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab observe-mesh finish
```

This concludes the lab.

Solution

Observing an OpenShift Service Mesh

Performance Checklist

In this lab, you will troubleshoot performance issues with an application using distributed tracing, and enable metrics collection for the application.

Outcomes

You should be able to:

- Identify and fix performance issues with an application deployed on Red Hat OpenShift Service Mesh.
- Enable distributed tracing for a Quarkus based microservice and visualize the traces using Jaeger.
- Enable custom application metrics for a Quarkus based microservice and create a custom dashboard for visualizing the collected metrics.
- Visualize communication between services in the application using Kiali.

Before You Begin

To perform this lab, ensure you have:

- A configured and running OpenShift cluster.
- An installed and running OpenShift Service Mesh in the OpenShift cluster.
- The OpenShift client (oc), OpenJDK 1.8, and podman installed on workstation.

You will be using an application that simulates an online currency exchange for this lab. The application has four microservices:

- **`currencies`**: Written in Python using the **`Flask`** framework. It provides a REST API, which returns a list of all currencies supported by the exchange.
- **`history`**: Written in JavaScript using the **`Node.js`** runtime. It provides a REST API, which returns historical data of exchange rates between currencies.
- **`frontend`**: Written using HTML, JavaScript, and CSS using the **`React.js`** library. It provides a web user interface for users of the currency exchange application.
- **`exchange`**: Written in Java using the **`Quarkus`** framework. It acts as an API gateway and communicates with the **`currencies`** and **`history`** microservices. It acts as a single point of communication for the **`frontend`** microservice.

The source code for the four microservices are available in the **`exchange-traced`** folder in the GitHub repository at https://github.com/RedHatTraining/DO328-apps.

On the **`workstation`** machine, use the **`lab`** command to prepare your system for this exercise.

```
[student@workstation ~]$ lab observe-mesh start
```

The **`lab observe-mesh start`** command will create a new project called **`observe-mesh`** owned by the developer (the value of the **`$RHT_OCP4_DEV_USER`** environment variable in your **`/usr/local/etc/ocp4.config`** file) user. It will deploy an initial version of the four microservices in this project.

You can examine the template that deploys the microservices in the **`~/DO328/labs/observe-mesh/exchange-app-template.yaml`** file.

1. Log in to OpenShift as the developer user, and inspect the deployed applications. Verify that the four microservices are deployed and running.

   Do not forget to load the environment variables from the **`/usr/local/etc/ocp4.config`** file in your command line terminal.

   1.1. Run the following command to load the environment variables created in the first guided exercise:

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

   1.2. Log in to OpenShift:

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_API}
Login successful.
...output omitted...
```

   1.3. Set the current project to **`observe-mesh`**:

```
[student@workstation ~]$ oc project observe-mesh
Now using project "observe-mesh" on server ...output omitted...
```

   1.4. Verify that there are four pods in **`Running`** state:

```
[student@workstation ~]$ oc get pods
NAME                        READY   STATUS    RESTARTS   AGE
currency-cc8566cdd-fc4qx    2/2     Running   0          13m
exchange-8cf576667-9z7h5    2/2     Running   0          13m
frontend-7846899665-8bpwr   2/2     Running   0          13m
history-db65bfb86-rwmdr     2/2     Running   0          13m
```

2. Identify the gateway URL for the service mesh. The currency exchange application is available at the URL **`/frontend`** relative to the gateway URL.

   Explore the currency exchange application by clicking on the **Historical Data**, and **Exchange** menu options in the left navigation panel.

   You can ignore the **News** menu option because it requires communicating with an external service, which is not deployed in this lab.

   Note that fetching the historical data of currency rates in the **Historical Data** page is very slow. It takes more than 5 seconds to fetch the data.

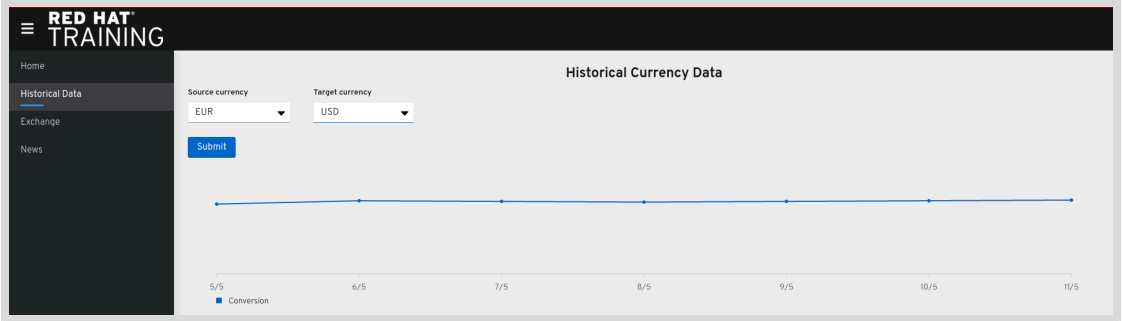
2.1. Run the **`oc get route`** command to gather the service mesh gateway URL.

```
[student@workstation ~]$ export  \
> GW_URL=$(oc get route istio-ingressgateway -n istio-system \
> -o template --template '{{ "http://" }}{{ .spec.host }}')
```

2.2. Access the currency exchange application using the **`firefox`** browser on your workstation.

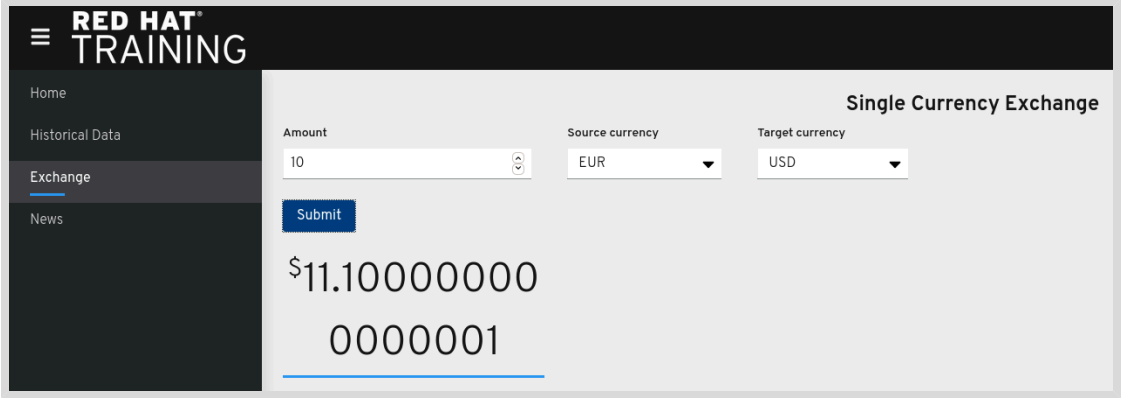
```
[student@workstation ~]$ firefox ${GW_URL}/frontend &
```

2.3. Click **Historical Data** in the left navigation panel. Select a source and target currency and click **Submit** to see historical exchange data.



Note the slow response after you click **Submit** to fetch historical data.

2.4. Click **Exchange** in the left navigation panel. Enter an amount, select the source and target currency and click **Submit**.



3. Visualize the service mesh communication using the Kiali web console. View the traces and spans generated by the currency exchange application using the Jaeger web console.

Identify the microservice that is causing the slow response time in the **Historical Data** page. From the traces and spans in the Jaeger console, you can identify the function name that is causing the slowdown.

Although distributed tracing was enabled for the services that you originally deployed, the source code of the problematic microservice that is available in this lab does not include tracing instrumentation. Therefore, to keep distributed tracing active after fixing the slowdown problem, you will add tracing instrumentation when you make changes to the source code of the problematic microservice. You will clone the source code of this microservice from GitHub and add tracing code in a subsequent step of the lab.

Warning

The lab start script updates the **`ServiceMeshMemberRoll`** resource of the service mesh control plane. This will cause the Kiali pod to be redeployed after some time. Check the status of the Kiali pod by running **`oc get pods -n istio-system`**, and proceed after you see it in **`Running`** state.

If the Kiali pod is restarted after you have logged into Kiali, or if you see errors in the Kiali console, verify the Kiali pod status and log in again.

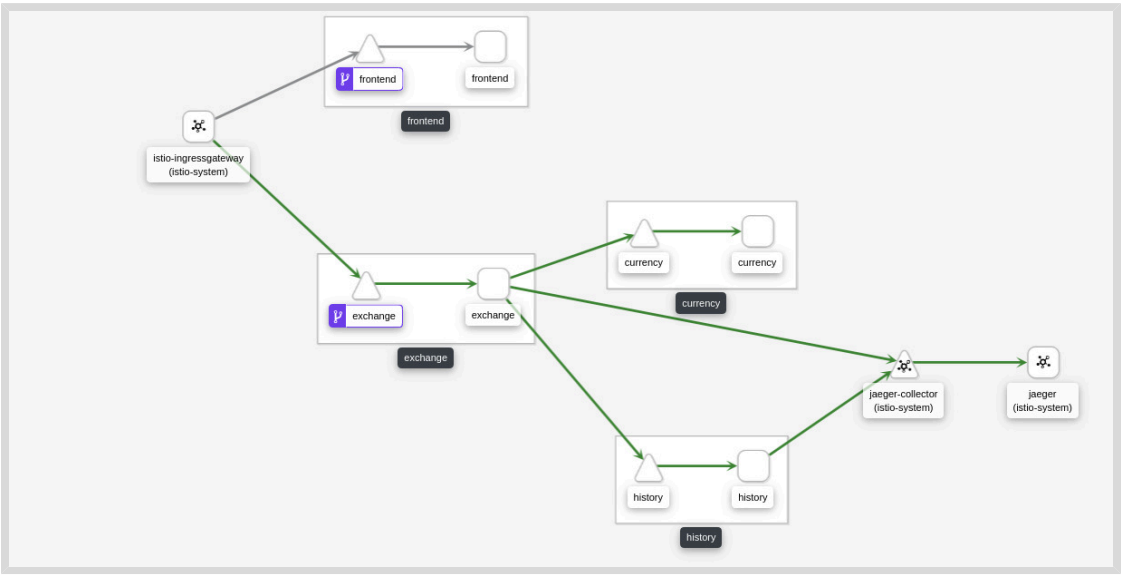
3.1. Run the **`oc get route`** command to gather the Kiali web console URL.

```
[student@workstation ~]$ export  \
> KIALI_URL=$(oc get route kiali -n istio-system \
> -o template --template '{{ "https://" }}{{ .spec.host }}')
```

3.2. Access the Kiali web console using the **`firefox`** browser on your workstation. Log in using your OpenShift developer user account.

```
[student@workstation ~]$ firefox ${KIALI_URL} &
```

3.3. Click **Graph** in the left navigation panel, and select the **observe-mesh** namespace to view the service mesh graph. Your graph could look different than the one shown below.



Note

You will not see the **`currency`** microservice sending traces to the **`jaeger-collector`** because, the currency service is written in Python and does not yet support sending traces to jaeger over TCP. Instead the microservice sends trace information using UDP datagrams to the **`jaeger-agent`** service, which forwards it to the jaeger backend. Kiali cannot capture this UDP traffic to render the graphs.

You will see traces from the **`currency`** microservice in the Jaeger web console and verify this in the next step.

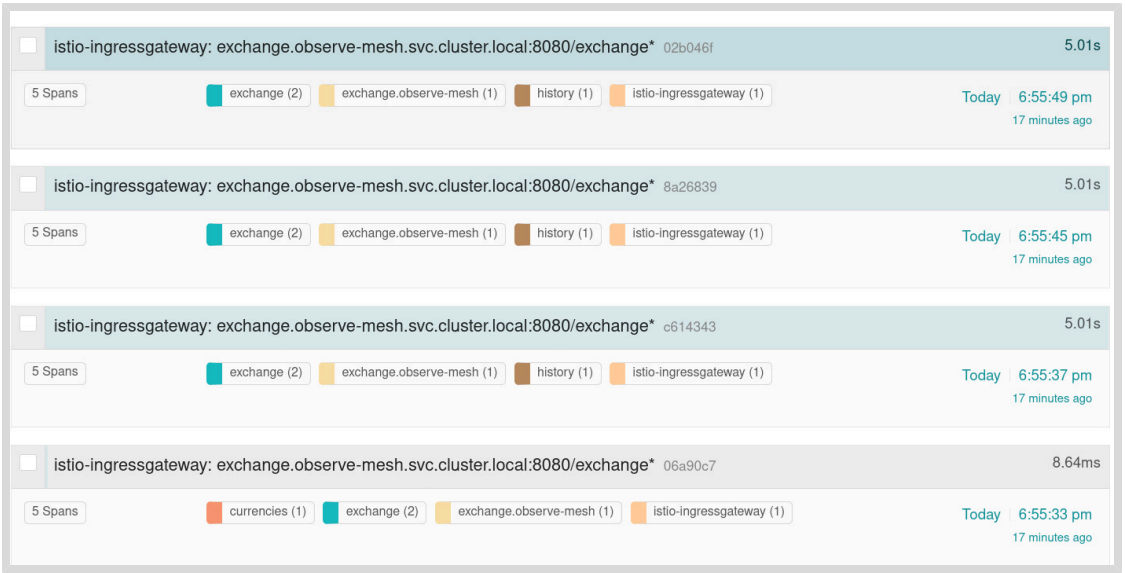
3.4. Run the **`oc get route`** command to gather the Jaeger web console URL.

```
[student@workstation ~]$ export  \
> JAEGER_URL=$(oc get route jaeger -n istio-system \
> -o template --template '{{ "https://" }}{{ .spec.host }}')
```

3.5. Use the Firefox browser to access the Jaeger web console and log in using your OpenShift developer user account.

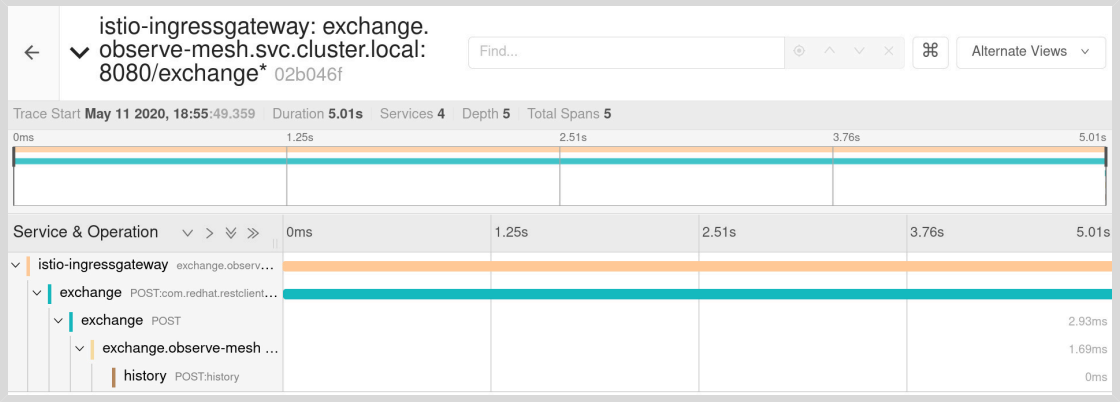
```
[student@workstation ~]$ firefox ${JAEGER_URL} &
```

3.6. In the Jaeger web console, select the **istio-ingressgateway** service in the search panel on the left of the page. Click **Find Traces**. You will see a list of traces for the currency exchange application.

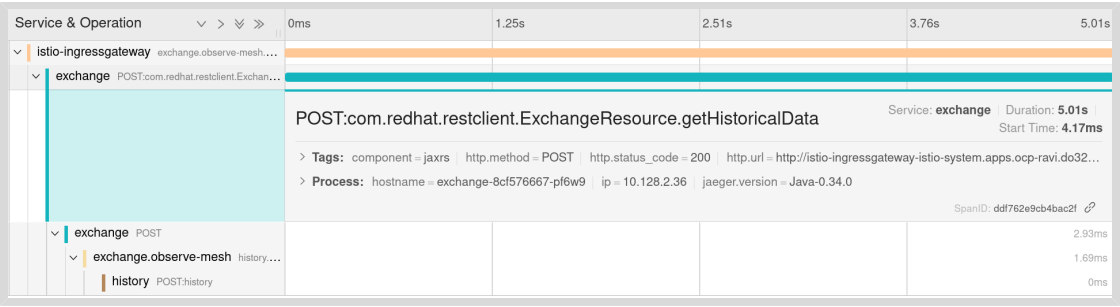


3.7. Click any trace that is greater than 5 seconds. These traces are generated by the functionality in the **Historical Data** page of the currency exchange application.

Note the hierarchy of service calls for this trace. The front end calls the **exchange** service, which then calls the **`history`** service to fetch historical data.



3.8. To identify the method causing the bottleneck, click the span corresponding to the **exchange** service. This span has the longest execution time (> 5 seconds on average).



There seems to be an issue in the **getHistoricalData** function of the **com.redhat.restclient.ExchangeResource** class in the **exchange** service. The class name and function is displayed as the title of the span (the text following **POST:**).

4. Clone the source code for the traced currency exchange application from the GitHub repository at https://github.com/RedHatTraining/DO328-apps. The application code is located in the **exchange-traced** directory. Correct the source code of the microservice you identified in the previous step to eliminate the performance issue.

Use a text editor like **VSCodium**, which supports syntax highlighting for editing source files.

4.1. If you have not cloned the source code from the **DO328-apps** GitHub repository in a previous exercise, then do so now using the **git clone** command.

```
[student@workstation ~]$ git clone https://github.com/RedHatTraining/DO328-apps
...output omitted...
Cloning into 'DO328-apps'...
...output omitted...
```

4.2. Copy the contents of the **/home/student/DO328-apps/exchange-traced** folder from your local Git repository to the **/home/student/DO328/labs/observe-mesh** folder.

```
[student@workstation ~]$ cp -Rv ~/DO328-apps/exchange-traced \
> ~/DO328/labs/observe-mesh/
```

4.3. Fix the issue in the source code for the **exchange** microservice. Edit the **/home/student/DO328/labs/observe-mesh/exchange-traced/exchange/src/main/java/com/redhat/restclient/ExchangeResource.java** file.

Note the hard coded **Thread.sleep(5000)** try-catch block in the **getHistoricalData()** method. This was added to simulate slow processing (for example, due to an inefficient algorithm, a slow database query, or high latency due to invoking external services).

Remove the try-catch block that was causing the slowdown.

5. Enable distributed tracing for the problematic microservice.

The distributed tracing properties for the problematic microservice must be configured as follows:

- The service name for identifying traces and spans in Jaeger must be named **exchange**.
- Enable Jaeger to collect all sample traces from all requests to the service.
- The URL of the Jaeger collector is **http://jaeger-collector.istio-system.svc:14268/api/traces**.
- Enable propagation of all **x-b3-*** HTTP headers, and log span information for all incoming and outgoing requests to and from this service.

5.1. Enable distributed tracing for the **exchange** microservice.

Edit the **/home/student/DO328/labs/observe-mesh/exchange-traced/exchange/pom.xml** file and add the **quarkus-smallrye-opentracing** dependency.

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-smallrye-opentracing</artifactId>
</dependency>
```

5.2. Edit the **/home/student/DO328/labs/observe-mesh/exchange-traced/exchange/src/main/resources/application.properties** file and add the Jaeger related properties.

```
quarkus.jaeger.service-name=exchange
quarkus.jaeger.sampler-type=const
quarkus.jaeger.sampler-param=1
quarkus.jaeger.endpoint=http://jaeger-collector.istio-system.svc:14268/api/traces
quarkus.jaeger.propagation=b3
quarkus.jaeger.reporter-log-spans=true
```

6. You have been instructed by the operations team to add custom metrics to the problematic microservice, which will keep track of its performance in future deployments.

Add the following custom metrics to the problematic microservice:

- A timer that tracks how long (in milliseconds) the **getHistoricalData()** function takes to execute. Provide a unique name for this metric called **exchange_svc:history_fetch_time**.

- The rate of requests (per minute) served by the **`getHistoricalData()`** function. This will provide information for capacity planning in the future. Provide a unique name for this metric called **`exchange_svc:history_fetch_rate`**.

6.1. Edit the **`/home/student/DO328/labs/observe-mesh/exchange-traced/exchange/pom.xml`** file and add the **`quarkus-smallrye-metrics`** dependency.

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-smallrye-metrics</artifactId>
</dependency>
```

6.2. Edit the source code for the **`getHistoricalData()`** function in the **`/home/student/DO328/labs/observe-mesh/exchange-traced/exchange/src/main/java/com/redhat/restclient/ExchangeResource.java`** file.

Import the required metrics classes from the OpenTracing API. Add the following import statements at the top of the file.

```
...output omitted...
import org.eclipse.microprofile.metrics.MetricUnits;
import org.eclipse.microprofile.metrics.annotation.SimplyTimed;
import org.eclipse.microprofile.metrics.annotation.Metered;
...output omitted...
```

Add the following annotations to the **`getHistoricalData()`** function (below the **`@Path("/historicalData")`** line) to enable custom metrics.

```
@SimplyTimed(name = "exchange_svc:history_fetch_time",
           description = "A measure of how long it takes to fetch history data",
           unit = MetricUnits.MILLISECONDS)
@Metered(name = "exchange_svc:history_fetch_rate",
           unit = MetricUnits.MINUTES,
           description = "Rate at which historical data is fetched (minutes)",
           absolute = true)
```

7. Rebuild the container image for the problematic microservice. Fully complete Dockerfiles are provided to you for all four microservices in their respective folders.

Create a new container image repository in Quay.io called **`ossm-`*`microservice`*`-traced`**. Replace **`microservice`** with the name of the problematic microservice.

Push the newly built container image with a tag named **`1.0`** to the **`ossm-`*`microservice`*`-traced`** image repository.

7.1. Create a new public container image repository called **`ossm-exchange-traced`** in Quay.io. To create a public container image repository, refer to instructions in the *Appendix: Creating a Quay Account*.

Warning

If you skip this step and push the container images without creating public repositories, the **`podman push`** commands will create private container image repositories by default.

7.2. To ensure that there are no syntax errors, run **`mvn clean package`** and ensure that a fat JAR is created in the **`target`** folder.

```
[student@workstation ~]$ cd \
> ~/DO328/labs/observe-mesh/exchange-traced/exchange
[student@workstation exchange]$ mvn clean package
...output omitted...
[INFO] BUILD SUCCESS
...output omitted...
```

7.3. Review the **`Dockerfile`** for the **`exchange`** microservice. You will use Red Hat Universal Base Images (UBI) as the base for building your container image. Do not make any changes to it.

Build the container image using **`podman`**.

```
[student@workstation exchange]$ podman build -t \
> quay.io/${RHT_OCP4_QUAY_USER}/ossm-exchange-traced:1.0 .
STEP 1: FROM registry.access.redhat.com/ubi8:8.1
...output omitted...
STEP 15: COMMIT quay.io/youruser/ossm-exchange-traced:1.0
```

7.4. Log in to your Quay.io account using **`podman`**.

```
[student@workstation exchange]$ podman login -u ${RHT_OCP4_QUAY_USER} quay.io
```

You will be prompted for your Quay.io password.

7.5. Push the updated container image for the **`exchange`** microservice to Quay.io.

```
[student@workstation exchange]$ podman push \
> quay.io/${RHT_OCP4_QUAY_USER}/ossm-exchange-traced:1.0
...output omitted...
Writing manifest to image destination
Storing signatures
```

8. Edit the deployment resource for the **`exchange`** service.

Replace the container image for the problematic microservice with your newly built container image.

Enable metrics collection for the problematic microservice by adding the correct annotations.

8.1. Edit the deployment resource for the **`exchange`** microservice.

```
[student@workstation exchange]$ oc edit deployment exchange
```

Edit the **`spec.template.spec.containers.image`** attribute for the **`exchange`** deployment, and add the Quay.io URL of the container image for the **`exchange`** deployment.

```
...output omitted...
spec:
      containers:
      - env:
```

```
        - name: NEWS_ENDPOINT
          value: http://feed-news.apps-crc.testing
        image: quay.io/youruser/ossm-exchange-traced:1.0
        imagePullPolicy: Always
        name: exchange
...output omitted...
```

8.2. Add annotations to allow the Prometheus instance to collect metrics from the **exchange** microservice. Add the following annotations below the **sidecar.istio.io/inject: "true"** annotation for the **exchange** microservice.

```
...output omitted...
template:
  metadata
    annotations:
    sidecar.istio.io/inject: "true"
    prometheus.io/scrape: "true"
    prometheus.io/port: "8080"
    prometheus.io/scheme: "http"
...output omitted...
```

Save your changes. OpenShift will detect the changed deployment, and delete the old pod. It will create a new pod for the **exchange** microservice.

8.3. Verify that the four microservices are ready and running using the **oc get pods** command.

```
[student@workstation exchange]$ oc get pods
NAME                        READY   STATUS    RESTARTS   AGE
currency-699d4dd98d-rzlvx   2/2     Running   0          42s
exchange-5454495f78-47457   2/2     Running   0          41s
frontend-5b78fc6bb5-ztdvq   2/2     Running   0          43s
history-8c875469d-csfsv     2/2     Running   0          42s
```

9. Test the application with the updated **exchange** microservice. Verify that the performance issues identified earlier are no longer present in the **Historical Data** page.

9.1. Access the currency exchange application at the URL **GW_URL/frontend**. Click **Historical Data**, and verify that the response time is much improved (the spinner icon after you click **Submit** should disappear quickly).

9.2. You will see span information from the **exchange** microservice. Access the Jaeger console and verify that there is no longer a 5 second pause in the span details for the historical data page.

10. Create a custom dashboard in Grafana for the custom application metrics you added in a previous step.

10.1. Run the **oc get route** command to gather the Grafana web console URL.

```
[student@workstation exchange]$ export  \
> GRAFANA_URL=$(oc get route grafana -n istio-system \
> -o template --template '{{ "https://" }}{{ .spec.host }}')
```

10.2. Use the Firefox web browser to access the Grafana web console. Log in using your developer user account.

```
[student@workstation exchange]$ firefox ${GRAFANA_URL} &
```

10.3. Click the plus (+) icon in the left navigation menu of Grafana web console to open the **Create** menu. Select **Dashboard** to create a new dashboard.

10.4. Click **Add Query** in the **New Panel** panel.

In the query expression editor named **A** (next to the **Metrics** label), type **`history_fetch_time`** and select the first auto completed option (ends with **`elapsed`**). Click the graph icon in the left menu (to the left of the **Query** window) to select how you want to visualize the metric.

In the **Visualization** panel, expand the **Visualization** selection, and select **Graph**.

10.5. Click **General** (gear icon below **Visualize** icon in the left menu).

Enter **`Historical Data Fetch Time`** as the **Title**.

Click the left arrow icon (top left corner to the left of **New dashboard**) to go back to the dashboard page.

10.6. Click **Add panel** (graph icon with yellow plus in top right) to add another panel.

Click **Add Query** in the **New Panel**, and type **`history_fetch_rate`**. Prometheus will auto-complete and provide you with 5 options. Select the option ending with **`fetch_rate_total`**.

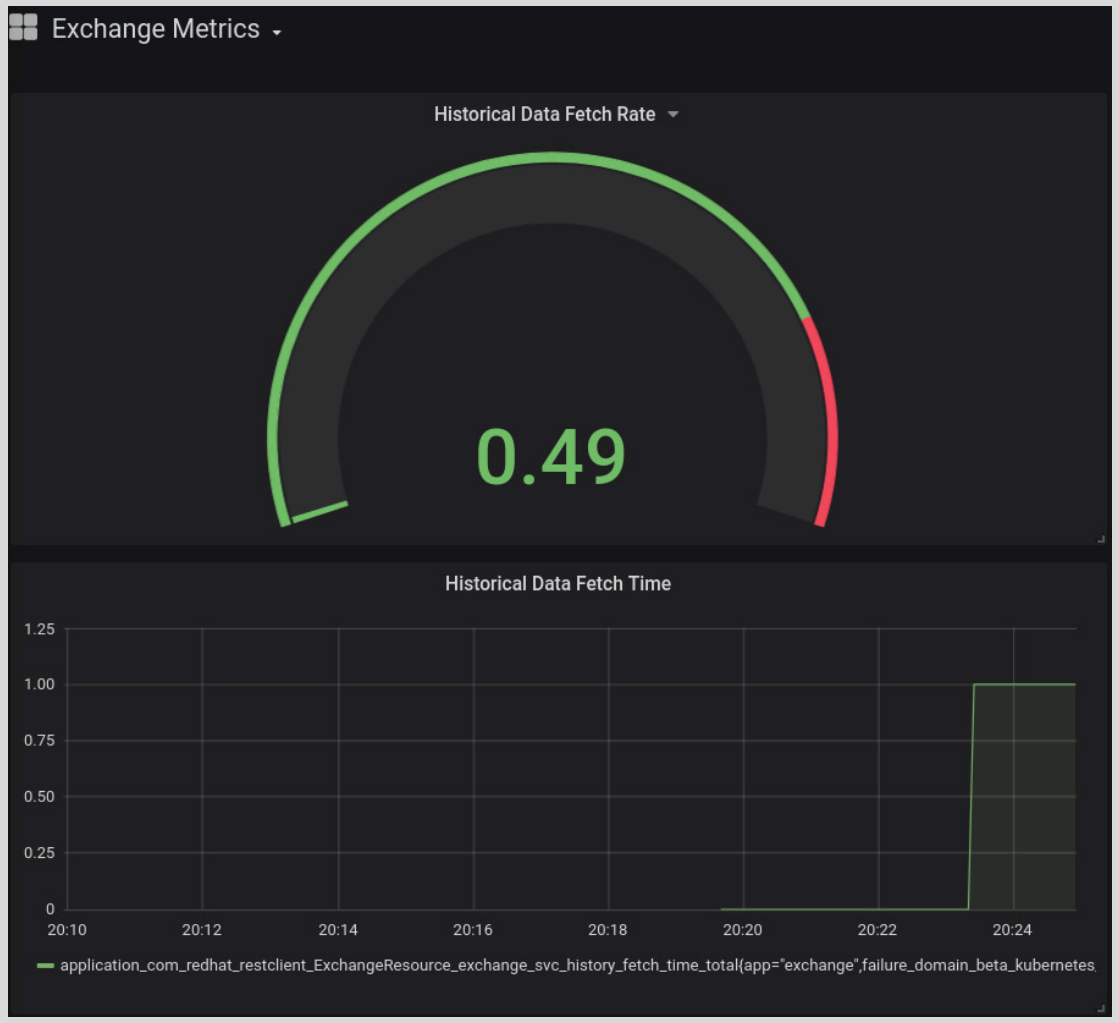
Click **Visualization** in the left menu, and select the **Gauge** option.

Click **General** in the left panel, and change the **Title** field to **`Historical Data Fetch Rate`**. Click the left arrow icon in the top left corner to go back to the **New dashboard** page.

10.7. Your new dashboard should now show both metrics from the **`exchange`** service.

Click the **Save dashboard** icon to save your new dashboard. Enter **`Exchange Metrics`** in the **Save As** dialog.

Your final Grafana dashboard should look like the following.



11. Return to the home directory.

```
[student@workstation exchange]$ cd ~
[student@workstation ~]$
```

Evaluation

Grade your work by running the **`lab observe-mesh grade`** command from your **`workstation`** machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab observe-mesh grade
```

Finish

On the **`workstation`** machine, use the **`lab`** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab observe-mesh finish
```

This concludes the lab.

Summary

In this chapter, you learned:

- Distributed tracing helps you identify performance issues in microservices based applications. You need to instrument your applications with Jaeger and OpenTracing libraries to enable distributed tracing.
- You can use Prometheus libraries to instrument your applications for generating custom application metrics. You can use Grafana to generate custom dashboards for your application metrics.
- Kiali can be used to visualize the communication between services in your service mesh.

Chapter 4

Controlling Service Traffic

Goal

Manage and route traffic with Red Hat OpenShift Service Mesh

Objectives

- Manage and route traffic with Red Hat OpenShift Service Mesh.
- Route traffic to services in a mesh, based on request headers.
- Control egress traffic to access external services.

Sections

- Managing Service Connections with Envoy and Pilot (and Guided Exercise)
- Routing Traffic Based on Request Headers (and Guided Exercise)
- Accessing External Services (and Guided Exercise)

Lab

Controlling Service Traffic

Managing Service Connections with Envoy and Pilot

Objectives

After completing this section, you should be able to manage and route traffic with Red Hat OpenShift Service Mesh.

Explaining Traffic Management

Traffic management, in the context of cloud-native microservices, is the process of monitoring and controlling the external and internal network communications of an application.

As cloud-native applications evolve and grow, the number of microservices increases, which makes it more difficult to manage and observe the network connections in these kinds of applications. In addition to an increasing number of services to manage, the reliability of microservice applications is dependent on a reliable network.. For these reasons, developers need a way to improve reliability, security, and observability of the network connections.

OpenShift Service Mesh abstracts network communications, allowing you to manage them using Kubernetes custom resources. With OpenShift Service Mesh, you control the flow of traffic and API calls of your applications.

Projects that are not using Service Mesh usually use libraries embedded in the source code of the application.

Describing the Sidecar Pattern

Cloud applications usually require functionalities outside of the application domain, such as monitoring, logging, and authentication.

The *sidecar pattern* is an architectural pattern where a main process (the main application) segregates non-business related functionalities to an auxiliary process (the sidecar).

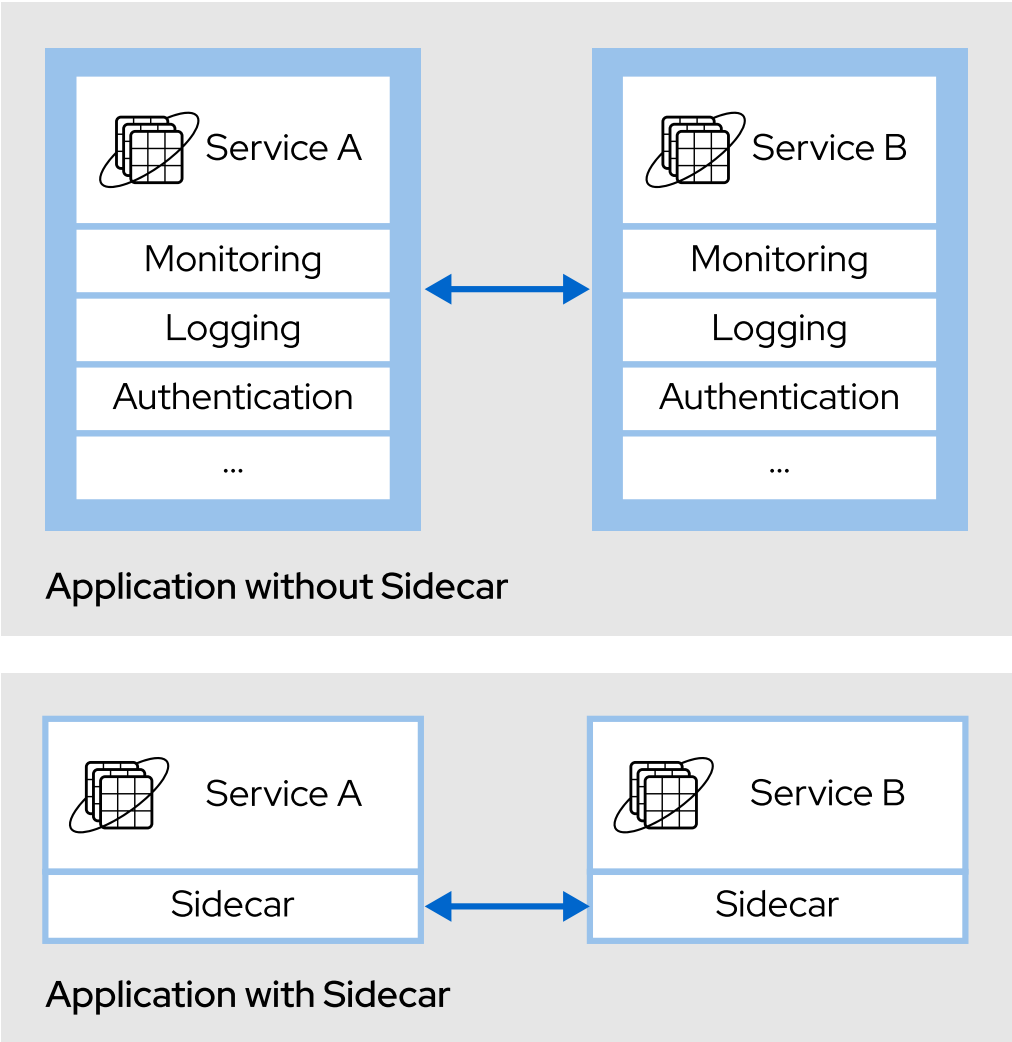


Figure 4.1: The sidecar pattern

Applying the sidecar pattern minimizes coupling between the application and the underlying infrastructure and reduces the complexity of the main application.

OpenShift Service Mesh and the Istio project are good examples of implementing the sidecar pattern at a network level.

The Sidecar Pattern and OpenShift Service Mesh

On Service Mesh enabled services, an Envoy proxy instance is injected into the application pod using the sidecar pattern. After the Envoy sidecar is injected, it takes control of all of the network communications for the pod.

OpenShift Service Mesh does not automatically inject the sidecar into every pod. You must explicitly specify the pods that you want Service Mesh to manage by adding annotations to the deployment configuration. This manual approach ensures that the automatic sidecar injection does not interfere with other Red Hat OpenShift features.

Note

The default configuration in the upstream Istio version uses automatic sidecar injection at a namespace level.

Injecting the Envoy Sidecar Automatically

To automatically inject the Envoy sidecar into a service, you must specify the **`sidecar.istio.io/inject`** annotation with the value set to **`"true"`** in the **`Deployment`** resource.

Example of automatic sidecar injection:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: history
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: history
      annotations:
        sidecar.istio.io/inject: "true"
    spec:
      containers:
        - name: history
          image: quay.io/redhattraining/ossm-history:1.0
          ports:
            - containerPort: 8080
```

Distinguishing Ingress and Egress Traffic

In the context of a Service Mesh, we can distinguish two types of network traffic:

Ingress
: Incoming traffic from sources external to the cluster, as well as calls originating from other services within the cluster.

Egress
: Outgoing traffic to sources outside the cluster, as well as calls to services within the cluster.

The default installation of OpenShift Service Mesh provides an instance of **`istio-ingressgateway`** and an instance of **`istio-egressgateway`** to manage the ingress and egress traffic in a Service Mesh. Both gateways can be customized to suit the needs of your applications.

Understanding the Gateway Custom Resource

A *gateway* is a custom resource that operates as a load balancer at the edge of the Service Mesh, managing ingress or egress connections. Gateway configurations are applied to Envoy proxies running at the edge of the Service Mesh.

In a default installation of OpenShift Service Mesh, a gateway instance called **`istio-ingressgateway`** manages ingress connections and a gateway instance called **`istio-egressgateway`** manages egress connections.

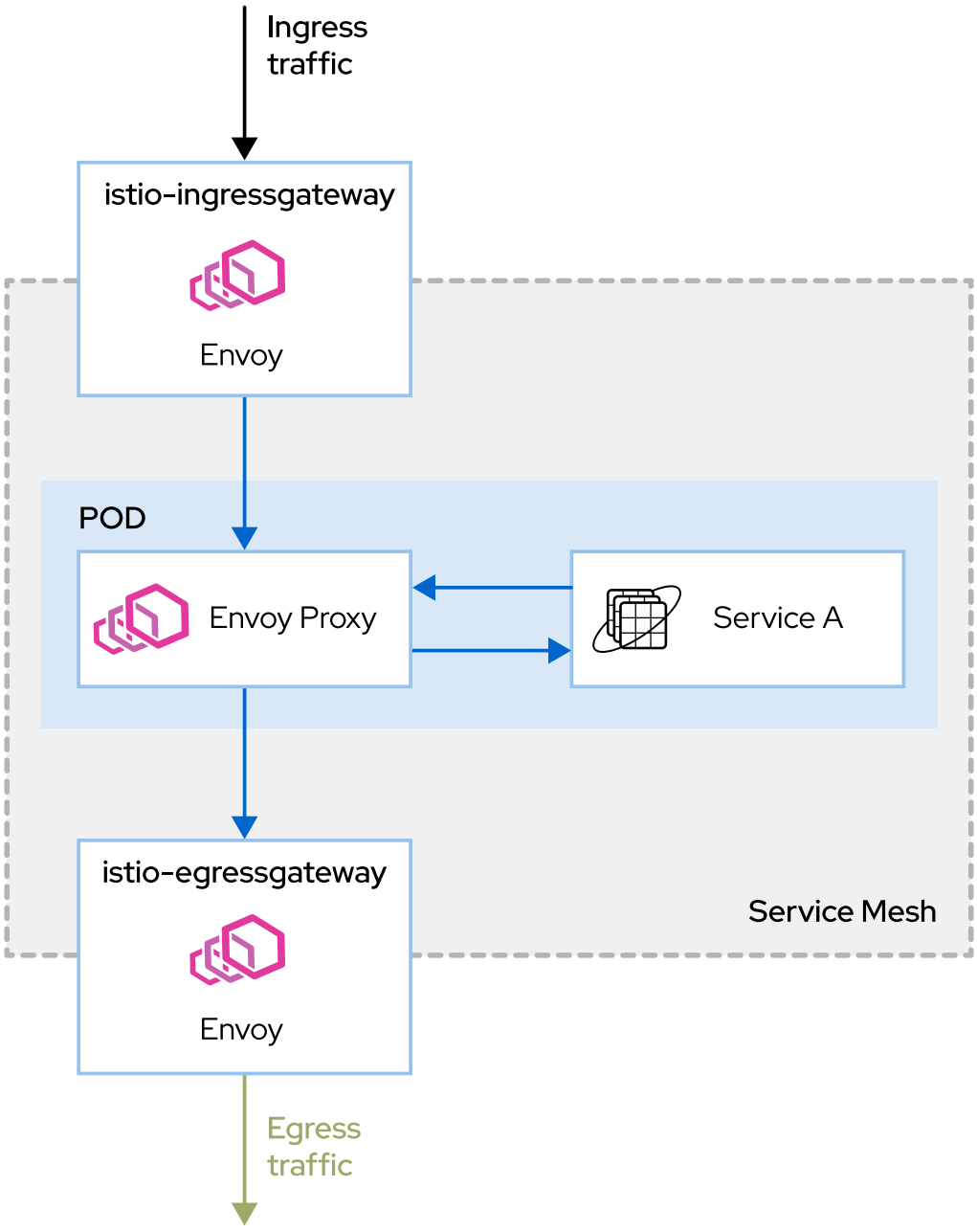


Figure 4.2: Ingress and egress gateways

The Open Systems Interconnection (OSI) model is a conceptual model that splits networking systems into seven abstraction layers. It provides a standard to describe how applications communicate over the network.

A gateway follows the OSI model, letting you configure Layer 4, Layer 5, and Layer 6 load balancing properties, and also delegates application-layer traffic routing (Layer 7) to virtual

services. This way of splitting the configuration between different and specialized components gives you more control and flexibility over the communications of your application.

Example of ingress gateway:

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: exchange-gw
spec:
  selector:
    istio: ingressgateway
  servers:
    - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
      - "exchange.example.com"
```

The preceding setup configures the **`ingressgateway`** to expose the combination of virtual hostname/DNS **`exchange.example.com`** protocol **`HTTP`** and port **`80`**. This means that all HTTP traffic for that hostname/DNS is allowed to enter the mesh.

Describing the VirtualService Custom Resource

A *virtual service* is a Kubernetes custom resource, which allows you to configure how the requests to services in the Service Mesh are routed.

A virtual service is composed of a list of routing rules that are evaluated in order, from top to bottom. Each routing rule consists of a traffic destination and zero or more match conditions that, if met, direct traffic to the destination defined by the rule.

Pilot translates the **`VirtualService`** custom resources to Envoy configuration, thereby propagating it to the data plane. In the absence of virtual services, Envoy distributes traffic between all service instances using a round-robin algorithm.

Example of a virtual service:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: exchange-vs (1)
spec:
  hosts: (2)
  - exchange
  http: (3)
  - match: (4)
    - headers:
        end-user:
          exact: test
    route:
    - destination:
        host: reviews
        subset: v2
```

```
  - route: ❺
    - destination:
        host: exchange
        subset: v1
```

❶ Virtual service name.
❷ List of destinations to which these routing rules apply.
❸ List of routing rules to apply to HTTP/1.1, HTTP2, and gRPC traffic.
❹ Rule with a match condition.
❺ Default no condition rule. All traffic will be routed to the specified destination if no previous conditions are met.

The preceding example redirects all requests containing the header **`end-user`** and the value **`test`** from the **`exchange`** service to the subset **`v2`**. In any other case, the match condition is not fired so the traffic is redirected to the default destination.

Combining Gateways and Virtual Services

To make gateways and virtual services work together, you must bind them using the **`gateways`** field for the virtual service. As a result, the traffic managed by the gateways listed in the gateways field is checked against the routing rules defined in the virtual service.

Example of virtual service bound to a gateway:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: exchange-vs
spec:
  hosts:
  - exchange
  gateways:
  - exchange-gw
  http:
  - match:
    - headers:
        end-user:
          exact: test
    route:
    - destination:
        host: reviews
        subset: v2
  - route:
    - destination:
        host: exchange
        subset: v1
```

In the preceding example the **`exchange-gw`** gateway is bound to the **`exchange-vs`** virtual service.

Ingress flow with OpenShift Service Mesh

In a default OpenShift Service Mesh installation, an OpenShift route is assigned to the Istio ingress gateway (**`istio-ingressgateway`**), which is the resource in charge of managing the

routing inside the mesh. All ingress traffic originating from outside the service mesh flows through this gateway into the service mesh.

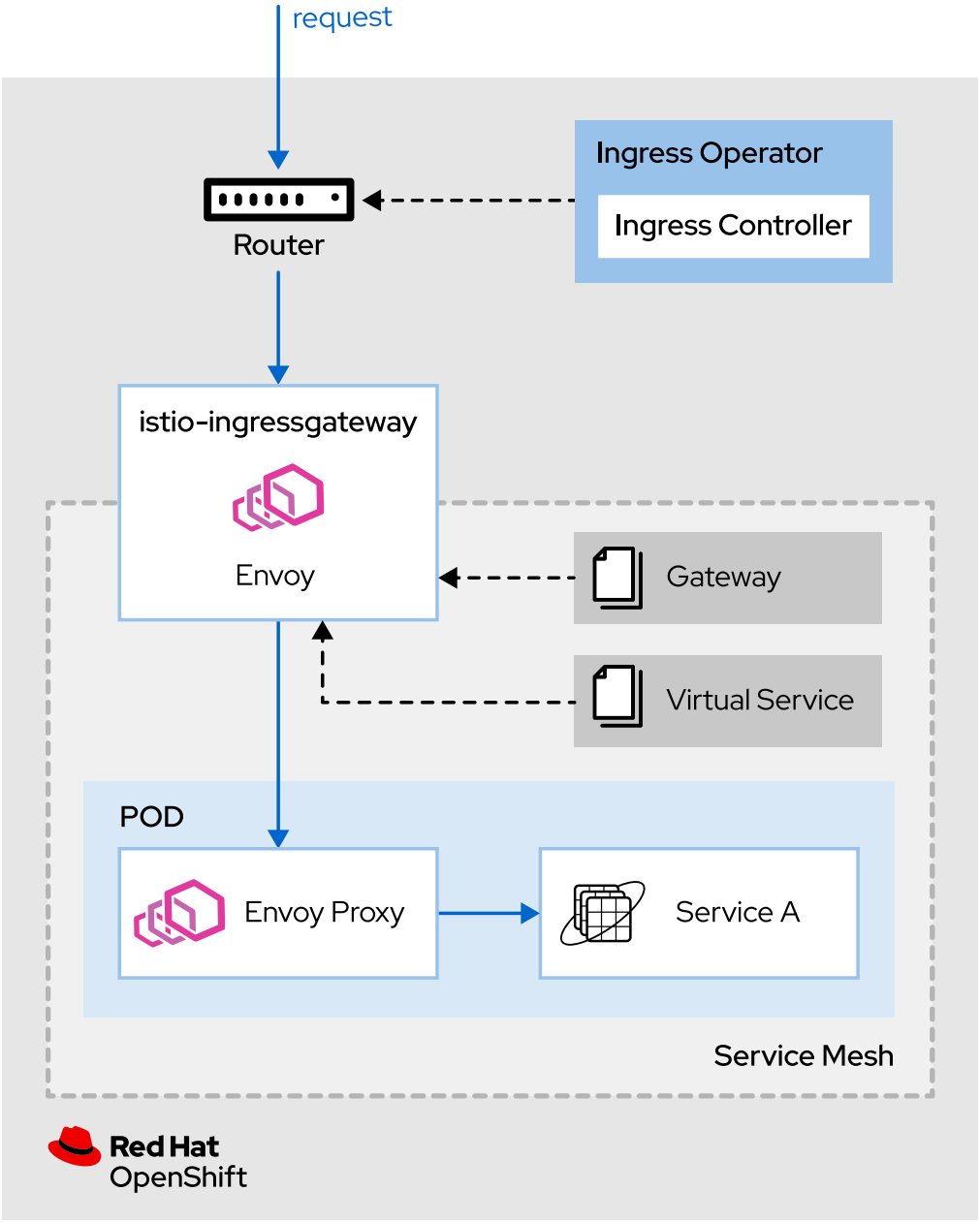


Figure 4.3: Ingress request flow in Red Hat OpenShift Service Mesh

The ingress request flow is as follows:

1. A external request enters the cluster.

2. A router instance checks the routing rules implemented by the ingress controller. If a match is found, the request is sent to the ingress gateway service pod (**`istio-ingressgateway`**).

3. The ingress gateway service pod evaluates the request against the **`Gateway`** configurations to check if the request matches with any configuration. If a match is found, the request is allowed to enter the mesh.

4. The ingress gateway service pod evaluates the **`VirtualService`** rules to find the application service pod in charge of processing the request.

5. If a **`VirtualService`** rule is matched, the ingress gateway service pod sends the request to the designated pod to process the request.

References

Traffic management with Istio
https://archive.istio.io/v1.4/docs/concepts/traffic-management/

Automatic sidecar injection in Istio
https://archive.istio.io/v1.4/docs/setup/additional-setup/sidecar-injection/#automatic-sidecar-injection

For more information, refer to the *Ingress Operator in OpenShift Container Platform* section in the *OpenShift Container Platform Service Mesh* documentation at
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.4/html-single/service_mesh/index

Decyphering the OSI model of networking
https://www.redhat.com/sysadmin/osi-model-bean-dip

References

For more information, refer to the *Red Hat OpenShift Service Mesh's sidecar injection* section in the *OpenShift Container Platform Service Mesh* documentation at
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.4/html-single/service_mesh/index

Guided Exercise

Exposing a Service

In this exercise, you will deploy and expose an application in Red Hat OpenShift Service Mesh.

Outcomes

You should be able to deploy applications to OpenShift Service Mesh and allow clients outside the service mesh to invoke them.

Before You Begin

To perform this exercise, ensure you have access to:

- A configured and running OpenShift cluster.
- A installed and running OpenShift Service Mesh in the OpenShift cluster.
- The OpenShift CLI (**/usr/local/bin/oc**).

As the **student** user on the **workstation** machine, use the **lab** command to validate the prerequisites for this exercise:

```
[student@workstation ~]$ lab traffic-deploy start
```

1. Log in to the OpenShift cluster.

   1.1. Source the classroom configuration file that is accessible at **/usr/local/etc/ocp4.config**.

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

   1.2. Log in to OpenShift as the **developer** user.

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_API}
Login successful.
...output omitted...
```

2. Create the **hello** project and and then add it to the **ServiceMeshMemberRoll** resource.

   2.1. The required YAML files and scripts for this guided exercise are located in **/home/student/DO328/labs/traffic-deploy**. Change to that directory using the **cd** command.

```
[student@workstation ~]$ cd /home/student/DO328/labs/traffic-deploy
[student@workstation traffic-deploy]$
```

2.2. Create the **`hello`** project.

```
[student@workstation traffic-deploy]$ oc new-project hello
Now using project "hello" on server
"https://api.cluster.domain.example.com:6443".
...output omitted...
```

2.3. Add the **`hello`** project to the list of members in the **`ServiceMeshMemberRoll`** resource. The **`ServiceMeshMemberRoll`** is available in the **`istio-system`** project.

Use the **`add-project-to-smmr.sh`** script to add the **`hello`** project to the list of members in the **`ServiceMeshMemberRoll`** resource.

```
[student@workstation traffic-deploy]$ sh add-project-to-smmr.sh
servicemeshmemberroll.maistra.io/default patched
```

3. Deploy a simple Vert.X application. This application exposes a single GET endpoint that returns "Hello World!" for every request.

The source code is in the Git repository at `https://github.com/RedHatTraining/DO328-apps/` in the **`maven-simplest`** directory.

3.1. Examine the **`application.yaml`** file, which describes the necessary resources to deploy the application. The deployment resource contains the **`sidecar.istio.io/inject: "true"`** annotation to inject the Envoy proxy.

Run the **`oc create -f application.yaml`** command to deploy the application.

```
[student@workstation traffic-deploy]$ oc create -f application.yaml
deployment.apps/hello created
service/hello created
```

The **`create`** command creates a deployment and a service for the application (both named **`hello`**). At this point the Vert.X application is only accessible from inside the mesh.

4. Create an ingress gateway to allow ingress traffic to the mesh.

4.1. Examine the **`gateway.yaml`** file, which describes the traffic allowed to enter the mesh.

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: hello-gateway(1)
spec:
  selector:
    istio: ingressgateway(2)
  servers:
    - port:(3)
        number: 80
        name: http
        protocol: HTTP
      hosts:(4)
        - "*"
```

1. Name assigned to the gateway configuration.
2. Indicates to which proxy gateway implementations the rules apply. In this case, it is the ingress gateway Envoy proxy.
3. Port and protocol where the gateway is listening for incoming connections.
4. Hosts exposed by this gateway; the **"*"** means that this field is not used to filter the incoming traffic.

Use the **oc create** command to create the ingress gateway configuration.

```
[student@workstation traffic-deploy]$ oc create -f gateway.yaml
gateway.networking.istio.io/hello-gateway created
```

5. Create a **VirtualService** to redirect the ingress traffic to the Vert.x application.

5.1. Examine the **virtual-service.yaml** file, which links the ingress traffic with the deployed Vert.X application.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: hello-vs (1)
spec:
  hosts:
  - "*"
  gateways:
  - hello-gateway (2)
  http: (3)
  - route: (4)
    - destination:
        host: hello (5)
        port: (6)
          number: 8080
```

1. Name assigned to the virtual service configuration.
2. List of gateways that should apply the routes. This virtual service is applied to the traffic configured by the **hello-gateway** gateway.
3. List of route rules for HTTP traffic.
4. Default route, as no match conditions are defined. This route redirects all traffic to the specified destination.
5. Destination rule which sends the traffic to the **hello** service.
6. Destination rule which sends the traffic to the port **8080** of the **hello** service.

Use the **oc create** command to create a virtual service.

```
[student@workstation traffic-deploy]$ oc create -f virtual-service.yaml
virtualservice.networking.istio.io/hello-vs created
```

6. Test the application.

6.1. Examine the **get-ingress-gateway-url.sh** script, which uses the **oc** command to gather the Istio ingress gateway URL.

Export the ingress gateway URL to an environment variable called **GATEWAY_URL**.

```
[student@workstation traffic-deploy]$ export  \
> GATEWAY_URL=$(sh get-ingress-gateway-url.sh)
```

6.2. Execute the **curl** command in combination with the **GATEWAY_URL** variable to confirm the access to the application from the terminal.

```
[student@workstation traffic-deploy]$ curl ${GATEWAY_URL}
Hello World!
```

7. Visualize the ingress traffic with Kiali

7.1. Examine the **get-kiali-url.sh** script, which uses the **oc** command to gather the Kiali URL.

Export the Kiali URL to an environment variable called **KIALI_URL**.

```
[student@workstation traffic-deploy]$ export KIALI_URL=$(sh get-kiali-url.sh)
```

7.2. Open the **KIALI_URL** URL in a browser to access Kiali.

```
[student@workstation traffic-deploy]$ firefox ${KIALI_URL}
```

7.3. Log in as the **developer** user.

7.4. Click **Graph** in the sidebar to visualize the traffic.

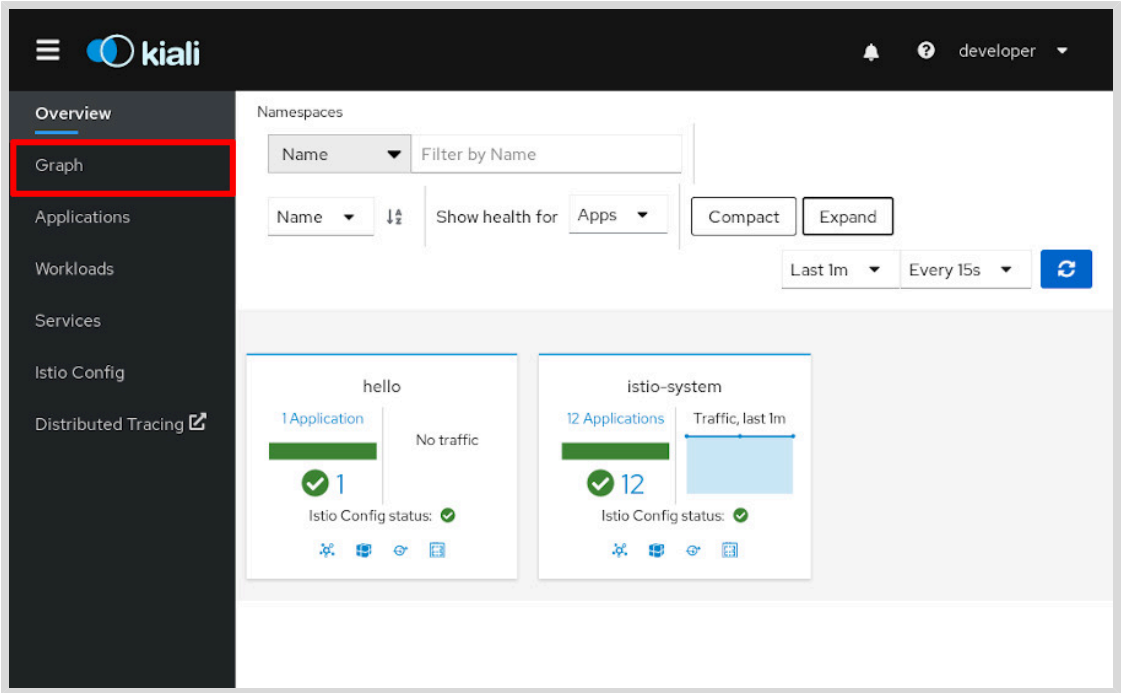


Figure 4.4: Kiali dashboard

7.5. From the **Graph** page, select the **hello namespace**.

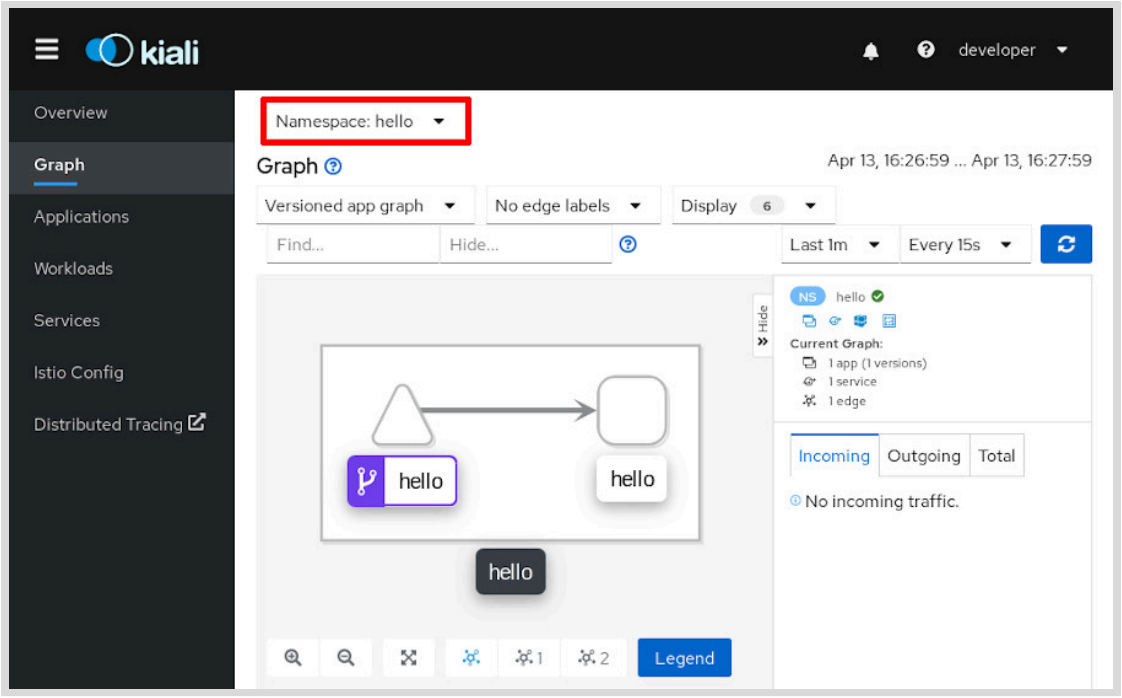


Figure 4.5: Graph representation of the hello application

7.6. Click the **Display** → **Traffic Animation** option to add motion to the graphics.

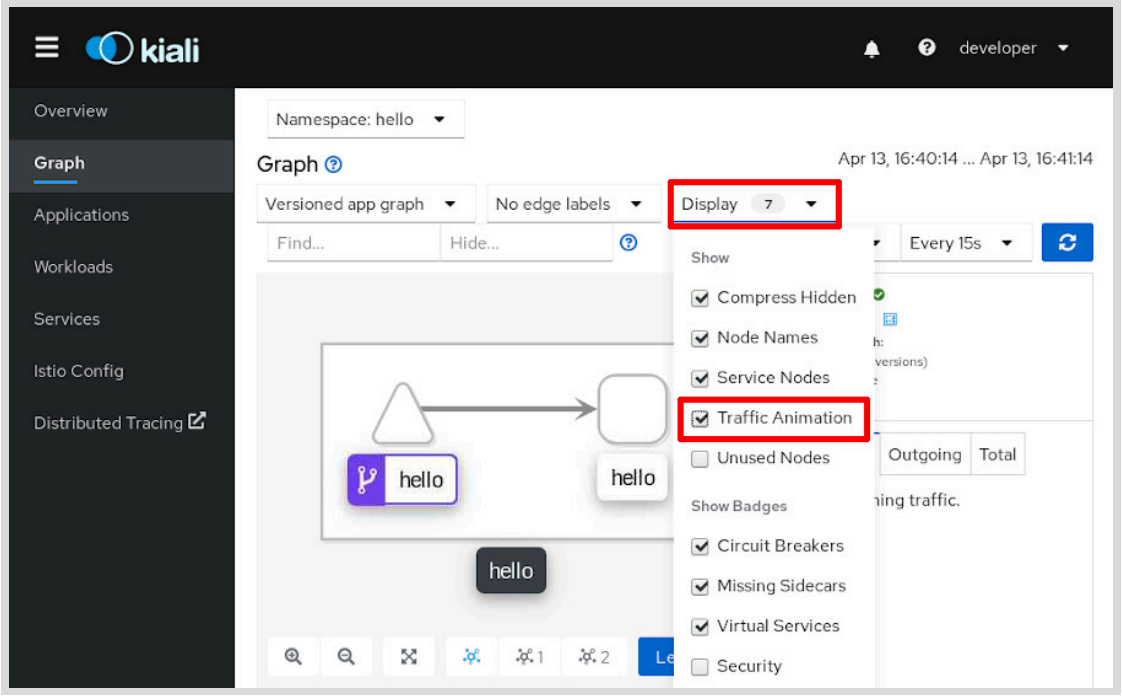


Figure 4.6: Adding animation to the Kiali graphics

7.7. Run the **`curl ${GATEWAY_URL}`** command multiple times in the background.

```
[student@workstation traffic-deploy]$ while true; \
> do curl -o /dev/null -s ${GATEWAY_URL}; \
> sleep 2; done
```

7.8. Observe the Kiali console to see how the traffic flows into the application. You might have to wait for a few seconds before you can see the changes on the Graph page in Kiali

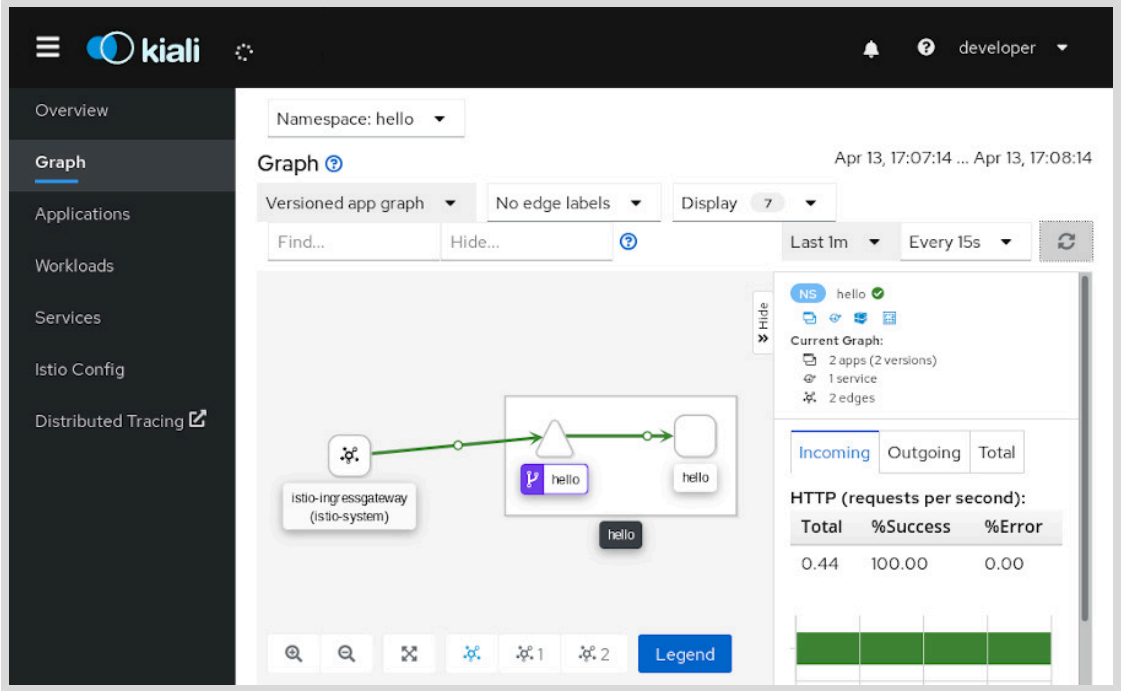


Figure 4.7: Traffic flow

7.9. Press **Ctrl**+**C** in the terminal to stop the **curl** command.

```
> sleep 2; done
^C
[student@workstation traffic-deploy]$
```

8. Return to the home directory.

```
[student@workstation traffic-deploy]$ cd ~
[student@workstation ~]$
```

Finish

On the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab traffic-deploy finish
```

This concludes the guided exercise.

Routing Traffic Based on Request Headers

Objectives

After completing this section, you should be able to route traffic to services in a mesh, based on request headers.

Describing Destination Rules

Destination rules are custom resources that define policies that apply to the traffic of a service. Using those traffic policies, you can configure the load balancing behavior to distribute traffic between the instances of a service.

Virtual services route traffic to a specific destination, and destination rules operate in the traffic routed to that destination.

The policies defined in destination rules are applied after the routing rules in the virtual services are evaluated. With destination rules you can define load balancing, connection limits, and outlier detection policies.

Load Balancing Traffic

With destination rules, you can specify the strategy used to distribute traffic between the instances of a service.

Round-robin
Requests are sent to each service instance in turn.

Random
Requests are sent to the service instances randomly.

Weighted
Request are sent to the service instances according to a specific weight (percentage).

Least request
Requests are sent to the least busy service instances.

Note
When no balancing option is specified, OpenShift Service Mesh uses the round-robin strategy.

Example of a destination rule that uses random load balancing strategy:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: my-destination-rule (1)
spec:
  host: my-svc (2)
```

```
  trafficPolicy: (3)
    loadBalancer:
      simple: RANDOM (4)
```

(1) Name of the destination rule.
(2) Service affected by the defined policies.
(3) Traffic policy defined for the **my-svc** service.
(4) Random load balancing strategy for the traffic sent to the **my-svc** service.

Splitting Services into Subsets

A service can have variants of the application running concurrently with destination rules. You can group those variants into *subsets* using Kubernetes tags.

When you have subsets, destination rules allow you to define a global traffic policy for the service and override the policy on the subsets.

Example of destination rule with subsets and policy overrides:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: my-destination-rule
spec:
  host: my-svc
  trafficPolicy: (1)
    loadBalancer:
      simple: RANDOM
  subsets: (2)
  - name: v1 (3)
    labels: (4)
      version: v1
  - name: v2
    labels:
      version: v2
    trafficPolicy: (5)
      loadBalancer:
        simple: ROUND_ROBIN
```

(1) Traffic policy defined at a service level.
(2) List of subsets defined for the service.
(3) Name of the subset.
(4) List of label tags used to select the service instances belonging to the subset.
(5) Traffic policy set at the subset level, overriding the policy set at the service level.

Routing Traffic

OpenShift Service Mesh traffic management relies on virtual services and destination rules. After combining these custom resources, you can perform A/B testing or route traffic to a specific version of a service.

Route traffic based on request headers with the following steps:

- Deploy different services or different versions of the same service.

- Create destination rules to split the service into subsets.
- Create a virtual service to check the request headers and route the request to a destination service or to a subset.

Creating Routing Rules

A virtual service is a compilation of conditions and actions that you can use to route HTTP, TCP, and unterminated TLS traffic to a desired destination.

Virtual services in combination with destination rules allow you to route traffic based on request headers. The following conditions and actions are involved:

HTTPRoute
Conditions and actions defined for HTTP/1.1, HTTP2, and gRPC traffic.

HTTPMatchRequest
List of match conditions to meet in order to execute the action defined by the rule.

HTTPRouteDestination
Action that routes the traffic to a desired destination.

StringMatch
Rule to compare a string against a value. The available options to do the comparison are: **`exact`**, **`prefix`**, and **`regex`**.

Destination
Destination for traffic that matches specified conditions.

Example of a virtual service with routing based on request headers:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: my-virtual-service
spec:
  hosts:
    - "*"
  http: (1)
    - match: (2)
        - headers: (3)
          end-user: (4)
            exact: redhatter (5)
      route: (6)
        - destination: (7)
            host: my-svc
            subset: v2
    - route: (8)
        - destination:
            host: my-svc
            subset: v1
```

(1) List of **`HTTPRoute`** conditions and actions.
(2) List of **`HTTPMatchRequest`** conditions and actions.
(3) Header rule.
(4) Name of the HTTP header to check.

5. **`StringMatch`** rule for the HTTP header. The match condition is activated when the request has an HTTP header called **`end-user`** with the value **`redhatter`**.
6. **`HTTPRouteDestination`**. When the match condition is activated, the request is redirected to this route.
7. **`Destination`**. When the headers condition is satisfied, the traffic is redirected to the **`v2`** subset of the **`my-svc`** service.
8. Default route. Without a previous matching, the traffic is routed to this destination.

The previous example routes all the traffic that has the HTTP header **`end-user`**, with a value of **`redhatter`**, to the subset **`v2`** of the **`my-svc`** service. When the HTTP header does not match the defined values, the traffic goes to the subset **`v1`** of the **`my-svc`** service.

Redirecting traffic to an endpoint based on the presence of a specific HTTP header is often used for A/B testing and for HTTP authorization.

References

Destination Rule reference
https://archive.istio.io/v1.4/docs/reference/config/networking/destination-rule/
in Istio documentation.

Virtual Service reference
https://archive.istio.io/v1.4/docs/reference/config/networking/virtual-service/
in Istio documentation.

Guided Exercise

Routing Traffic Based on Request Headers

In this exercise, you will route traffic to services in Red Hat OpenShift Service Mesh based on request headers.

Outcomes

You should be able to create subsets of a service, and route traffic to services in a mesh, based on request headers.

Before You Begin

To perform this exercise, ensure you have access to:

- A configured and running OpenShift cluster.
- A installed and running OpenShift Service Mesh in the OpenShift cluster.
- The OpenShift CLI (**/usr/local/bin/oc**).

As the **student** user on the **workstation** machine, use the **lab** command to validate the prerequisites for this exercise:

```
[student@workstation ~]$ lab traffic-route start
```

1. Log in to the OpenShift cluster.

   1.1. Source the classroom configuration file that is accessible at **/usr/local/etc/ocp4.config**.

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

   1.2. Log in to OpenShift as the **developer** user.

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_API}
Login successful.
...output omitted...
```

2. Create the **headers** project, and then add it to the **ServiceMeshMemberRoll** resource.

   2.1. The required YAML files and scripts for this guided exercise are located in the **/home/student/DO328/labs/traffic-route** directory. Change to this directory using the **cd** command.

```
[student@workstation ~]$ cd /home/student/DO328/labs/traffic-route
[student@workstation traffic-route]$
```

2.2. Create the **headers** project.

```
[student@workstation traffic-route]$ oc new-project headers
Now using project "headers" on server
"https://api.cluster.domain.example.com:6443".
...output omitted...
```

2.3. Add the **headers** project to the list of members in the **ServiceMeshMemberRoll** resource. The **ServiceMeshMemberRoll** is available in the **istio-system** project.

Use the **add-project-to-smmr.sh** script to add the **headers** project to the list of members in the **ServiceMeshMemberRoll** resource.

```
[student@workstation traffic-route]$ sh add-project-to-smmr.sh
servicemeshmemberroll.maistra.io/default patched
```

3. Deploy a simple Vert.X application with two versions of the same service, and with **/headers** as the URI prefix. This application exposes a single GET endpoint that returns:

- "Hello World!" on version **v1**.
- "Hello Red Hat!" on version **v2**.

The source code is in the Git repository at `https://github.com/RedHatTraining/DO328-apps/` in the **maven-simplest** and **maven-simplest-v2** directories.

3.1. Examine the **application.yaml** file, which describes the necessary resources to deploy the application. This file defines two different deploys of an application, a service, a virtual service, and a gateway.

Run the **oc apply -f application.yaml** command to deploy the application.

```
[student@workstation traffic-route]$ oc apply -f application.yaml
deployment.apps/headers-v1 created
deployment.apps/headers-v2 created
service/headers created
gateway.networking.istio.io/headers-gateway created
virtualservice.networking.istio.io/headers-vs created
```

Now the Vert.X application is deployed in the service mesh and is accessible from outside the mesh.

3.2. Examine the **get-ingress-gateway-url.sh** script, which uses the **oc** command to gather the Istio ingress gateway URL.

Export the ingress gateway URL to an environment variable called **GATEWAY_URL**.

```
[student@workstation traffic-route]$ export  \
> GATEWAY_URL=$(sh get-ingress-gateway-url.sh)
```

3.3. Execute the **`curl`** command several times, in combination with the **`GATEWAY_URL`** variable, to confirm access to the application from the terminal.

```
[student@workstation traffic-route]$ while true; \
> do curl -s ${GATEWAY_URL}; \
> sleep 1; done
Hello World!
Hello Red Hat!
Hello Red Hat!
Hello World!
...output omitted...
```

When you have variants of a service without destination rules defined, Kubernetes balances the traffic between service instances randomly.

3.4. Press **`Ctrl`**+**`C`** in the terminal to stop the **`curl`** command.

```
...output omitted...
Hello World!
^C
[student@workstation traffic-route]$
```

4. Route all traffic to the **`v1`** subset of the application.

4.1. Examine the **`destination-rule.yaml`** file, which defines the subsets of the Vert.X application.

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: headers
spec:
  host: headers(1)
  subsets:
  - name: v1(2)
    labels:
      version: v1(3)
  - name: v2
    labels:
      version: v2
```

(1) Service affected by the defined policies.
(2) Name assigned to the subset.
(3) Labels used to make the subset grouping.

The **`destination-rule.yaml`** file defines two subsets for the **`headers`** service:

- A subset called **`v1`** grouping all the application instances that have the **`version: v1`** flag.
- A subset called **`v2`** grouping all the application instances that have the **`version: v2`** flag.

Use the **`oc create`** command to create a destination rule.

```
[student@workstation traffic-route]$ oc create -f destination-rule.yaml
destinationrule.networking.istio.io/headers created
```

4.2. Examine the **virtual-service-subset-v1.yaml** file, which uses destination rules subsets to route traffic.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: headers-vs
spec:
  hosts:
    - "*"
  gateways:
    - headers-gateway
  http:
    - match:
        - uri:(1)
            prefix: /headers
      route:(2)
        - destination:
            host: headers(3)
            subset: v1(4)
```

1. Rule to accept only traffic with the **/headers** prefix in the URI.
2. Route assigned to all ingress traffic with the **/headers** prefix in the URI.
3. Destination service.
4. Destination service subset.

The virtual service defined in **virtual-service-subset-v1.yaml** redirects all the ingress traffic for **/headers** to the subset **v1** of the **headers** service.

Use the **oc apply** command to update the **headers-vs** virtual service with the new configuration.

```
[student@workstation traffic-route]$ oc apply -f virtual-service-subset-v1.yaml
virtualservice.networking.istio.io/headers-vs configured
```

4.3. Execute the **curl** command several times, in combination with the **GATEWAY_URL** variable, to confirm the routing of all traffic to the **v1** subset.

```
[student@workstation traffic-route]$ while true; \
> do curl -s ${GATEWAY_URL}; \
> sleep 1; done
Hello World!
Hello World!
Hello World!
Hello World!
...output omitted...
```

4.4. Press **Ctrl**+**C** in the terminal to stop the **curl** command.

```
...output omitted...
Hello World!
^C
[student@workstation traffic-route]$
```

5. Route traffic based on headers.

5.1. Examine the **`virtual-service-with-header-subsets.yaml`** file, which uses destination rules subsets in combination with header rules to route traffic.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: headers-vs
spec:
  hosts:
    - "*"
  gateways:
    - headers-gateway
  http:
    - match:(1)
        - uri:
            prefix: /headers
          headers:
            end-user:
              exact: redhatter
      route:
        - destination:(2)
            host: headers
            subset: v2
    - match:
        - uri:
            prefix: /headers
      route:(3)
        - destination:
            host: headers
            subset: v1
```

(1) Match condition for HTTP traffic. This rule is activated when the request URI has the **`/headers`** prefix, and an HTTP header called **`end-user`** with **`redhatter`** as value.

(2) Route destination assigned to the match condition. When the match condition is activated, the request is redirected to the **`v2`** subset of the **`headers`** service.

(3) When no previous match conditions are activated for traffic to **`/headers`**, the request is redirected to the **`v1`** subset of the **`headers`** service.

Use the **`oc apply`** command to update the **`headers-vs`** virtual service with the new configuration.

```
[student@workstation traffic-route]$ oc apply \
> -f virtual-service-with-header-subsets.yaml
virtualservice.networking.istio.io/headers-vs configured
```

5.2. Execute the **curl** command in combination with the **GATEWAY_URL** variable.

```
[student@workstation traffic-route]$ curl ${GATEWAY_URL}
Hello World!
```

Without specifying any extra HTTP headers, the request is redirected to the **v1** subset, returning "Hello World!"

5.3. Execute the **curl** command in combination with the **GATEWAY_URL** variable and send the **end-user** HTTP header.

```
[student@workstation traffic-route]$ curl -H "end-user: redhatter" ${GATEWAY_URL}
Hello Red Hat!
```

When the **end-user: redhatter** HTTP header is added to the request, the service mesh redirects the request to the **v2** subset.

6. Visualize traffic routing with Kiali.

6.1. Examine the **get-kiali-url.sh** script, which uses the **oc** command to gather the Kiali URL.

Export the Kiali URL to an environment variable called **KIALI_URL**.

```
[student@workstation traffic-route]$ export KIALI_URL=$(sh get-kiali-url.sh)
```

6.2. Execute the **firefox** command in combination with the **KIALI_URL** variable to access Kiali.

```
[student@workstation traffic-deploy]$ firefox ${KIALI_URL}
```

6.3. Log in as the **developer** user.

6.4. Click **Graph** to visualize the traffic.

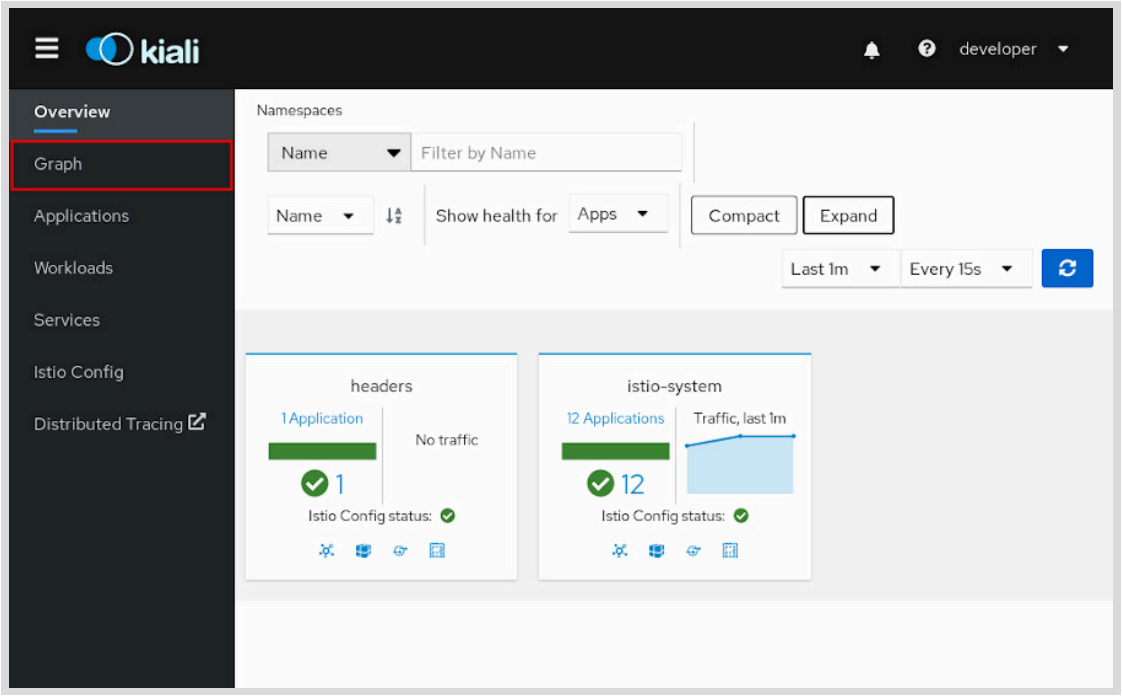


Figure 4.8: Kiali dashboard

6.5. On the **Graph** page, select the **`headers` namespace**.

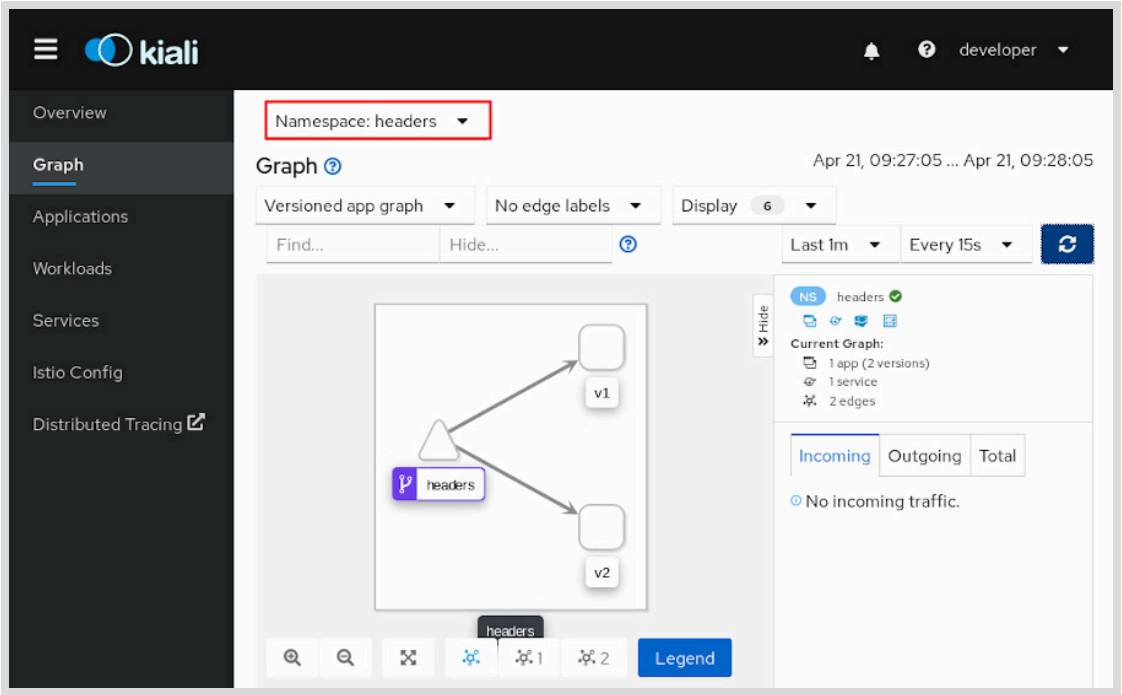


Figure 4.9: Graph representation of the headers application

6.6. Click the **Display** → **Traffic Animation** option to add motion to the graphics.

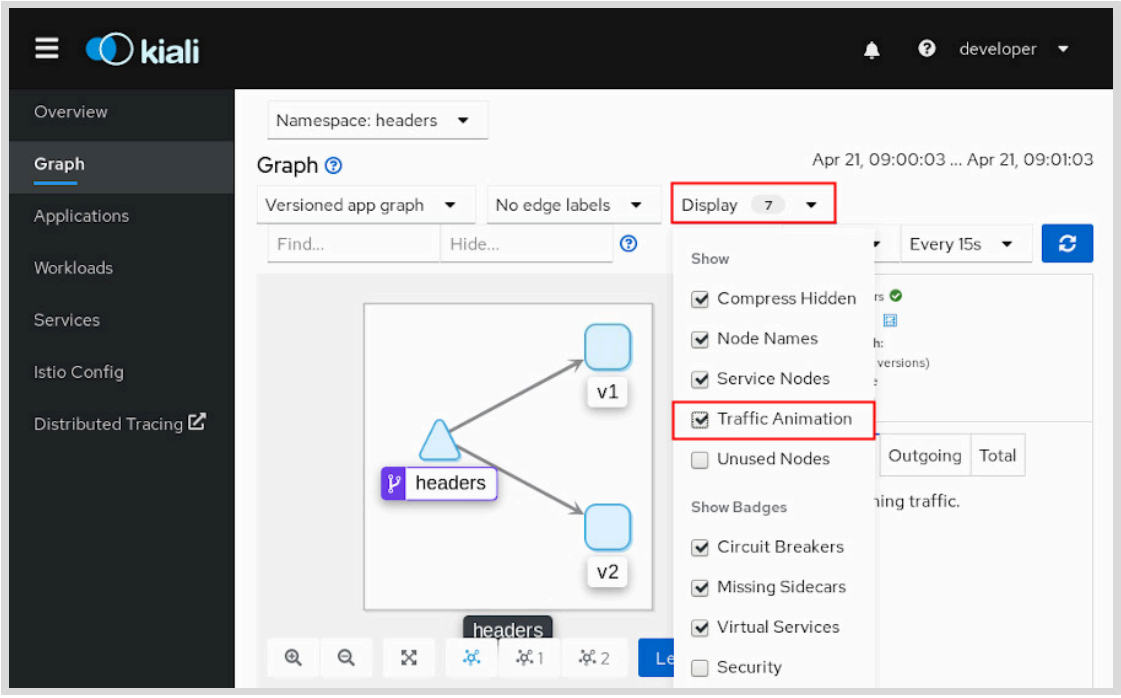


Figure 4.10: Adding animation to the Kiali graphics

6.7. Examine the **`traffic-simulator.sh`** script. This script simulates traffic to the deployed application, making **`curl`** calls to the ingress URL.

Run the **`traffic-simulator.sh`** script to generate some traffic.

```
[student@workstation traffic-deploy]$ sh traffic-simulator.sh
```

6.8. Observe the Kiali console to see how the traffic flows into the application to the two service subsets. You may have to wait for a few seconds until you can see the changes in the Graph page in Kiali.

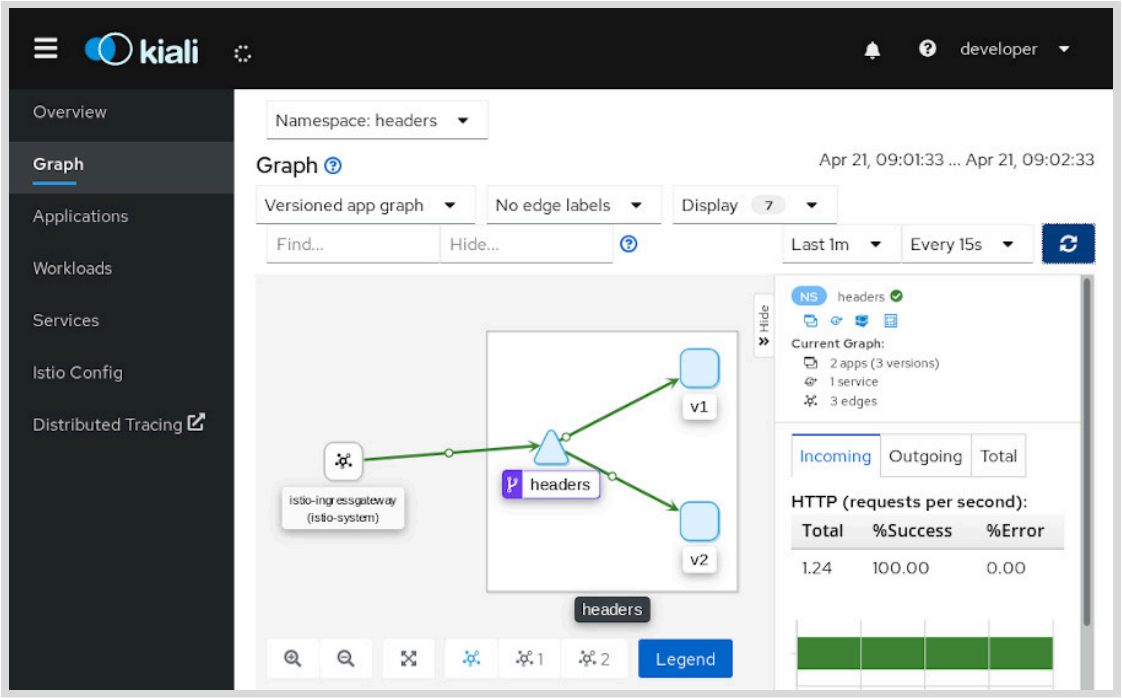


Figure 4.11: Routed traffic flow

6.9. Press **Ctrl+C** in the terminal to stop the **traffic-simulator.sh** script.

```
[student@workstation traffic-route]$ sh traffic-simulator.sh
^C
[student@workstation traffic-route]$
```

7. Return to the home directory.

```
[student@workstation traffic-route]$ cd ~
[student@workstation ~]$
```

Finish

On the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab traffic-route finish
```

This concludes the guided exercise.

Accessing External Services

Objectives

After completing this section, you should be able to control egress traffic to access external services.

Managing and Routing Egress Traffic in OpenShift Service Mesh

Preceding sections introduced the capabilities of OpenShift Service Mesh to route between services inside the mesh. It is common, however, for applications deployed in an OpenShift project to require access to services in different projects, or even outside the OpenShift cluster. This section describes how OpenShift Service Mesh enables the management and routing of service requests outside the mesh. Traffic originating from services inside the mesh and targeting external services is called *egress traffic*.

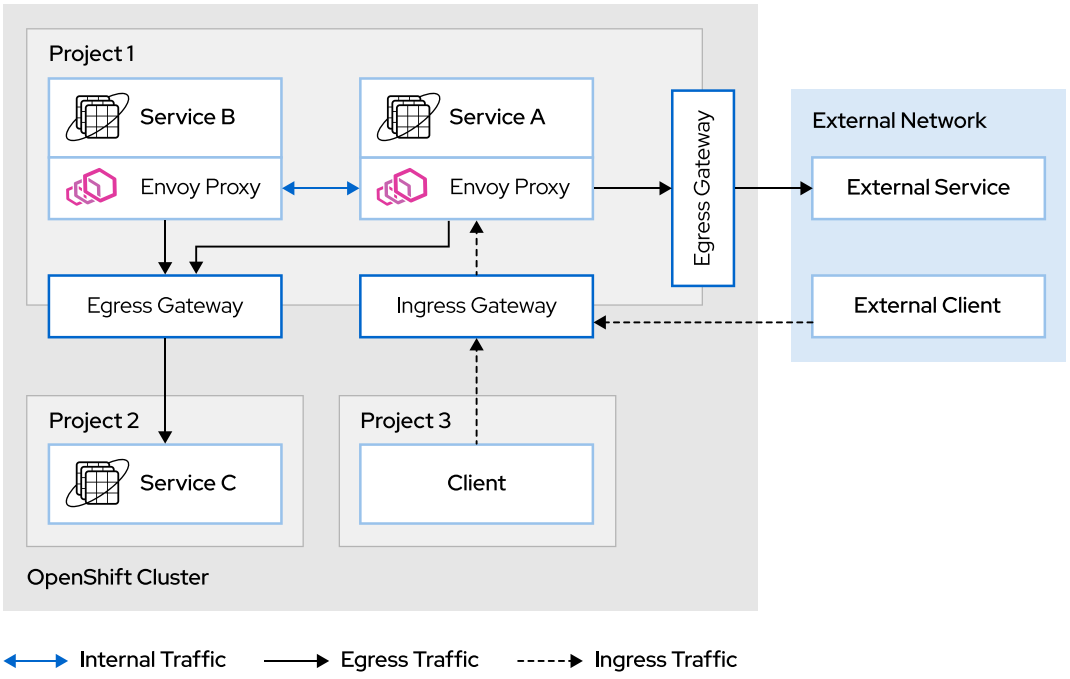


Figure 4.12: Ingress and Egress Traffic in Red Hat OpenShift Service Mesh

There are two elements related to egress traffic in OpenShift Service Mesh. The Istio control plane includes egress gateways, configuring these gateways to allow all egress traffic, or to restrict egress traffic to registered services. **`ServiceEntry`** resources register external services that are requested by internal services.

Configuring Egress Traffic Configuration in Istio

By default, OpenShift Service Mesh allows all egress traffic. If a service invokes another service not managed by OpenShift Service Mesh, the Envoy proxy redirects the requests to the default

Istio gateway. By default, this egress gateway forwards the requests to the external network, allowing all external requests to be serviced.

This setup allows all services in the mesh to reach any external service without restrictions. In some cases, it is beneficial to restrict the external services allowed to a specific list of approved services. The **`spec.istio.global.outboundTrafficPolicy.mode`** configuration value in the **`ServiceMeshControlPlane`** resource controls this behavior.

The default value for this entry is **`ALLOW_ANY`**. This value instructs Istio to allow all egress traffic regardless of the destination. If this configuration holds the value **`REGISTRY_ONLY`**, the gateway only forwards requests to services explicitly registered.

Registering External Services

The **`REGISTRY_ONLY`** configuration value restricts traffic to registered external services only. To register external services, create a **`ServiceEntry`** resource associated to the external service, as follows:

```
[user@host ~]$ oc create -f - <<'EOF'
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: my-external-service (1)
spec:
  hosts:
  - example.external.com (2)
  ports:
  - number: 80
    name: http
    protocol: HTTP
  resolution: DNS(3)
  location: MESH_EXTERNAL(4)
  EOF
```

1. Give each service a meaningful name for easy identification.
2. The hostname where the external service is exposed.
3. The actual IP of the service must be resolved via DNS by the proxy.
4. **`MESH_EXTERNAL`** indicates that the service is external to the mesh.

This **`ServiceEntry`** configures Istio to allow egress traffic to `example.external.com:80` from any service in the mesh.

Enabling Direct Access to External Services

Sometimes, it is necessary to access external services bypassing the Envoy proxy, such as for extreme performance requirements or strict immutability of the requests. You can configure Istio to denylist or allowlist ranges of IP addresses for the proxy to intercept. The Envoy proxy intercepts all IPs belonging to any range in the **`global.proxy.includeIPRanges`** configuration entry. Istio traffic management policies handle all requests to those IPs. The Envoy proxy *does not* intercept any request targeting an IP belonging to a range in the **`global.proxy.excludeIPRanges`** configuration entry. Those requests bypass Istio policies and monitoring.

To update the configuration, edit the **`ServiceMeshControlPlane`** resource or use an **`oc patch`** command similar to:

```
[user@host ~]$ oc patch smcp basic-install -n istio-system --type merge \
> -p '{"spec":{"istio":{"global":{"proxy":{"includeIPRanges":"10.0.0.1/24"}}}}}'
```

Note that this configuration is global for the whole Istio installation, and affects all traffic in all meshes managed by Istio. To apply this same behavior for specific pods, add the **`traffic.sidecar.istio.io/excludeOutboundPorts`** or **`traffic.sidecar.istio.io/includeOutboundIPRanges`** annotations in the **`Pod`** resource.

```
kind: Pod
apiVersion: v1
metadata:
  name: application_pod
  annotations:
    sidecar.istio.io/inject: 'false'
    traffic.sidecar.istio.io/includeOutboundIPRanges: '10.0.0.1/24'
    ...output omitted...
  namespace: application_project
  ...output omitted...
spec:
  ...output omitted...
```

References

Accessing External Services

https://archive.istio.io/v1.4/docs/tasks/traffic-management/egress/egress-control/

in Istio documentation.

Guided Exercise

Accessing External Services

In this exercise, you will connect an application with OpenShift with an external service, deployed in a namespace not managed by Istio. Those same steps are also valid for connecting with a service deployed outside the OpenShift cluster.

Outcomes

You should be able to enable egress connections from the mesh, either globally for all services or for a single external service.

Before You Begin

As the **student** user on the **workstation** machine, use the **lab** command to prepare your system for this exercise.

This command deploys the Financial application and the News service in two separate projects in the Red Hat OpenShift cluster. The command also includes the Financial application into the Red Hat OpenShift Service Mesh.

```
[student@workstation ~]$ lab traffic-external start
```

1. Review the installed applications and acknowledge the correct functioning.

1.1. Log in to the Red Hat OpenShift cluster using the **developer** account.

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
[student@workstation ~]$ oc login \
> -u ${RHT_OCP4_DEV_USER} -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_API}
Login successful.
...output omitted...
```

1.2. Retrieve the URL of the Istio ingress gateway by running the following command:

```
[student@workstation ~]$ ISTIO_GW=$(oc get route istio-ingressgateway \
> -n istio-system -o jsonpath="{.spec.host}{.spec.path}")
[student@workstation ~]$ echo $ISTIO_GW
istio-ingressgateway-istio-system.apps.ocp4.example.com
```

1.3. Open a web browser to verify the application is functioning correctly. The Financial application is accessed through the ingress gateway URL as just retrieved, with the **/frontend** path appended. You can use your favorite browser to open that URL, or use the following command to open the URL in Firefox:

```
[student@workstation ~]$ firefox $ISTIO_GW/frontend &
```

1.4. Navigate through the application to generate traffic between services. Make sure you gather historical data, exchange data, and news. Note that the News service resides in a different namespace, **`traffic-external-news`**, and hence is considered egress traffic. Gathering news successfully proves that egress traffic is allowed.

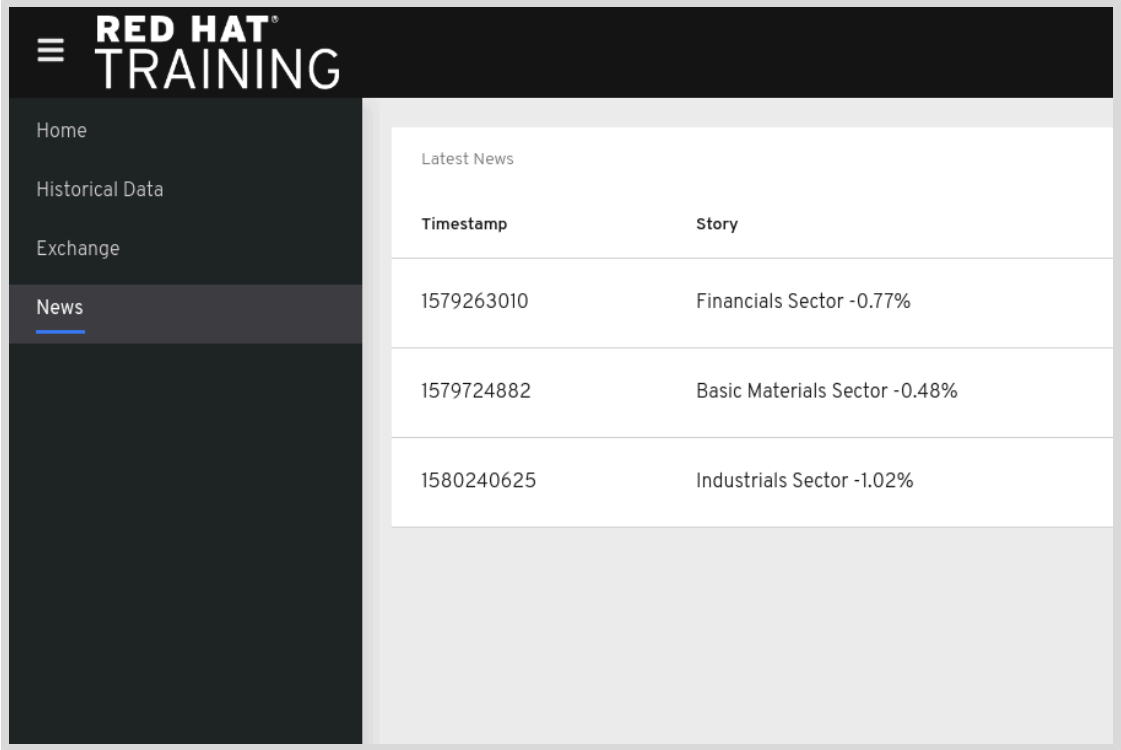


Figure 4.13: News Section in the Finance Application

1.5. Observe the application graph Kiali generates for the Financial application. You can find the Kiali URL in the Red Hat OpenShift console, or use the following command:

```
[student@workstation ~]$ oc get route kiali \
> -n istio-system -o jsonpath="https://{.spec.host}{.spec.path}"
https://kiali-istio-system.apps.ocp4.example.com
```

Open the resulting URL, and log in user the **`developer`** credentials. Go to the **Graph** menu and review the **`Service graph`** for the **`traffic-external`** namespace. The graph should be similar to the following one:

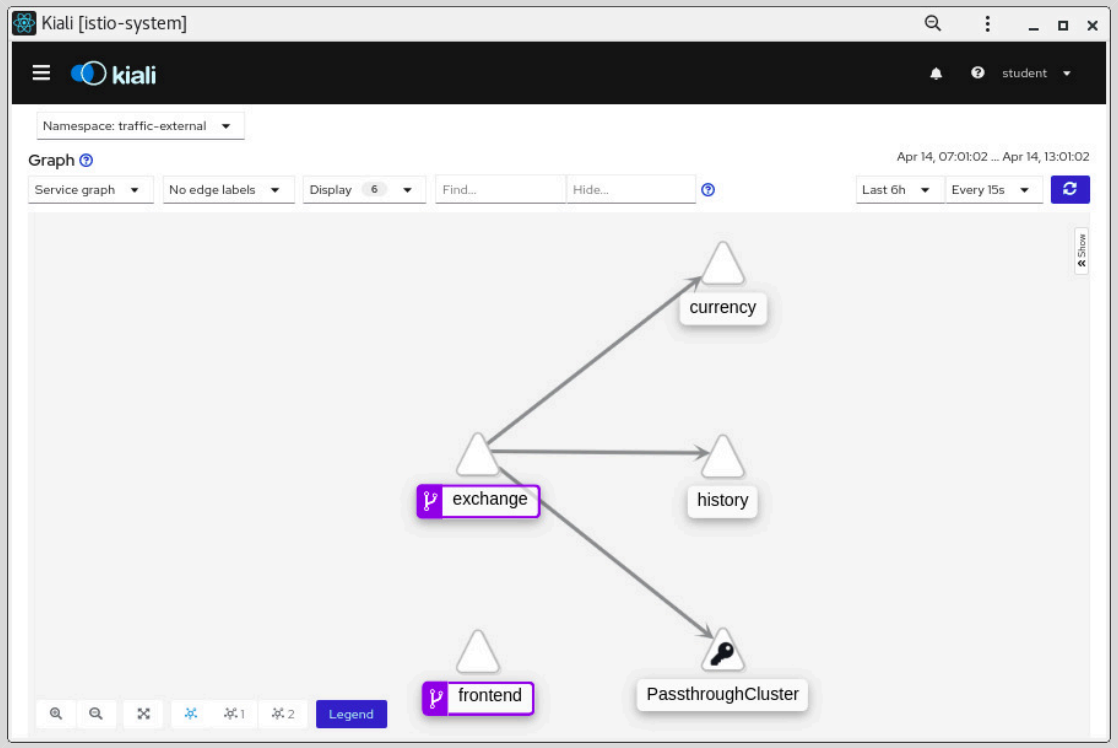


Figure 4.14: Service Graph for the Finance Application

For details about how to log in and navigate through Kiali interface refer to the *Observing Service Interactions with Kiali* section in the *Chapter 3, Observing a Service Mesh* chapter.

Note that the News service does not appear in the graph. Instead, a generic **`PassthroughCluster`** restricted namespace shows. This generic entry proves that the service mesh is not managing external traffic to the News service.

2. Restrict egress traffic globally to registered services only.

2.1. Update Istio configuration and define outbound traffic policy to allow egress traffic only to registered services. To do so, edit the **`basic-install`** control plane, and set the **`global.outboundTrafficPolicy.mode`** entry to **`REGISTRY_ONLY`**.

```
[student@workstation ~]$ oc patch smcp basic-install \
> --type merge -n istio-system \
> -p '{"spec":{"istio":{"global":{"outboundTrafficPolicy":
{"mode":"REGISTRY_ONLY"}}}}}'
servicemeshcontrolplane.maistra.io/basic-install patched
```

2.2. Validate the external News service is not available anymore. Get back to the Financial application front end in your browser and navigate to the **News** section in the left menu. The application shows that no news can be loaded.

Alternatively, you can check how Kiali detects the failing connection and updates the service graph:

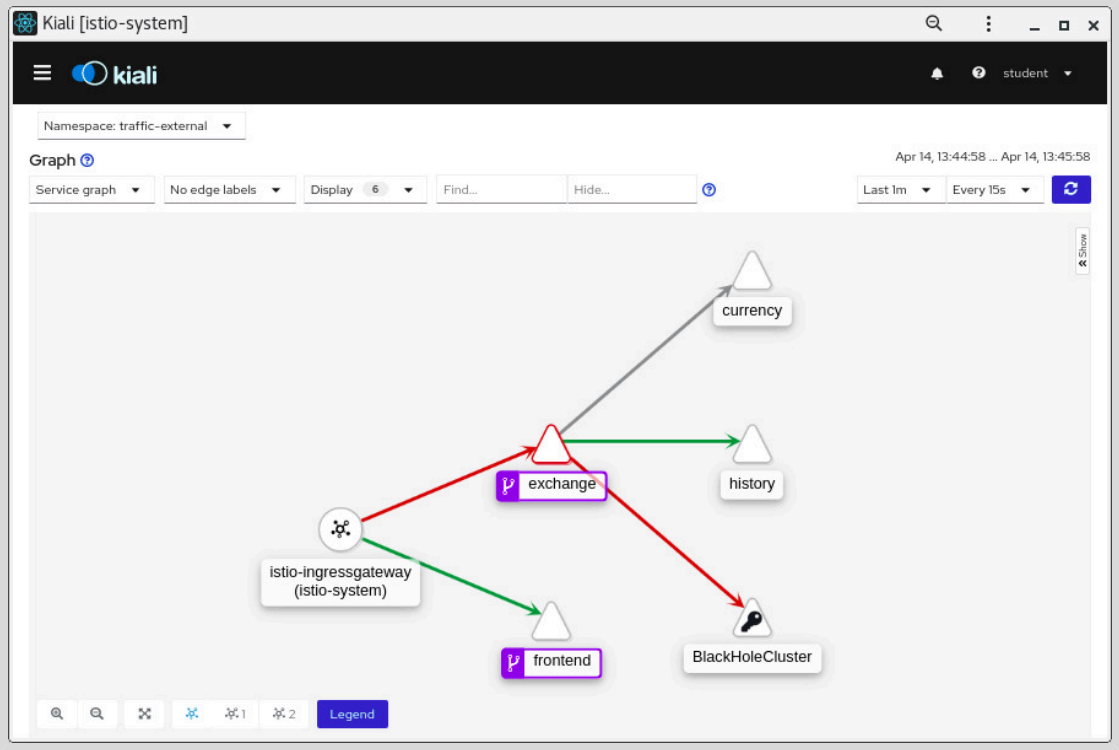


Figure 4.15: Service Graph Displaying Blocked Egress Traffic

Note

Browser cache may cause the application to still display the news on the page. Force reload the web page to instruct the browser to obtain the new page, or use browser instructions to delete cache, and reload the page.

The Service graph in Kiali may also require some time to update. Some arrows may look different, depending on the time span selected and traffic analyzed.

3. Create a **`ServiceEntry`** resource for the News service. The presence of that resource includes the News service in the registry of allowed targets for egress traffic.

3.1. Retrieve the host where the external News service is available, either through the OpenShift console or through CLI commands:

- Log in to the OpenShift console and select to the **`traffic-external-news`** project. Within this project, navigate to the **Networking** → **Routes** menu entry. The only route available shows the host used to publish the service.

- Use the **`oc get route`** command to retrieve the host defined in the **news** route in the **`traffic-external-news`** project:

```
[student@workstation ~]$ NEWS_HOST=$(oc get route news -n traffic-external-news \
> -o jsonpath="{.spec.host}")
```

3.2. Create a **`ServiceEntry`** resource in the **`traffic-external`** namespace. You can use the file **`~/DO328/solutions/traffic-external/`**

`news_serviceentry.yml` as a template, and replace the host with the one obtained in the previous step.

```
[student@workstation ~]$ cd ~/DO328/solutions/traffic-external/
[student@workstation traffic-external]$ sed -e "s/_NEWS_HOST_/$NEWS_HOST/g" \
> news_serviceentry.yml | oc create -n traffic-external -f -
serviceentry.networking.istio.io/news created
[student@workstation traffic.external]$ cd ~
[student@workstation ~]$
```

3.3. Validate now the external News service is allowed. Get back to the Financial application front end in your browser and navigate to the **News** section in the left menu. The application shows that news are again displayed.

After reviewing the application, you can also review how Kiali is now displaying the News service as a managed service:

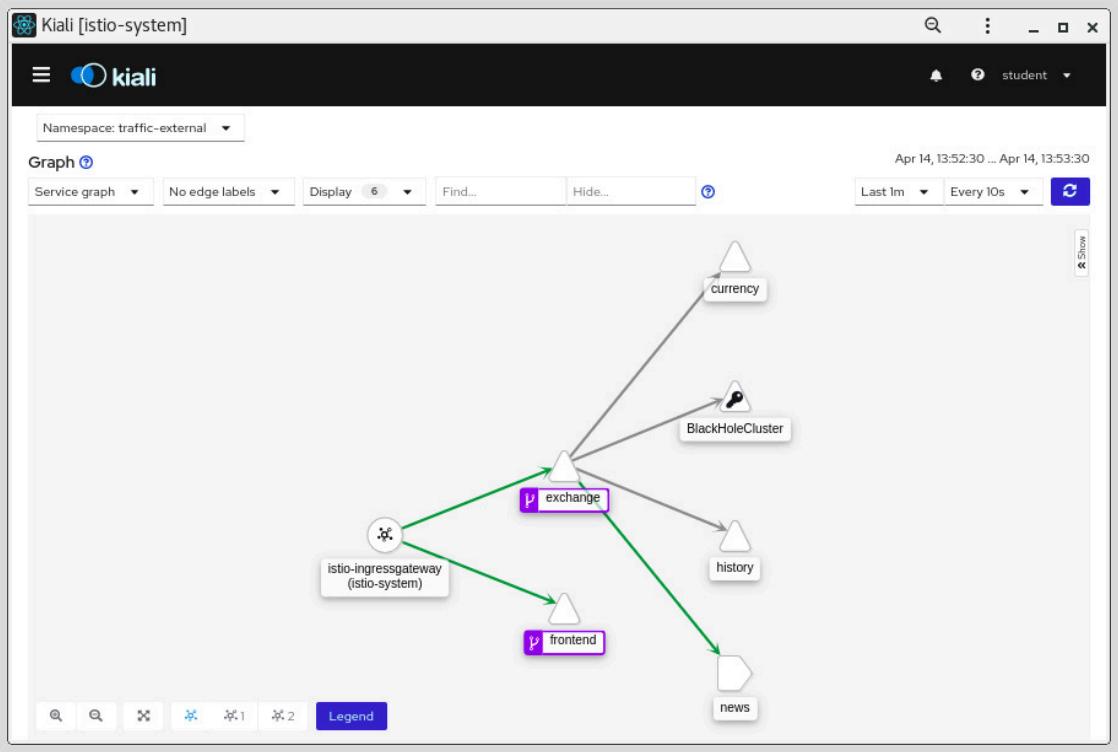


Figure 4.16: Service Graph Displaying Allowed Egress Traffic

Finish

On the **`workstation`** machine, use the **`lab`** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab traffic-external finish
```

This concludes the guided exercise.

Lab

Controlling Service Traffic

Performance Checklist

In this lab, you will deploy multiple versions of a service in the service mesh, route traffic based on request headers, and restrict the egress traffic.

Outcomes

You should be able to deploy applications on Red Hat OpenShift Service Mesh, route traffic inside the mesh and restrict egress traffic.

Before You Begin

To perform this exercise, ensure you have access to:

- A configured and running OpenShift cluster.
- A installed and running OpenShift Service Mesh in the OpenShift cluster.
- The OpenShift CLI (**`/usr/local/bin/oc`**).

As the **`student`** user on the **`workstation`** machine, use the **`lab`** command to prepare your system for this lab.

This command deploys the Financial application, the News service, and the AB Proxy in separate projects in the Red Hat OpenShift cluster. The command also includes the Financial application in the Red Hat OpenShift Service Mesh.

```
[student@workstation ~]$ lab traffic-mesh start
```

The following project information is needed for this exercise. This lab uses two projects:

- **`traffic-mesh`**: financial application composed of multiple services. This project is used to deploy a variation of the front end service and to set routes based on request headers.
- **`traffic-mesh-news`**: project deployed outside of the service mesh to provide a news feed for the financial application. This project is used to configure restrictions to the egress traffic in OpenShift Service Mesh.

Variations of the front end service that you will use in this lab:

- **`v2`**: initial deployment of the front end service that has a dark header.
- **`beta`**: new version of the front end service that has a red header.

To help with the testing, the **`traffic-mesh-proxy`** project is deployed and ready to use in the cluster. This project contains an application that works as a proxy for the main application, adding custom HTTP headers to the requests.

To configure the behavior of the proxy, the following helper scripts are available:

- **`proxy-set-beta-config.sh`**: configures the proxy to add the **`version:beta`** header to all requests.

- **`proxy-set-v2-config.sh`**: configures the proxy to add the **`version:v2`** header to all requests.

1. The companion scripts for this lab are located in **`~/DO328/labs/traffic-mesh`**. Change to that directory using the **`cd`** command.

```
[student@workstation ~]$ cd ~/DO328/labs/traffic-mesh
[student@workstation traffic-mesh]$
```

2. Log in to the OpenShift cluster as a unprivileged user and verify that the lab projects are successfully deployed.
3. Configure an ingress gateway named **`traffic-mesh-gateway`** to allow ingress HTTP traffic on port **`80`** to enter the mesh.
4. Deploy a new **`beta`** version of the **`frontend`** service with the following characteristics:
   - **Name**: **`frontend-beta`**.
   - Use the **`quay.io/redhattraining/ossm-frontend:beta`** image and port **`3000`**.
   - The container needs a environment variable called **`REACT_APP_GW_ENDPOINT`**. This variable gathers the value for the key **`GW_ADDR`** stored in the **`frontend-cm`** config map.
5. Split the **`frontend`** service into subsets with the following characteristics:
   - **Name**: **`frontend-destination-rule`**.
   - Create a group with all the instances that have the **`version: v2`** tag and assign to this group the name **`v2`**.
   - Create a group with all the instances that have the **`version: beta`** tag and assign to this group the name **`beta`**.
6. Create a virtual service named **`frontend-vs`**, which uses the ingress gateway and redirects the **`frontend`** traffic following this rules:
   - Traffic with a header matching **`version: beta`** is redirected to the **`beta`** subset.
   - Traffic with wrong or missing headers is redirected to the **`v2`** subset.

Note
All traffic for the front end service uses the **`/frontend`** prefix.

7. Test the routing configuration using the **`traffic-mesh-proxy`** application.
8. Restrict egress traffic globally to registered services only.
9. Test the restricted egress policy using the **`traffic-mesh-proxy`** application.
10. Allow egress traffic for the **`news`** service. Assign the name **`news-se`** to the required custom resource.
11. Test the egress traffic using the **`traffic-mesh-proxy`** application.
12. Return to the home directory.

```
[student@workstation traffic-mesh]$ cd ~
[student@workstation ~]$
```

Evaluation

Grade your work by running the **lab traffic-mesh grade** command from your **workstation** machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab traffic-mesh grade
```

Finish

On the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab traffic-mesh finish
```

This concludes the lab.

Solution

Controlling Service Traffic

Performance Checklist

In this lab, you will deploy multiple versions of a service in the service mesh, route traffic based on request headers, and restrict the egress traffic.

Outcomes

You should be able to deploy applications on Red Hat OpenShift Service Mesh, route traffic inside the mesh and restrict egress traffic.

Before You Begin

To perform this exercise, ensure you have access to:

- A configured and running OpenShift cluster.
- A installed and running OpenShift Service Mesh in the OpenShift cluster.
- The OpenShift CLI (**`/usr/local/bin/oc`**).

As the **`student`** user on the **`workstation`** machine, use the **`lab`** command to prepare your system for this lab.

This command deploys the Financial application, the News service, and the AB Proxy in separate projects in the Red Hat OpenShift cluster. The command also includes the Financial application in the Red Hat OpenShift Service Mesh.

```
[student@workstation ~]$ lab traffic-mesh start
```

The following project information is needed for this exercise. This lab uses two projects:

- **`traffic-mesh`**: financial application composed of multiple services. This project is used to deploy a variation of the front end service and to set routes based on request headers.
- **`traffic-mesh-news`**: project deployed outside of the service mesh to provide a news feed for the financial application. This project is used to configure restrictions to the egress traffic in OpenShift Service Mesh.

Variations of the front end service that you will use in this lab:

- **`v2`**: initial deployment of the front end service that has a dark header.
- **`beta`**: new version of the front end service that has a red header.

To help with the testing, the **`traffic-mesh-proxy`** project is deployed and ready to use in the cluster. This project contains an application that works as a proxy for the main application, adding custom HTTP headers to the requests.

To configure the behavior of the proxy, the following helper scripts are available:

- **`proxy-set-beta-config.sh`**: configures the proxy to add the **`version:beta`** header to all requests.

- **proxy-set-v2-config.sh**: configures the proxy to add the **version:v2** header to all requests.

1. The companion scripts for this lab are located in **~/DO328/labs/traffic-mesh**. Change to that directory using the **cd** command.

```
[student@workstation ~]$ cd ~/DO328/labs/traffic-mesh
[student@workstation traffic-mesh]$
```

2. Log in to the OpenShift cluster as a unprivileged user and verify that the lab projects are successfully deployed.

   2.1. Source the classroom configuration file that is accessible at **/usr/local/etc/ocp4.config**.

```
[student@workstation traffic-mesh]$ source /usr/local/etc/ocp4.config
```

   2.2. Log in to OpenShift as the **developer** user.

```
[student@workstation traffic-mesh]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_API}
Login successful.
...output omitted...
```

   2.3. Change to the **traffic-mesh** project.

```
[student@workstation traffic-mesh]$ oc project traffic-mesh
Now using project "traffic-mesh" on server ...
```

   2.4. Verify the status of the **traffic-mesh** project pods.

```
[student@workstation traffic-mesh]$ oc get pods
NAME                          READY   STATUS    RESTARTS   AGE
currency-v1-67bfdfd775-59f8v  2/2     Running   0          66s
exchange-v2-5d5c669777-s9zgk  2/2     Running   0          66s
frontend-v2-5cf6499bcd-hfp44  2/2     Running   0          66s
history-v1-8656f7d44f-ddp7c   2/2     Running   0          66s
```

   2.5. Verify the status of the **traffic-mesh-news** project pods.

```
[student@workstation traffic-mesh]$ oc get pods -n traffic-mesh-news
NAME                   READY   STATUS    RESTARTS   AGE
news-cfcd97f4f-9nfkr   1/1     Running   0          10m
```

   2.6. Verify the status of the **traffic-mesh-proxy** project pods.

```
[student@workstation traffic-mesh]$ oc get pods -n traffic-mesh-proxy
NAME                         READY   STATUS    RESTARTS   AGE
ab-proxy-7bc6db76dc-bfjgp    1/1     Running   0          10m
```

3. Configure an ingress gateway named **traffic-mesh-gateway** to allow ingress HTTP traffic on port **80** to enter the mesh.

3.1. Create a Gateway object YAML file, for example **traffic-mesh-gateway.yaml**, to store the object definition.

The completed object definition is available in the **~/DO328/solutions/traffic-mesh/traffic-mesh-gateway.yaml** file. Use it to verify your file and fix mistakes.

3.2. Create the gateway configuration with **oc create**.

```
[student@workstation traffic-mesh]$ oc create -f traffic-mesh-gateway.yaml
gateway.networking.istio.io/traffic-mesh-gateway created
```

4. Deploy a new **beta** version of the **frontend** service with the following characteristics:

- **Name**: **frontend-beta**.
- Use the **quay.io/redhattraining/ossm-frontend:beta** image and port **3000**.
- The container needs a environment variable called **REACT_APP_GW_ENDPOINT**. This variable gathers the value for the key **GW_ADDR** stored in the **frontend-cm** config map.

4.1. Create a Deployment object YAML file, for example **new-frontend-deployment.yaml**, to store the object definition.

The completed object definition is available in the **~/DO328/solutions/traffic-mesh/new-frontend-deployment.yaml** file. You can use the solution file to verify and fix mistakes in your file.

4.2. Create the deployment with the **oc create** command.

```
[student@workstation traffic-mesh]$ oc create -f new-frontend-deployment.yaml
deployment.apps/frontend-beta created
```

5. Split the **frontend** service into subsets with the following characteristics:

- **Name**: **frontend-destination-rule**.
- Create a group with all the instances that have the **version: v2** tag and assign to this group the name **v2**.
- Create a group with all the instances that have the **version: beta** tag and assign to this group the name **beta**.

5.1. Create a DestinationRule object YAML file, for example **frontend-destination-rule.yaml**, to store the object definition.

The completed object definition is available in the **~/DO328/solutions/traffic-mesh/frontend-destination-rule.yaml** file. You can use the solution file to verify and fix mistakes in your file.

5.2. Create the destination rules with the **oc create** command.

```
[student@workstation traffic-mesh]$ oc create -f frontend-destination-rule.yaml
destinationrule.networking.istio.io/frontend-destination-rule created
```

6. Create a virtual service named **frontend-vs**, which uses the ingress gateway and redirects the **frontend** traffic following this rules:

   - Traffic with a header matching **version: beta** is redirected to the **beta** subset.
   - Traffic with wrong or missing headers is redirected to the **v2** subset.

Note
All traffic for the front end service uses the **/frontend** prefix.

6.1. Create a VirtualService object YAML file, for example **frontend-virtual-service.yaml**, to store the object definition.

The completed object definition is available in the **~/DO328/solutions/traffic-mesh/frontend-virtual-service.yaml** file. You can use the solution file to verify and fix mistakes in your file.

6.2. Create the virtual service with the **oc create** command.

```
[student@workstation traffic-mesh]$ oc create -f frontend-virtual-service.yaml
virtualservice.networking.istio.io/frontend-vs created
```

7. Test the routing configuration using the **traffic-mesh-proxy** application.

7.1. Export the **traffic-mesh-proxy** URL to an environment variable called **AB_PROXY_URL**.

```
[student@workstation traffic-mesh]$ export AB_PROXY_URL=$(oc get route ab-proxy \
> -n traffic-mesh-proxy -o template --template '{{ "http://" }}{{ .spec.host }}')
```

7.2. Use the **proxy-set-v2-config.sh** helper script to configure the proxy to append the **version: v2** headers to all requests.

```
[student@workstation traffic-mesh]$ sh proxy-set-v2-config.sh
Updated proxy to send version:v2 headers
```

7.3. Use a web browser to verify the route configuration for the **version: v2** HTTP headers. The Financial application is accessible through the AB proxy URL. Use your favorite browser to open that URL, or use the following command to open the URL in Firefox:

```
[student@workstation traffic-mesh]$ firefox ${AB_PROXY_URL}/frontend
```

All requests through the AB proxy append the **version: v2** header, so the web UI shows a dark header.

7.4. Use the **proxy-set-beta-config.sh** helper script to configure the proxy to append the **version: beta** headers to all requests.

```
[student@workstation traffic-mesh]$ sh proxy-set-beta-config.sh
Updated proxy to send version:beta headers
```

7.5. Reload the web page in the browser to verify the route configuration for the **`version: beta`** HTTP headers.

All requests through the AB proxy append the **`version: beta`** header, so the web UI shows a red header.

Note

Browser cache may cause the application to still display the dark header. Force the web page to reload, instructing the browser to obtain the new page, or use browser instructions to delete the cache and reload the page.

8. Restrict egress traffic globally to registered services only.

8.1. Update the Istio configuration and define outbound traffic policy to allow egress traffic only to registered services.

```
[student@workstation traffic-mesh]$ oc patch smcp basic-install \
> --type merge -n istio-system \
> -p '{"spec":{"istio":{"global":{"outboundTrafficPolicy":
{"mode":"REGISTRY_ONLY"}}}}}'
servicemeshcontrolplane.maistra.io/basic-install patched
```

9. Test the restricted egress policy using the **`traffic-mesh-proxy`** application.

9.1. Reload the web page in your browser, and then navigate to the **`News`** section. The application reports that it cannot load any news.

Note

The propagation of the new policy can take several seconds; you might need to wait to see the changes.

10. Allow egress traffic for the **`news`** service. Assign the name **`news-se`** to the required custom resource.

10.1. Retrieve the host where the external **`news`** service is available.

```
[student@workstation traffic-mesh]$ oc get route news -n traffic-mesh-news \
> -o jsonpath="{.spec.host}"
news-traffic-mesh-news.apps.example.com
```

10.2. Create a service entry resource YAML file, for example **`service-entry.yaml`**, to store the object definition.

A template object definition is available in the **`~/DO328/solutions/traffic-mesh/service-entry.yaml`** file. You can use the solution file to verify and fix mistakes in your file.

Note

Remember to set the host to the correct one, obtained in the previous step.

10.3. Create the service entry with the **`oc create`** command.

```
[student@workstation traffic-mesh]$ oc create -f service-entry.yaml
serviceentry.networking.istio.io/news-se created
```

11. Test the egress traffic using the **traffic-mesh-proxy** application.

11.1. Reload the web page in your browser and navigate to the **News** section. The application shows a list of news.

12. Return to the home directory.

```
[student@workstation traffic-mesh]$ cd ~
[student@workstation ~]$
```

Evaluation

Grade your work by running the **lab traffic-mesh grade** command from your **workstation** machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab traffic-mesh grade
```

Finish

On the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab traffic-mesh finish
```

This concludes the lab.

Summary

In this chapter, you learned:

- Red Hat OpenShift Service Mesh implements the sidecar pattern injecting an Envoy sidecar into application pods.
- A **`Gateway`** resource configures how Istio handles ingress and egress connections and exposes services deployed in the mesh.
- **`DestinationRule`** resources define how to route traffic to subsets of services based on different conditions, such as request headers.
- Gateways either allow all egress traffic or restrict egress traffic to services that are registered as **`ServiceEntry`** resources.

Chapter 5

Releasing Applications with OpenShift Service Mesh

Goal	Release applications with canary and mirroring release strategies.
Objectives	• Release application services with a safe canary rollout. • Deploy a "mirror" service to test a new service with a realistic load.
Sections	• Deploying an Application with Canary Releases (and Guided Exercise) • Deploying an Application with a Mirror Launch (and Guided Exercise)
Lab	Releasing Applications with OpenShift Service Mesh

Deploying an Application with Canary Releases

Objectives

After completing this section, you should be able to release application services with a safe canary rollout.

Describing Canary Releases

Deploying software is a complex and risky process. After a release, developers carry out verification activities to check that the deployment is successful, such as watching logs, monitoring metrics and validating that the changes are applied. If something goes wrong, the team rolls back the deployment to the previous stable version.

A simple release approach is to deploy the new version replacing the old one. However, this deployment strategy immediately exposes all users to any defects introduced by the new version.

Canary releases address this problem using a progressive and safe deployment approach where both versions of the application, the old and the new, run in parallel until the new version is completely validated and ready for all users. The new version, also called the canary, initially receives only a small amount of all application traffic. Therefore, if something goes wrong with this new version of the application, a minimal number of users are affected. As you gain confidence with how the canary works, you progressively route more traffic to it.

The complete process is as follows:

- Initially, send a small amount of the traffic to the canary. The old version receives most of the traffic, whereas only a small portion goes to the new version. This ensures that the new version can be tested in production without compromising the stability of the service for the majority of users.

- Test the new version against the limited amount of traffic you configured in the previous step. When you are satisfied with the results, increase the amount of traffic sent to the canary. Repeat this step to slowly make the canary available to more users. Continue this process until the canary receives most of the traffic, while monitoring test results and performance.

- After demonstrating that the new version is sufficiently stable, route the remaining traffic to the canary. At this point, the old version stops receiving traffic. You can remove the old version of the application from the cluster or leave it there temporarily in case you need the option of a quick rollback.

Use Cases

Canary deployment is a good strategy whenever you want to have more control and increase the level of confidence in your deployments. The following cases are scenarios where a canary release approach makes sense:

- Your application handles high loads and you want to perform load or stress testing on a new version.

- You want to validate the new version against a reduced group of users to analyze how this affects your key performance indicators. User groups can be defined according to different

conditions, such as user type or location. For example, a common scenario is releasing canaries only for internal users or trusted clients.

- You need a safe strategy to deploy a new critical version.

Deploying a Canary Release with OpenShift Service Mesh

Canary releases are not specific to OpenShift Service Mesh. You can adopt a canary strategy if you a have a way to control how traffic is distributed between each version. The most common case is using a router or a load balancer that lets you manage traffic routing.

In Kubernetes, you can use the canary strategy by creating a new deployment with the same service selector label as the old version, and adjusting the replica ratio between the old and the new version. The number of replicas of each version dictates the amount traffic that each version receives. Therefore, if both versions, the old and the new, scale to one replica, each of them receives 50% of the traffic.

However, if you want to send only 1% of the traffic to the canary, you would need to scale the old version to 99 replicas to adjust the replica ratio. This is a clear problem, as it requires more cluster resources and leads to an inefficient use of those resources. Moreover, this solution only allows you to control how traffic is routed based on percentages.

With OpenShift Service Mesh, you can take advantage of the Istio traffic management features to handle traffic distribution without a dependency on replica ratios. Furthermore, Istio offers various traffic routing policies, so you are not restricted to a strategy based on traffic percentage.

Virtual services, in combination with destination rules, can define traffic routes for each version. Each version is released as a **`Deployment`** and represented by a **`DestinationRule`** subset, which filters the service endpoints for that version. *Figure 5.1* shows how these configuration resources relate in a canary scenario.

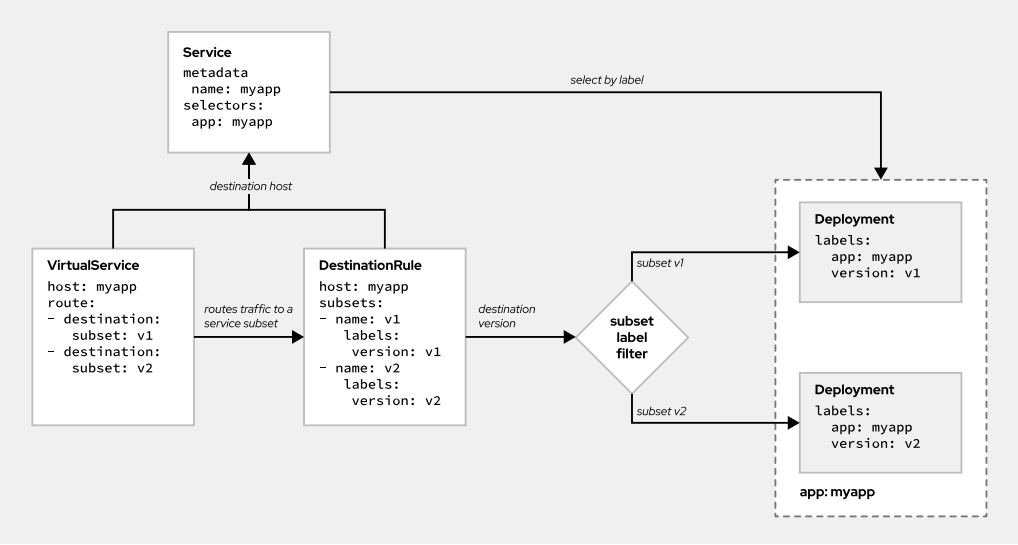


Figure 5.1: Example of Resources Configuration for Canary Releases

First, assume that you have already deployed an application called **`myapp`**. The **`Deployment`** and the **`Service`** resources also already exist. The service that represents the application looks like the following:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: myapp
  name: myapp
spec:
  selector:
    app: myapp
  ...
```

The deployment for the initial version of the application (**v1**) looks like the following:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-v1
spec:
  ...
  template:
    metadata:
      labels:
        app: myapp
        version: v1
      annotations:
        sidecar.istio.io/inject: "true"
    spec:
        ...
```

Pay special attention to the **version** label of the deployment. Labels are the main property used to identify the version of the application, and to split traffic between versions. Also note that you must enable the injection of the Envoy sidecar by setting the **sidecar.istio.io/inject** annotation to **"true"**. This allows the application to use Istio routing features.

Deploying a Canary Release

To deploy a canary release, you must create a new deployment for the new version and split traffic between versions using a virtual service and a destination rule.

1. Create a **Deployment** resource for the new version. For example, for **v2**, create a new **Deployment** that looks like this:

```
kind: Deployment
metadata:
  name: myapp-v2 (1)
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: myapp (2)
        version: v2 (3)
      annotations:
```

```
        sidecar.istio.io/inject: "true"
    spec:
      containers:
        ...
```

As the example shows, the new deployment must meet the following requirements:

1. Use a unique name for this deployment, different from the name of the old deployment.
2. Specify the same application label used in the old deployment, so that the service is aware that the new version belongs to the same application as the old version.
3. Change the value of the version label so that OpenShift Service Mesh can distinguish between versions when routing traffic.

2. Create a **`DestinationRule`** resource to define the subsets that represent each version.

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: myapp
spec:
  host: myapp(1)
  subsets:
    - name: v1(2)
      labels:
        version: v1
    - name: v2(3)
      labels:
        version: v2
```

This destination rule defines the following configuration:

1. The name of the targeted service.
2. A subset for **`v1`**, filtering labels specified in the **`myapp-v1`** deployment. In this case, **`version: v1`**.
3. A subset for **`v2`**, filtering labels specified in the **`myapp-v2`** deployment. In this case, **`version: v2`**.

3. Create a **`VirtualService`** resource to define the traffic route for each version. Associate each subset with a route destination and add a weight.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: myapp
spec:
  hosts:
    - "*"
  gateways:
    - my-gateway
  http:
    - route:
      - destination:
          host: myapp(1)
          subset: v1(2)
          port:
```

```
            number: 3000
        weight: 90 (3)
      - destination:
          host: myapp
          subset: v2
          port:
            number: 3000
        weight: 10
```

There are two **`destination`** configurations defined in this virtual service, one for each version. **`v1`** receives 90% of the traffic, whereas **`v2`** receives the remaining 10%. Each destination must contain the following information:

(1) The service hostname. It must be the same as the **`host`** field defined in the destination rule.
(2) The name of one of the subsets defined in the associated destination rule.
(3) The percentage of traffic routed to the version.

If you need to route traffic using more advanced criteria, then you can use **`spec.http.match`** for HTTP traffic or **`spec.tcp.match`** for TCP traffic.

After you create these resources, ensure that traffic is being routed as expected. To verify the routing, send traffic to your application and check that each version receives the expected amount of traffic. You can check your application responses, inspect the logs, or use Kiali, as discussed in *Inspecting Canary Traffic with Kiali*.

When you are ready to send more traffic to your canary, update the route weights or matching filters. For example, if you want to increase the traffic share received by the canary to 60%, update the **`weight`** field of each subset to adjust the new percentages.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: myapp
spec:
  ...
  http:
    - route:
      - destination:
          host: myapp
          subset: v1
          ...
        weight: 40
      - destination:
          host: myapp
          subset: v2
          ...
        weight: 60
```

Similarly, if you need to quickly roll back to the previous version, update the weights to direct more traffic to it.

Splitting Traffic Using Kiali

Kiali provides a web user interface to observe and manage the services in your service mesh in real time. As well as providing observability of the platform, Kiali offers configuration features to help you manage the platform. With Kiali, you can edit the configuration of Istio traffic management features to control canary releases. In particular, you can edit resource files in **`yaml`** format to modify the configuration of Istio resources. You can also use traffic management wizards to balance traffic between versions in a user-friendly interface.

Kiali can help you to control how traffic is routed to different versions without explicitly creating the **`VirtualService`** and the **`DestinationRule`** resources. Instead of creating or editing these resources in OpenShift, for example with the **`oc`** command, you can use the Kiali configuration capabilities.

In the Kiali menu, navigate to **Services** to open the services page.

Next, select your project in the **Namespace** selector and click the name of your application service to go to the service configuration page:

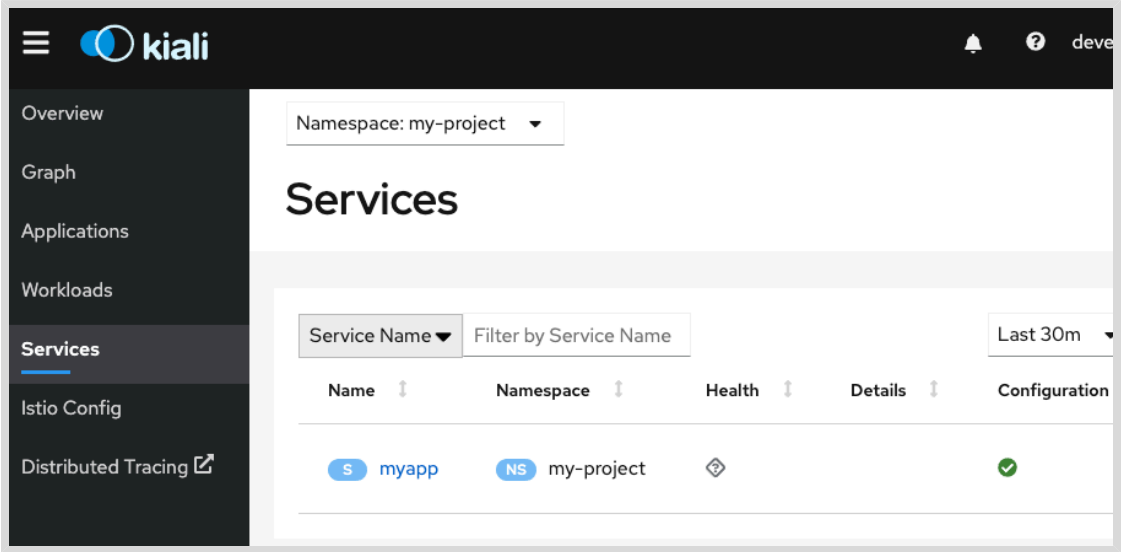


Figure 5.2: Kiali Services Page

On the lower part of the page, information about workloads, virtual services, and destination rules is distributed in tabs. If you have created the deployments but have not yet created any Istio traffic management resources, neither Virtual Services nor Destination Rules should exist on the screen, as shown in the following example:

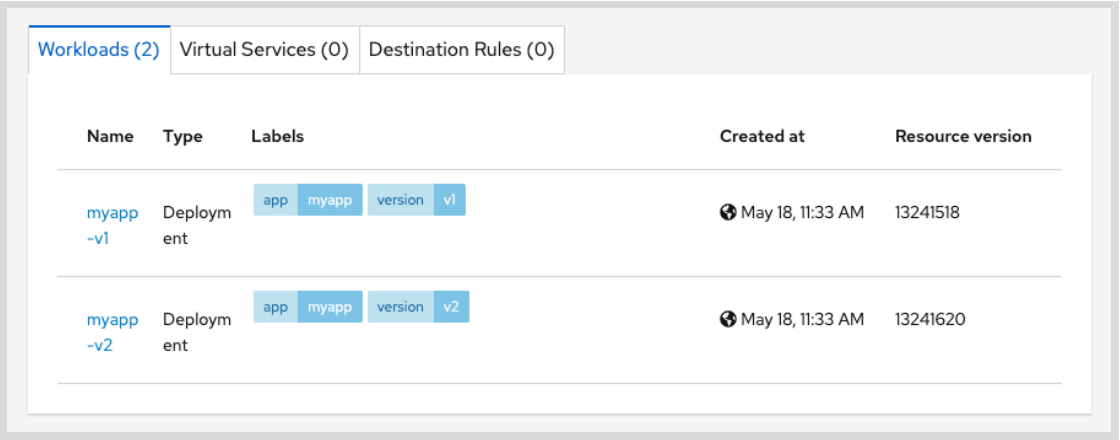


Figure 5.3: Service Resources Section in Kiali

Open the weighted routing wizard to configure traffic splitting. Select **Create Weighted Routing** from the Actions list in the upper right.

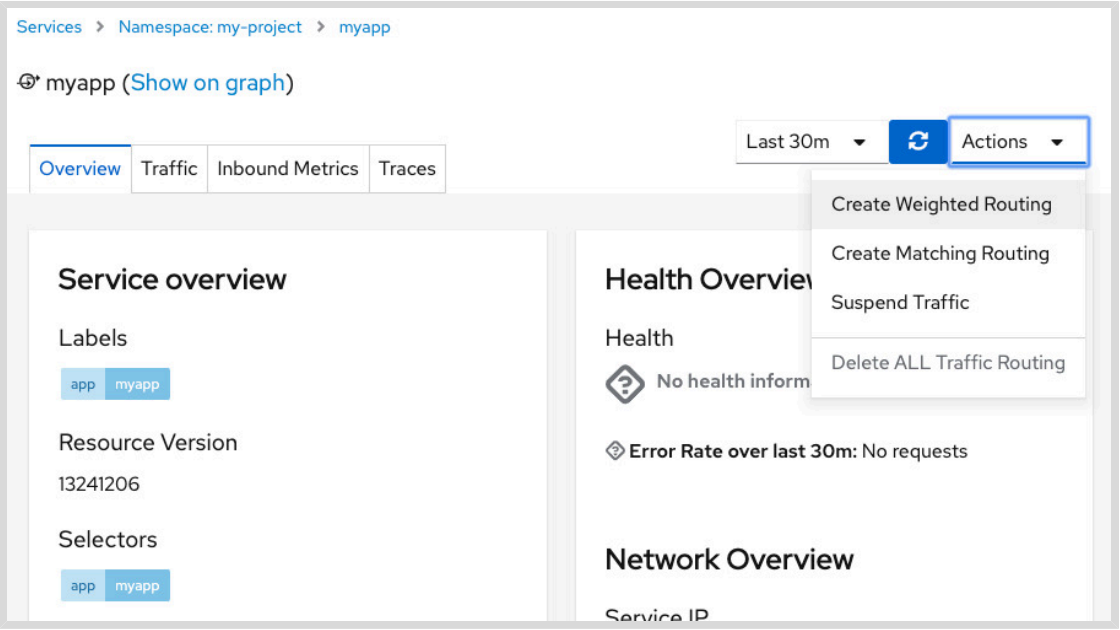


Figure 5.4: Kiali Weighted Routing Wizard Selector

In the Create Weighted Routing window that displays, use the sliders to adjust the traffic percentage assigned to each version.

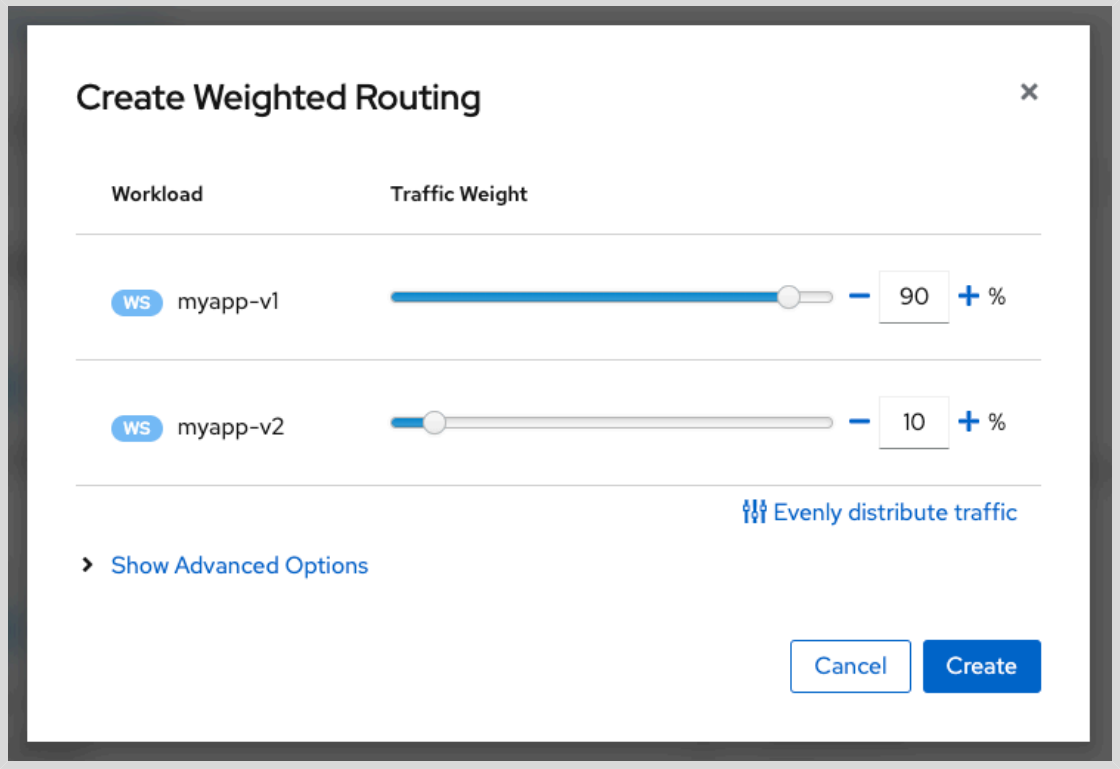


Figure 5.5: Kiali Weighted Routing Wizard

If you need to setup additional options, such as binding TLS or gateway binding, click **Show Advanced Options**.

Note
Currently, the weighted routing wizard has limited capabilities and only handles very basic scenarios, such as routing traffic based on percentages. Moreover, it is only capable of managing **`VirtualService`** and **`DestinationRule`** resources created with the wizard. It does not allow the management of **`VirtualService`** and **`DestinationRule`** resources created directly in OpenShift.

When you need to increase the amount of traffic directed to the canary, reopen the wizard and adjust the sliders.

Warning
There is a bug in Kiali UI v1.12 that causes the deletion of the gateway field when updating the configuration of the weighted routing wizard. If you encounter this problem, edit the wizard configuration again to restore the gateway value.

Inspecting Canary Traffic with Kiali

You can inspect the traffic flow in the Kiali Graph section and visualize how traffic is distributed to each version. Click **`Graph`** in the navigation pane to visualize the graph. Make sure that you select **`Versioned app graph`** as the graph type and **`Requests percentage`** as the graph edge labels.

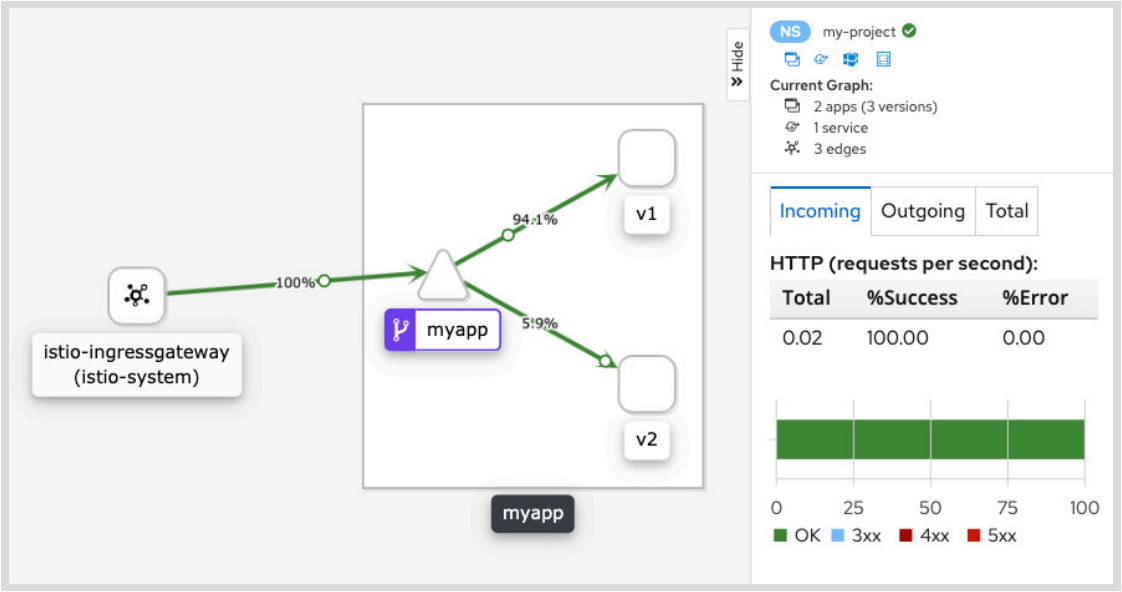


Figure 5.6: Kiali Traffic Graph

Notice how Kiali identifies that the two versions belong to the same application. You can also see the amount of traffic that each version receives. This allows you to verify that your traffic splitting configuration works as expected.

Configuring Istio Resources in Kiali

As well as providing wizards to manage traffic, Kiali also allows you to edit the configuration of Istio resources in **`yaml`** format. To edit the configuration of these resources, select **Services** in the Kiali navigation pane, select the resource you want to edit, and then edit the **`yaml`** code.

Editing the configuration of a resource from the Kiali UI has the same result as editing the resource using the **oc** command.

Additionally, you can modify Istio configuration in Kiali by editing the **`yaml`** code in the Istio Config section. Select **Istio config** in navigation pane. Select the resource you want to edit, and then click the YAML tab. In this tab, you can view the specification in **`yaml`** format and modify it.

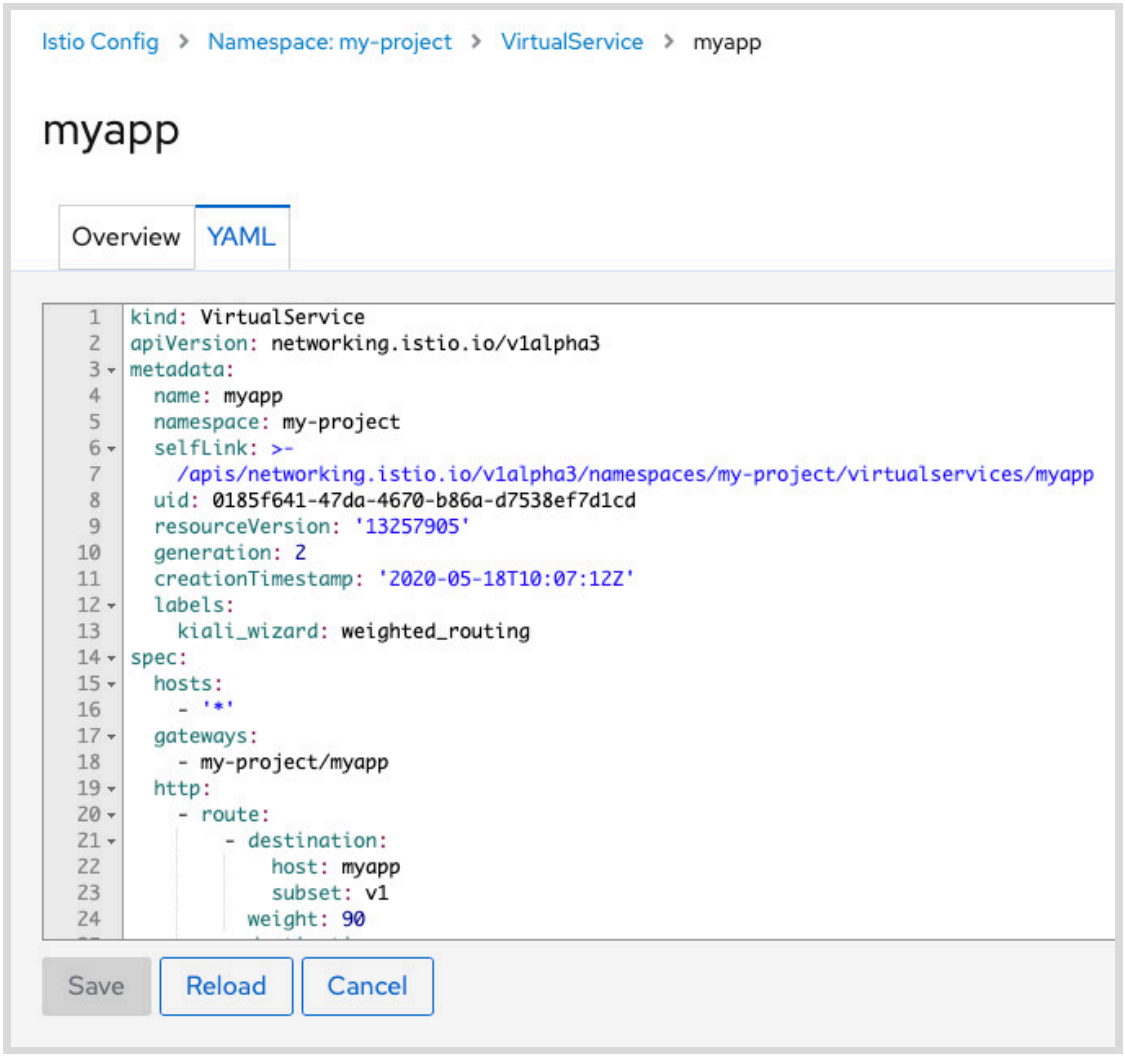


Figure 5.7: YAML edition screen in Kiali

References

Canary Deployments using Istio
https://istio.io/blog/2017/0.1-canary/

Istio 1.4 Docs: Destination Rule
https://archive.istio.io/v1.4/docs/reference/config/networking/destination-rule/

Istio 1.4 Docs: Virtual Service
https://archive.istio.io/v1.4/docs/reference/config/networking/virtual-service/

References

For more information, refer to the *Kiali overview* section in the *Red Hat Service Mesh Guide* at
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.4/html-single/service_mesh/index

Guided Exercise

Deploying an Application with Canary Releases

In this exercise, you will use Red Hat OpenShift Service Mesh to deploy different versions of a service using canary releases.

Outcomes

You should be able to:

- Roll out versions of an application as canary releases using OpenShift Service Mesh.
- Specify the traffic percentage directed to each canary deployment.
- Visualize traffic distribution and identify errors using Kiali.

Before You Begin

To perform this exercise, ensure you have access to:

- A configured and running OpenShift cluster.
- An installed and running OpenShift Service Mesh in the OpenShift cluster.
- The OpenShift CLI (**`/usr/local/bin/oc`**).

As the **`student`** user on the **`workstation`** machine, use the **`lab`** command to prepare your system for this exercise.

Run the following command on **`workstation`** to setup the environment.

```
[student@workstation ~]$ lab release-canary start
```

1. Review the resource files provided. The setup script used these files to deploy the initial version of the **`vertx-greet`** application.

   The **`vertx-greet`** application exposes a GET endpoint that returns a greeting message. It also implements a basic rate limit feature to help you understand one of the scenarios in this exercise. Note that this limit is not the same as the rate limit feature included in OpenShift Service Mesh. The source code is available in the Git repository at `https://github.com/RedHatTraining/DO328-apps/` in the **`vertx-greet`** directory.

   1.1. Navigate to the directory where the cluster resource files for this exercise are located.

```
[student@workstation ~]$ cd ~/DO328/labs/release-canary/
```

   1.2. Review the **`deployment-v1.yaml`** resource file.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: vertx-greet-v1
spec:
  selector:
    matchLabels:
      app: vertx-greet
  replicas: 1
  template:
    metadata:
      labels:
        app: vertx-greet
        version: v1
      annotations:
        sidecar.istio.io/inject: "true"
    spec:
      containers:
        - name: vertx-greet
          image: quay.io/redhattraining/ossm-vertx-greet:1.0
          ports:
            - containerPort: 8080
```

This file defines the **`Deployment`** resource for the initial version of the application. Each version is associated to a deployment. The initial version is **`v1`**, which is reflected both in **`metadata.name`** and **`spec.template.metadata.labels.version`**.

1.3. Review the **`service.yaml`** resource file.

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: vertx-greet
  name: vertx-greet
spec:
  ports:
    - name: http
      port: 8080
      protocol: TCP
      targetPort: 8080
  selector:
    app: vertx-greet
```

The service directs internal traffic to the deployment pods. You do not need to change this service or create new ones when deploying canary releases.

1.4. Review the **`gateway.yaml`** resource file.

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

```
  selector:
    istio: ingressgateway
  servers:
    - port:
        number: 80
        name: http
        protocol: HTTP
      hosts:
        - "*"
```

The istio gateway sits at the edge of the service mesh and allows incoming connections to the application.

1.5. Review the **`virtual-service.yaml`** resource file.

```
apiVersion: networking.istio.io/v1alpha3
  kind: VirtualService
  metadata:
    name: vertx-greet
  spec:
    hosts:
    - "*"
    gateways:
    - vertx-greet-gateway
    http:
    - route:
      - destination:
          host: vertx-greet
          port:
            number: 8080
```

The Istio virtual service defines traffic routes from the istio gateway to the application and defines traffic splitting for each version. Initially, the virtual service only includes one route, as only one version of the application is deployed.

2. Verify that the initial version application is deployed and running.

2.1. Run the following command to load the environment variables for this exercise.

```
[student@workstation release-canary]$ source /usr/local/etc/ocp4.config
```

2.2. Log in to OpenShift as the **`developer`** user.

```
[student@workstation release-canary]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_API}
Login successful.
...output omitted...
```

Ensure that you are using the **`release-canary`** project.

```
[student@workstation release-canary]$ oc project release-canary
...output omitted...
```

Next, perform the rest of the steps in this project.

2.3. Check that all the resources have been deployed by the setup script.

```
[student@workstation release-canary]$ oc get \
> deployment,pod,service,virtualservice,gateway
NAME                              READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/vertx-greet-v1    1/1     1            1           5m38s

NAME                                  READY   STATUS    RESTARTS   AGE
pod/vertx-greet-v1-bd87877dd-6rj2m    2/2     Running   0          5m38s

NAME                  TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
service/vertx-greet   ClusterIP   172.30.37.155   <none>        8080/TCP   5m38s

NAME                                           GATEWAYS                ...
virtualservice.networking.istio.io/vertx-greet   [vertx-greet-gateway]   ...

NAME                                           AGE
gateway.networking.istio.io/vertx-greet-gateway   5m38s
```

2.4. Use the **`oc get route`** command to get the URL of the istio gateway. Save the result into a variable for later use.

```
[student@workstation release-canary]$ GATEWAY_URL=$(oc get route istio-
ingressgateway \
> -n istio-system -o jsonpath='{.spec.host}')
```

2.5. Verify that the application works. Make a request to the route URL that you stored in the **`GATEWAY_URL`** variable.

```
[student@workstation release-canary]$ curl $GATEWAY_URL
Hello World!
```

3. Generate the canary release. Deploy version 2 of the **`vertx-greet`** application by creating a new deployment. After the new version is deployed, route a portion of the traffic to the new version.

3.1. Make a copy of the **`deployment-v1.yaml`** file and name it **`deployment-v2.yaml`**.

```
[student@workstation release-canary]$ cp deployment-v1.yaml deployment-v2.yaml
```

3.2. Modify the **`deployment-v2.yaml`** file to introduce the changes for version 2. Change the value of **`metadata.name`** to **`vertx-greet-v2`**, change the value of **`spec.template.metadata.labels.version`** to **`v2`**. Finally, add the **`GREETING`** environment variable with the value **`Hello Red Hat!`**.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: vertx-greet-v2
spec:
  selector:
    matchLabels:
```

```
      app: vertx-greet
  replicas: 1
  template:
    metadata:
      labels:
        app: vertx-greet
        version: v2
      annotations:
        sidecar.istio.io/inject: "true"
    spec:
      containers:
        - name: vertx-greet
          image: quay.io/redhattraining/ossm-vertx-greet:1.0
          ports:
            - containerPort: 8080
          env:
          - name: GREETING
            value: "Hello Red Hat!"
```

You can see the complete deployment for version 2 at **~/DO328/solutions/release-canary/deployment-v2.yaml**.

3.3. Deploy version 2 by creating the new deployment.

```
[student@workstation release-canary]$ oc create -f deployment-v2.yaml
deployment.apps/vertx-greet-v2 created
```

3.4. Review the **destination-rule.yaml** file provided.

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

The file defines two subsets, one for each version. Each subset represents a portion of the traffic, and includes a name and a label that associates the subset with the deployment version. The version is the one specified by the **spec.template.metadata.labels.version** property in each **Deployment** resource.

3.5. Create a DestinationRule with the **destination-rule.yaml** file.

```
[student@workstation release-canary]$ oc create -f destination-rule.yaml
destinationrule.networking.istio.io/vertx-greet created
```

3.6. Modify the **`VirtualService`** resource with the **`oc edit`** command to send 80% of the traffic to **`v1`** and the remaining 20% to **`v2`**. Associate the existing destination with the **`v1`** subset and add a weight of 80. Next, add another destination for the **`v2`** subset with a weight of 20.

```
[student@workstation release-canary]$ oc edit virtualservice vertx-greet
```

Apply the changes in the text editor.

```
...output omitted...
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  ...output omitted...
  name: vertx-greet
  ...output omitted...
spec:
  hosts:
  - "*"
  gateways:
  - vertx-greet-gateway
  http:
  - route:
    - destination:
        host: vertx-greet
        subset: v1
        port:
          number: 8080
      weight: 80
    - destination:
        host: vertx-greet
        subset: v2
        port:
          number: 8080
      weight: 20
...output omitted...
```

Save your changes to the resource, and then close the editor.

3.7. Run **`test_canary.py`** to check that the portion of responses expected for each service corresponds to the weights configured in the previous steps. This script sends a sequence of 50 requests to the specified URL and shows the result. From a previous step, you should have the gateway route URL stored in the **`GATEWAY_URL`** environment variable. Execute the script passing this variable as the first parameter.

```
[student@workstation release-canary]$ ./test_canary.py $GATEWAY_URL
```

Wait until the script ends.

```
...output omitted...
Hello World!
Hello World!
Hello World!
Hello World!
```

```
Hello World!
Hello World!
Hello World!
Hello Red Hat!
...output omitted...
Total requests: 50
* 'Hello World!' responses: 42 (84.0%)
* 'Hello Red Hat!' responses: 8 (16.0%)
* Errors: 0 (0.0%)
```

Notice how the response percentages for each version show values close to the weights that you specified in the **`VirtualService`** resource. Expect to see small deviations in the traffic statistics when compared to the configured weights, as there is a degree of random variation in traffic splitting.

4. Create another canary release for version 3. This version includes a rate limit feature, which establishes the maximum number of requests per second that the application can handle. The application will reject requests received at a rate that exceeds this limit. After deploying version 3, modify the traffic weights so that traffic is now distributed between the three versions.

4.1. Make a copy of the **`deployment-v2.yaml`** file and name it **`deployment-v3.yaml`**.

```
[student@workstation release-canary]$ cp deployment-v2.yaml deployment-v3.yaml
```

4.2. Introduce the changes for version 3 in **`deployment-v3.yaml`**. Change the name to **`vertx-greet-v3`**, change the value of **`spec.template.metadata.labels.version`** to **`v3`**, change the **`GREETING`** value, and add a new environment variable **`MAX_REQUESTS_PER_SECOND`** with a value of 1.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: vertx-greet-v3
spec:
  selector:
    matchLabels:
      app: vertx-greet
  replicas: 1
  template:
    metadata:
      labels:
        app: vertx-greet
        version: v3
      annotations:
        sidecar.istio.io/inject: "true"
    spec:
      containers:
        - name: vertx-greet
          image: quay.io/redhattraining/ossm-vertx-greet:1.0
          ports:
            - containerPort: 8080
          env:
```

```
          - name: GREETING
            value: "Hello v3!"
          - name: MAX_REQUESTS_PER_SECOND
            value: "1"
```

You can see the complete deployment for version 3 at **~/DO328/solutions/release-canary/deployment-v3.yaml**.

4.3. Deploy version 3 by creating the new deployment.

```
[student@workstation release-canary]$ oc create -f deployment-v3.yaml
deployment.apps/vertx-greet-v3 created
```

4.4. Modify the **DestinationRule** resource. Use the **oc edit** command to create a new **subset** for version 3.

```
[student@workstation release-canary]$ oc edit destinationrule vertx-greet
```

In the text editor, add the **subset** for **v3**.

```
...output omitted...
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  ...output omitted...
  name: vertx-greet
  ...output omitted...
spec:
  host: vertx-greet
  subsets:
  - labels:
      version: v1
    name: v1
  - labels:
      version: v2
    name: v2
  - labels:
      version: v3
    name: v3
...output omitted...
```

Save and close the editor to apply the changes.

4.5. Modify the **VirtualService** resource to assign a small portion of traffic to **v1** and balance the rest between **v2** and **v3**. Use the **oc edit** command to modify the **weight** property of **v1** and **v2**, and to add a new **destination** node for **v3**.

```
[student@workstation release-canary]$ oc edit virtualservice vertx-greet
```

In the text editor, modify the **weight** property of **v1** and **v2**, and add a new **destination** for **v3**.

```
...output omitted...
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  ...output omitted...
  name: vertx-greet
  ...output omitted...
spec:
  hosts:
  - "*"
  gateways:
  - vertx-greet-gateway
  http:
  - route:
    - destination:
        host: vertx-greet
        subset: v1
        port:
          number: 8080
      weight: 10
    - destination:
        host: vertx-greet
        subset: v2
        port:
          number: 8080
      weight: 45
    - destination:
        host: vertx-greet
        subset: v3
        port:
          number: 8080
      weight: 45
...output omitted...
```

Save and close the editor to apply the changes.

4.6. Run **`test_canary.py`** again to check that the portion of responses expected for each service corresponds to the weights configured. Execute **`test_canary.py`** passing **`GATEWAY_URL`** as a parameter.

```
[student@workstation release-canary]$ ./test_canary.py $GATEWAY_URL
...output omitted...
Hello Red Hat!
Hello World!
Hello Red Hat!
Hello Red Hat!
Hello v3!
Server responded with error  HTTP Error 503: Service Unavailable

Server responded with error  HTTP Error 503: Service Unavailable

Hello World!
...output omitted...
```

```
Total requests: 50
* 'Hello World!' responses: 5 (10.0%)
* 'Hello Red Hat!' responses: 18 (36.0%)
* 'Hello v3!' responses: 3 (6.0%)
* Errors: 24 (48.0%)
```

Notice how version 3 only returns a few correct responses, even though there are a significant number of errors. The new version, which limits the number of requests per second, is unable to attend all requests when they come at a high rate.

5. Inspect traffic with Kiali to gain more insight into traffic routing.

5.1. Obtain the Kiali web console url.

```
[student@workstation release-canary]$ KIALI_URL=$(oc get route \
> -n istio-system kiali \
> -o jsonpath='http://{.spec.host}')
```

5.2. Open the Kiali web console in the web browser.

```
[student@workstation release-canary]$ firefox $KIALI_URL &
```

5.3. Click **Log in with OpenShift**. Log in using the developer account. Find your credentials in the **`/usr/local/etc/ocp4.config`** classroom configuration file. The user name is in the **`RHT_OCP4_DEV_USER`** variable and the password is in **`RHT_OCP4_DEV_PASSWORD`**.

5.4. In the Kiali web console navigation pane, click **Graph** to open the graph view.

5.5. Click **Select a namespace** → **release-canary** to gather metrics for the **`release-canary`** project.

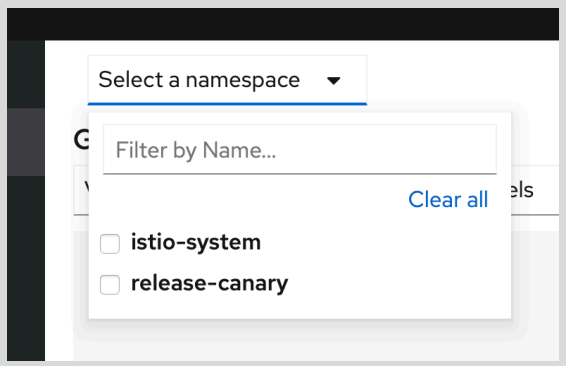


Figure 5.8: Kiali namespace selector

5.6. Click **No edge labels** → **Requests percentage** to activate traffic percentage labels in graph edges.

5.7. Click **Display** → **Traffic Animation** to include traffic animation in the graph.

5.8. Specify a longer duration for metric queries to make sure you can inspect all recent traffic. Click **Last 1m** → **Last 30m** to show the graph with metrics from the last 30 minutes.

5.9. Take a moment to inspect the traffic graph. Whereas **v1** and **v2** are working as expected, **v3** is causing the errors. Click the **v3** rounded square representing the workload. Notice how many of the requests to this version result in an error.

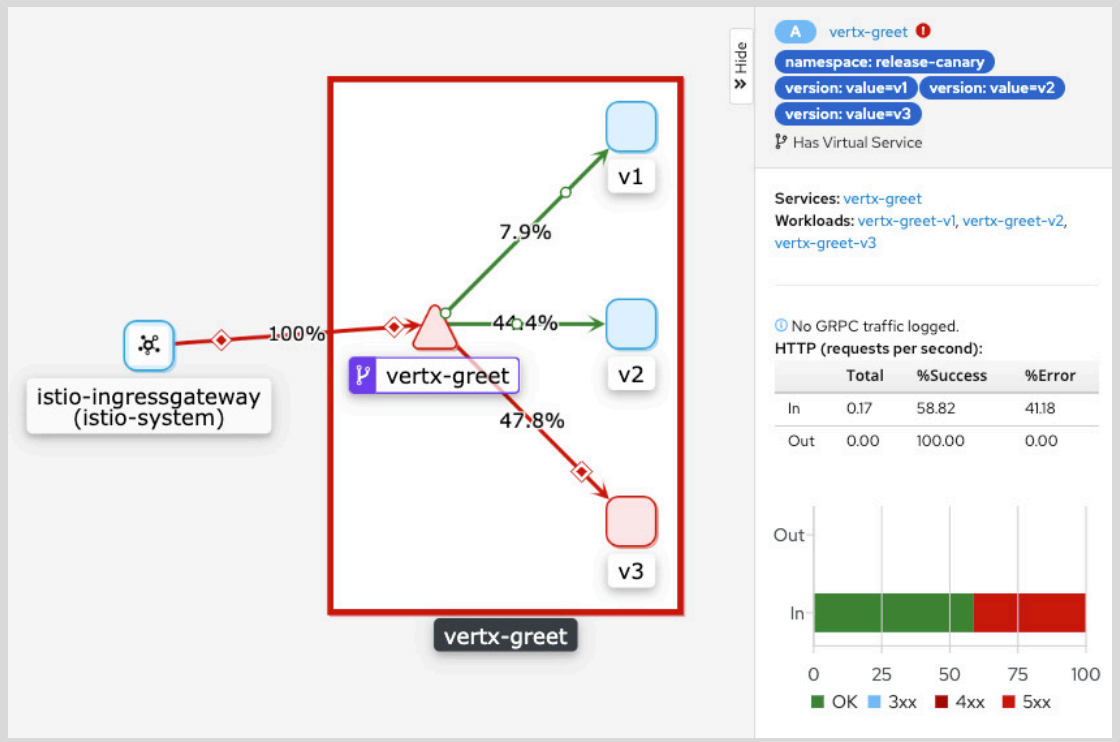


Figure 5.9: Kiali traffic graph with errors

The new rate limiting feature introduced in **v3** causes this behavior, as it does not allow for more than one request per second. When you run the **`test_canary.py`** script, you send a sequence of 50 requests to the application. Roughly 45% of them are routed to **v3**, as you specified in the **`VirtualService`** resource. The script does not apply any delay between requests, so as soon as one is completed, the next one is sent. Under this scenario, **v3** is receiving more than one request per second. Therefore, you must modify the traffic share so that **v3** does not receive more requests than it can handle.

6. Reduce the traffic share of version 3. Modify the **`VirtualService`** resource to adjust the traffic load of version 3.

6.1. Use the **`oc edit`** command to modify the **`VirtualService`** resource again and reduce the traffic share of **v3**.

```
[student@workstation release-canary]$ oc edit virtualservice vertx-greet
```

In the text editor, reduce the **`weight`** of **v3** and send the rest to **v2**.

```
...output omitted...
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: vertx-greet
spec:
```

```
  hosts:
  - "*"
  gateways:
  - vertx-greet-gateway
  http:
  - route:
    - destination:
        host: vertx-greet
        subset: v1
        port:
          number: 8080
      weight: 10
    - destination:
        host: vertx-greet
        subset: v2
        port:
          number: 8080
      weight: 88
    - destination:
        host: vertx-greet
        subset: v3
        port:
          number: 8080
      weight: 2
...output omitted...
```

Save and close the editor to apply the changes.

6.2. Run **`test_canary.py`** again. Increase the number of requests to 100 by adding the **`requests`** parameter, so that you can see some requests routed to **v3**.

```
[student@workstation release-canary]$ ./test_canary.py $GATEWAY_URL --requests 100
```

Wait until the script ends.

```
...output omitted...
Total requests: 100
* 'Hello World!' responses: 5 (5.0%)
* 'Hello Red Hat!' responses: 93 (93.0%)
* 'Hello v3!' responses: 2 (2.0%)
* Errors: 0 (0.0%)
```

Now, **v3** does not limit requests, as only 2% of the traffic is sent to this version. Therefore, **v3** returns no errors.

Note

If you do not see any requests routed to **v3**, run the **`test_canary.py`** script again. The traffic weight assigned to **v3** is now very low and it is possible that **v3** receives 0 out of 100 requests.

Similarly, errors can occur if, by chance, two requests are directed to **v3** in less than a second. Rerun the **`test_canary.py`** script in that case too.

7. Inspect the traffic graph in Kiali again and verify that **v3** is working as expected after the traffic reduction.

7.1. Switch back to the Kiali console in the browser.

7.2. Click **Graph** in the Kiali console navigation pane. Select **`Last 5min`** in the metrics duration selector to only show traffic for the last 5 minutes. Check that the traffic graph does not show any errors.

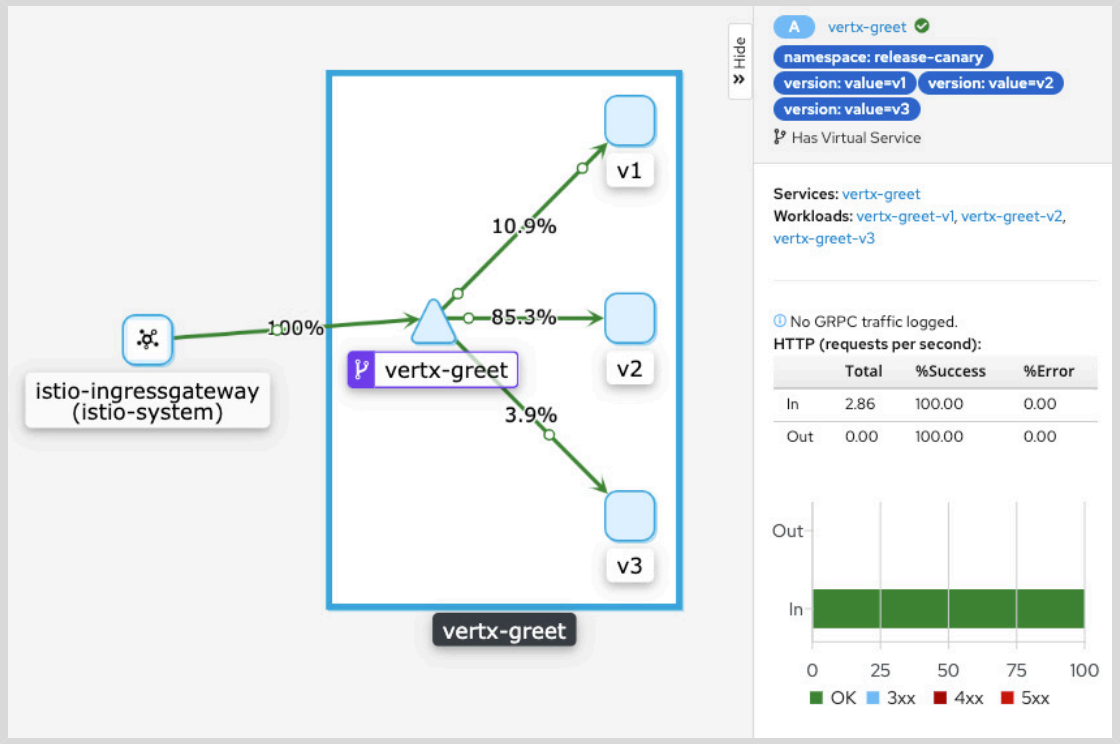


Figure 5.10: Kiali traffic graph without errors

8. Return to the home directory.

```
[student@workstation release-canary]$ cd ~
[student@workstation ~]$
```

Finish

On the **`workstation`** machine, use the **`lab`** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab release-canary finish
```

This concludes the guided exercise.

Deploying an Application with a Mirror Launch

Objectives

After completing this section, you should be able to deploy a "mirror" service to test a new service with a realistic load.

Testing in Production

When releasing new versions of services, it is often necessary to test the new versions of the services in a production environment. Testing in production is important because it is difficult to simulate production workloads or realistic production data in testing or staging environments.

Testing in production consists of:

- Deploying the new version of the service in a production environment alongside the existing version of the service.
- Sending a copy of production traffic to both the new service and the existing service. This is known as **`traffic mirroring`**.
- Verifying the correct behavior of the new service.

The responses sent back to clients are sent from the previous version of the service, so there is little risk of service disruption.

Traffic mirroring is also beneficial when the services involved are stateful. When the new version of the service starts functioning, it starts with a state known to be compatible with the current state of the old version. Sending the same traffic to both stateful services allows them to maintain a synchronized state. When the new version of the service becomes the production version, it will hold the right state.

These situations may be complex to solve programmatically, but OpenShift Service Mesh features make them feasible. Traffic mirroring allows testing the new service in production, with production requests, without service disruption, and keeping the state of both versions of stateful services synchronized.

Note

Traffic mirroring is also known as **`Mirror Launches`** or **`Dark Launches`**, referring to the capacity of releasing new versions of services while keeping them hidden from clients.

Mirroring in OpenShift Service Mesh

OpenShift Service Mesh uses **`DestinationRule`** resources to define subsets (usually service versions) and the **`destination`** entry in **`VirtualService`** resources to route the requests between subsets. OpenShift Service Mesh provides traffic mirroring by using the same **`DestinationRule`** resources and introducing a **`mirror`** entry in the **`VirtualService`** route:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: my_virtual_service
spec:
  hosts:
    - target_host
  http:
  - route:
    - destination:
        host: old_service_name
        subset: old_subset
    mirror: (1)
      host: new_service_name (2)
      subset: new_subset (3)
```

(1) The **`mirror`** entry defines the service to which Istio is sending request copies.
(2) The name of the service that receives the mirrored traffic.
(3) The subset of hosts that receive the mirrored traffic, as defined in the **`DestinationRule`**.

Note

The **`mirror`** entry is an attribute of the objects forming the **`http`** array. If you find the indentation and representation in **`yaml`** format to be confusing, consider the following snippet of the same resource in **`json`** format:

```
{
    "apiVersion": "networking.istio.io/v1alpha3",
    "kind": "VirtualService",
    "metadata": {
        "name": "my_virtual_service",
    },
    "spec": {
        "hosts": [
            "target_host"
        ],
        "http": [
            {
                "route": [
                    {
                        "destination": {
                            "host": "old_service_name",
                            "subset": "old_subset"
                        }
                    }
                ],
                "mirror": {
                    "host": "new_service_name",
                    "subset": "new_subset"
                }
            }
        ]
    }
}
```

Istio does not distinguish whether the mirror host is an external or an internal service. Istio can mirror traffic to any service with a related **`VirtualService`** resource.

Mirroring a Percentage of the Traffic

There are situations where it is not needed or desirable to mirror all the traffic to the new service. For example, when maintaining the latest state of the service is not required, or when reducing traffic between services is more important than testing all requests. In those situations, Istio and OpenShift Service Mesh allow defining a percentage of the mirrored traffic:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: my_virtual_service
spec:
  hosts:
    - target_host
  http:
```

```
  - route:
    - destination:
        host: old_service_name
        subset: old_subset
    mirror:
      host: new_service_name
      subset: new_subset
    mirror_percent: 10
```

In this example, only 10% of the requests sent to **`old_service_name`** are mirrored to **`new_service_name`**.

Selecting the Appropriate Deployment Strategy

Canary releases and traffic mirroring are both deployment strategies that help you validate the release of new service versions. Depending on your deployment and testing plan, you may want to use one or the other. To pick the right strategy, follow these guidelines:

- Use canary releases when you want to deploy a new version directly in production while minimizing the risk. Also, choose this strategy if you need to introduce real users in the validation process of the new version. By picking canary releases, you accept that a small portion of your users might experience problems derived from the new version.

- Use traffic mirroring when you want to test the new version with production load without making the version available to the public. For example, traffic mirroring can be suitable if you want to test how the application responds to 100% of the production load. This approach is less risky than canary releases. If something goes wrong, it does not affect users. However, traffic mirroring is also more limited, because you can not include real users in the validation process.

Combining both strategies is also an interesting option. First, you use traffic mirroring to validate that the new version works correctly with production traffic. Next, you deploy the new version as a canary to start validating it against real users.

References

Mirroring

https://archive.istio.io/v1.4/docs/tasks/traffic-management/mirroring/
in Istio documentation.

Guided Exercise

Deploying an Application with a Mirror Launch

In this exercise, you will perform a **`Dark Launch`** release of an application and control the amount of traffic mirrored to it.

The application you are deploying is a stateful variant of the greetings service: the service respond request with a predefined response in **`v1`**, and a configurable response in **`v2`**. The status kept by both versions includes the amount of requests received. This status is stored in memory, so it is lost if the service is restarted.

Outcomes

You should be able to mirror all or part of the traffic from one service to another to execute a **`Dark Launch`** release.

Before You Begin

To perform this exercise, ensure you have:

- A configured and running OpenShift cluster.
- A installed and running OpenShift Service Mesh in the OpenShift cluster.
- The OpenShift CLI (**`/usr/local/bin/oc`**).

As the **`student`** user on the **`workstation`** machine, use the **`lab`** command to prepare your system for this exercise.

This command creates the **`release-dark`** project in your OpenShift cluster and deploys a single service on it. Although not needed for this exercise, you can review the source code of this service at `https://github.com/RedHatTraining/DO328-apps/` in the **`vertex-greet`** directory.

```
[student@workstation ~]$ lab release-dark start
```

1. Validate that the application has successfully deployed.

1.1. Run the following command to load the environment variables created in *Guided Exercise: Creating a Lab Environment*:

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

1.2. Log in to OpenShift:

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_API}
Login successful.
...output omitted...
```

1.3. Use the **release-dark** namespace for the remainder of this exercise.

```
[student@workstation ~]$ oc project release-dark
Now using project "release-dark" on server ...output omitted...
```

1.4. Verify the application is running by checking its logs and pod status.

```
[student@workstation ~]$ oc get pods
NAME                              READY   STATUS    RESTARTS   AGE
vertx-greet-v1-f44b9976c-skbch    2/2     Running   0          3m11s
```

The actual name of your pods may be different.

2. Deploy a new version of the application, but continue sending traffic to the old version. The new version uses the same image as the old one, **quay.io/redhattraining/vertx-greet:latest**, but introduces a new environment variable that changes the greeting message: **GREETING="Hola Mundo!"**.

2.1. Create the deployment resource for the same **quay.io/redhattraining/vertx-greet:latest** image. Use **vertx-greet-v2** as the deployment name, and set the **version** label to **v2**. Inject the **GREETING** environment variable into the deployed container.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: vertx-greet-v2
spec:
  selector:
    matchLabels:
      app: vertx-greet
      version: v2
  replicas: 1
  template:
    metadata:
      labels:
        app: vertx-greet
        version: v2
      annotations:
        sidecar.istio.io/inject: "true"
    spec:
      containers:
        - name: vertx-greet
          image: quay.io/redhattraining/ossm-vertx-greet:latest
          ports:
            - containerPort: 8080
          resources:
```

```
            limits:
              memory: "200Mi"
              cpu: "250m"
          env:
          - name: GREETING
            value: "Hola Mundo!"
```

Save the **Deployment** resource in a file named **deployment-v2.yml**. You can use the provided solution file at **~/DO328/solutions/release-dark/deployment-v2.yml** to check your file and correct any errors.

2.2. Create the **Deployment** resource in OpenShift using the **oc create** command:

```
[student@workstation ~]$ oc create -f deployment-v2.yml
deployment.apps/vertx-greet-v2 created
```

2.3. Validate the pod for the new version of the service is deployed and running:

```
[student@workstation ~]$ oc get pods
NAME                              READY   STATUS    RESTARTS   AGE
vertx-greet-v1-f44b9976c-dps95    2/2     Running   0          5m4s
vertx-greet-v2-7849968c47-mq7hc   2/2     Running   0          2m39s
```

3. Generate traffic to the **v1** service. Visualize in Grafana, and verify **v2** is not receiving traffic.

3.1. Retrieve the gateway URL where the service is exposed. Open a new terminal and execute the following command:

```
[student@workstation ~]$ GATEWAY_URL=$(oc get route istio-ingressgateway \
> -n istio-system -o jsonpath='{.spec.host}')
```

3.2. In the same terminal, execute this script to send a request to the **vertx-greet** service every 0.5 seconds:

```
[student@workstation ~]$ watch -p -n0.5 curl -s $GATEWAY_URL
```

Keep this script active until the end of the exercise.

3.3. Return to the previous terminal window. Find the URL to **Grafana** by examining the routes available in the **istio-namespace**. You can use the OpenShift console, or execute the following command:

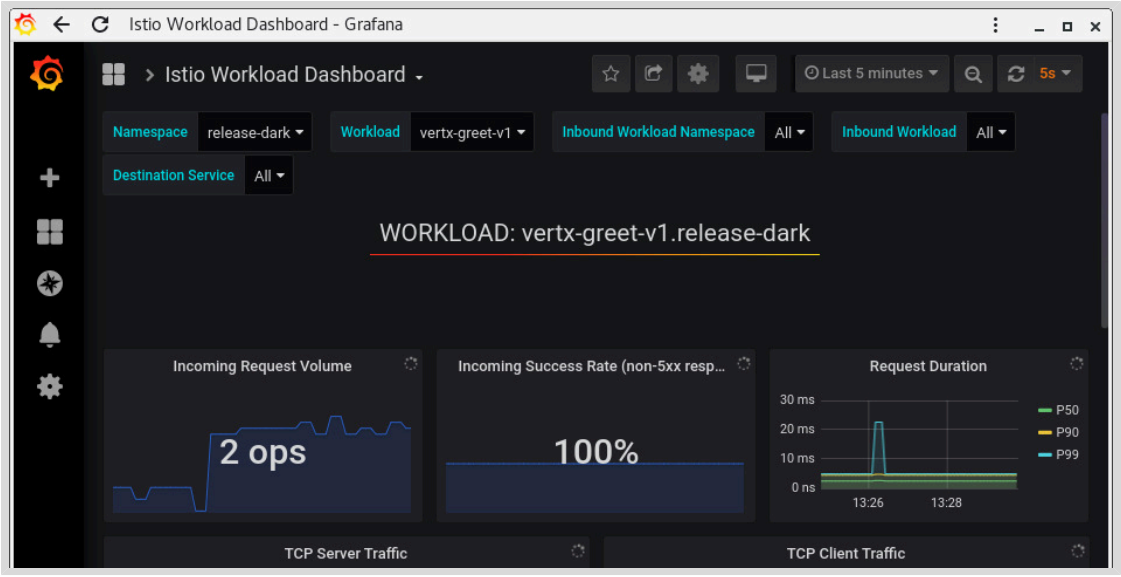
```
[student@workstation ~]$ GRAFANA_URL=$(oc get route grafana \
> -n istio-system -o jsonpath='https://{.spec.host}')
```

Open the URL in the Firefox web browser. Use the lessons learned in *Collecting Service Metrics*, and the instructions from *Guided Exercise: Collecting Service Metrics*, to log into **Grafana** using your developer credentials.

```
[student@workstation ~]$ firefox $GRAFANA_URL &
```

3.4. Go to the **Istio Workload Dashboard** by navigating to **Home** → **Istio** → **Istio Workload Dashboard**. In the dashboard, select the **release-dark** namespace

and the **vertx-greet-v1** workload. Note how the **Incoming Request Volume** indicates the service is receiving around two requests per second.



Leave this browser window open so that later you can review the amount of traffic received.

3.5. Validate that the state of the service changes with the requests. Execute the following command several times and see how the request number increases as the service receives requests. Note also that only the **v1** service is showing, indicating no traffic is sent to the **v2** service:

```
[student@workstation ~]$ oc get pods -o name | \
> xargs -L 1 oc logs --tail 1 -c vertx-greet
INFO: Attending greeting request #109 from vertx-greet-v1-f44b9976c-pkq44
[student@workstation ~]$
```

4. Configure OpenShift Service Mesh to mirror all requests sent to **v1** to **v2**. Visualize in Grafana, and verify **v2** is receiving the same amount of traffic. Validate that both versions change the state synchronously.

4.1. Update the **DestinationRule** resource to include the new version of the service as a **subset**. Edit the resource directly in the OpenShift console, or use the **oc edit DestinationRule vertx-greet -n release-dark** command to open the resource in your default editor. Add a subset named **v2** matching the **version: v2** label:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: vertx-greet
  labels:
    kiali_wizard: weighted_routing
spec:
  host: vertx-greet
  subsets:
  - labels:
```

```
      version: v1
    name: v1
  - labels:
      version: v2
    name: v2
```

4.2. Update the **`VirtualService`** to mirror traffic to the **`v2`** subset. In this way, a copy of all requests sent to the **`v1`** subset of the **`vertx-greet`** host are also sent to the **`v2`** subset. Use the OpenShift console, or the **`oc edit VirtualService vertx-greet -n release-dark`** command to edit the **`VirtualService`** resource:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: vertx-greet
  labels:
    kiali_wizard: weighted_routing
spec:
  hosts:
  - "*"
  gateways:
  - vertx-greet-gateway
  http:
  - route:
    - destination:
        host: vertx-greet
        subset: v1
        port:
          number: 8080
    mirror:
      host: vertx-greet
      subset: v2
```

4.3. Return to the browser window and observe that the amount of traffic sent to **`v1`** is roughly the same. Open a new browser tab to the same URL. Navigate back to the **`Istio Workload Dashboard`** and select the **`release-dark`** namespace. This time, select the **`vertx-greet-v2`** workload.

Review the **`Incoming Request Volume`** graph. Note the amount of traffic received by the **`v2`** version of the service is the same as the traffic received by the **`v1`** version.

4.4. Validate that both services change the state synchronously. Both services should update the request count on each request. Restart both services, so they start with the same initial state:

```
[student@workstation ~]$ oc delete pod --all -n release-dark
pod "vertx-greet-v1-f44b9976c-pkq44" deleted
pod "vertx-greet-v2-7849968c47-s2s5p" deleted
```

Wait several seconds for both services to restart, and then use the following command to check the status of both services. Run the command several times to validate that both services increase the request count, keeping their internal states synchronized:

```
[student@workstation ~]$ oc get pods -o name | \
> xargs -L 1 oc logs --tail 1 -c vertx-greet
May 12, 2020 12:52:06 PM io.vertx.greet.GreetServer lambda$0
INFO: Attending greeting request #55 from vertx-greet-v1-f44b9976c-pq457
May 12, 2020 12:52:07 PM io.vertx.greet.GreetServer lambda$0
INFO: Attending greeting request #55 from vertx-greet-v2-7849968c47-66fpg
```

5. Mirror 10% of the requests sent to **v1** to **v2**. Visualize in Grafana, and verify **v2** receives roughly one tenth of the traffic **v1** receives.

   5.1. Update the **VirtualService** resource to restrict traffic mirroring to 10% using the OpenShift console or the **oc edit VirtualService vertx-greet -n release-dark** command:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: vertx-greet
  labels:
    kiali_wizard: weighted_routing
spec:
  hosts:
  - "*"
  gateways:
  - vertx-greet-gateway
  http:
  - mirror:
      host: vertx-greet
      subset: v2
    mirror_percent: 10
    route:
    - destination:
        host: vertx-greet
        subset: v1
        port:
          number: 8080
```

   5.2. Return to your browser window. Observe that the traffic sent to **v2** has reduced to one tenth of the traffic received by **v2**. Verify that the traffic sent to **v1** is still the same as before, roughly two requests per second.

6. Accept the **v2** version of the service by transferring all traffic to this version. Update the **VirtualService** resource to route all traffic to the **v2** subset using the OpenShift console or the **oc edit VirtualService vertx-greet -n release-dark** command:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  creationTimestamp: "2020-05-12T10:30:01Z"
  generation: 4
```

```
    labels:
      kiali_wizard: weighted_routing
    name: vertx-greet
    namespace: release-dark
    resourceVersion: "20336884"
    selfLink: /apis/networking.istio.io/v1alpha3/namespaces/release-dark/
virtualservices/vertx-greet
    uid: 46be6a2b-1a89-4c98-9c5b-d342566329da
spec:
    gateways:
    - vertx-greet-gateway
    hosts:
    - '*'
    http:
    - route:
      - destination:
          host: vertx-greet
          port:
            number: 8080
          subset: v2
```

Confirm that the **v2** service is receiving all traffic by reviewing the **Incoming Request Volume** graph for both versions. The **v1** service receives no requests, while the **v2** service receives all requests.

7. Finalize the traffic generation process. In the window terminal running the **watch** command, press **Ctrl+C** to terminate the process. Then, you can close that terminal.

Finish

On the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab release-dark finish
```

This concludes the guided exercise.

Lab

Releasing Applications with OpenShift Service Mesh

Performance Checklist

In this lab, you will deploy different versions of a service using canary releases, and use service mesh mirroring to perform a dark launch.

Outcomes

You should be able to:

- Deploy a new version of a service using canary releases and redirect a percentage of traffic to the new version.
- Mirror traffic between two versions of a microservice using Red Hat OpenShift Service Mesh mirroring.

Before You Begin

To perform this lab, ensure you have:

- A configured and running OpenShift cluster.
- An installed and running OpenShift Service Mesh in the OpenShift cluster.
- The OpenShift client (oc) installed on workstation.

You will be using an application that simulates an online payment processor for an e-commerce application in this lab. The application has two microservices:

- **`gateway`**: Written in Java using the **`Quarkus`** framework. It simulates processing of payments using multiple payment gateways like banks, bitcoin, mobile wallets and more. This microservice is specific to this application and is not equivalent to the ingress gateway.
- **`payment`**: Written in Java using the **`Quarkus`** framework. The **`payment`** microservice acts as an API gateway and the single point of communication for traffic coming into the service mesh. It communicates with the **`gateway`** microservice to process payments.

The source code for the application is available in the **`payments`** folder in the GitHub repository at https://github.com/RedHatTraining/DO328-apps.

On the **`workstation`** machine, use the **`lab`** command to prepare your system for this exercise.

```
[student@workstation ~]$ lab release-mesh start
```

The **`lab release-mesh start`** command creates a new project called **`release-mesh`**, owned by the **`developer`** user (the value of the **`$RHT_OCP4_DEV_USER`** environment variable in your **`/usr/local/etc/ocp4.config`** file). It deploys an initial version of the **`payment`** and **`gateway`** microservices in this project.

You can examine the template that deploys the microservices in the **~/DO328/labs/release-mesh/app-deployment.yaml** file.

1. Log in to OpenShift as the **developer** user and inspect the deployed microservices in the **release-mesh** project. Verify that the **payment** and **gateway** microservices are deployed and running.

   Remember to set the environment variables in the **/usr/local/etc/ocp4.config** file.

2. Test the initial version (**v1**) of the **payment** and **gateway** microservices. Invoke the **/pay/{amount}** endpoint relative to the service mesh gateway URL, where **amount** is an integer value representing a random amount.

   You can also use the **test_app.py** script in the **/home/student/DO328/labs/release-mesh** folder to test the application. This script takes the service mesh gateway URL as an argument (**./test_app.py $GW_URL**). The script sends 50 requests and prints the version, and percentage of traffic processed by each version of the **payment** microservice. It also prints the error rate (if any) during request processing.

   Verify that you see the response only from **v1** of the **payment** microservice. Verify that all payment requests are processed by **v1** of the **gateway** microservice.

Note

You cannot directly invoke the **gateway** microservice. Inspect the log output from the **gateway** microservice to verify the version of the microservice that is processing the payments.

3. The development team is ready to deploy and test **v2** of the **payment** microservice. This version has several new enhancements that must be tested in a production environment.

   Deploy **v2** of the **payment** microservice. A prebuilt, publicly available container image is provided in the Quay.io registry at the URL **quay.io/redhattraining/ossm-payment:2.0**.

4. Route **10%** of all traffic to **v2** of the **payment** microservice. **v1** of the **payment** microservice should still handle most of the traffic (**90%**).

Note

A virtual service resource for the **payment** microservice was created by the lab start script.

   Use the **test_app.py** script to verify the traffic split.

5. The developers are also working on a new version of the **gateway** microservice. The developers are not yet ready to deploy the new version of the **gateway** microservice to

process payments. They want to test this new version with real production data and monitor the performance characteristics of the microservice.

Deploy **v2** of the **gateway** microservice. A prebuilt, publicly available container image is provided to you in the Quay.io registry at the URL **quay.io/redhattraining/ossm-gateway:2.0**.

Enable mirroring of traffic from **v1** of the **gateway** microservice to **v2** of the **gateway** microservice. **v1** of the **gateway** microservice should still exclusively process all transactions.

Use the **app_test.py** script to test the changes and verify that you do not see any errors. Inspect the log output from both versions of the **gateway** microservice to verify that all transactions sent to **v1** are mirrored to **v2**.

Note

A virtual service and destination rule resource has already been created by the lab start script.

6. Return to the home directory.

```
[student@workstation release-mesh]$ cd ~
[student@workstation ~]$
```

Evaluation

Grade your work by running the **lab release-mesh grade** command from your **workstation** machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab release-mesh grade
```

Finish

On the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab release-mesh finish
```

This concludes the lab.

Solution

Releasing Applications with OpenShift Service Mesh

Performance Checklist

In this lab, you will deploy different versions of a service using canary releases, and use service mesh mirroring to perform a dark launch.

Outcomes

You should be able to:

- Deploy a new version of a service using canary releases and redirect a percentage of traffic to the new version.
- Mirror traffic between two versions of a microservice using Red Hat OpenShift Service Mesh mirroring.

Before You Begin

To perform this lab, ensure you have:

- A configured and running OpenShift cluster.
- An installed and running OpenShift Service Mesh in the OpenShift cluster.
- The OpenShift client (oc) installed on workstation.

You will be using an application that simulates an online payment processor for an e-commerce application in this lab. The application has two microservices:

- **`gateway`**: Written in Java using the **`Quarkus`** framework. It simulates processing of payments using multiple payment gateways like banks, bitcoin, mobile wallets and more. This microservice is specific to this application and is not equivalent to the ingress gateway.
- **`payment`**: Written in Java using the **`Quarkus`** framework. The **`payment`** microservice acts as an API gateway and the single point of communication for traffic coming into the service mesh. It communicates with the **`gateway`** microservice to process payments.

The source code for the application is available in the **`payments`** folder in the GitHub repository at https://github.com/RedHatTraining/DO328-apps.

On the **`workstation`** machine, use the **`lab`** command to prepare your system for this exercise.

```
[student@workstation ~]$ lab release-mesh start
```

The **`lab release-mesh start`** command creates a new project called **`release-mesh`**, owned by the **`developer`** user (the value of the **`$RHT_OCP4_DEV_USER`** environment variable in your **`/usr/local/etc/ocp4.config`** file). It deploys an initial version of the **`payment`** and **`gateway`** microservices in this project.

You can examine the template that deploys the microservices in the **~/DO328/labs/release-mesh/app-deployment.yaml** file.

1. Log in to OpenShift as the **developer** user and inspect the deployed microservices in the **release-mesh** project. Verify that the **payment** and **gateway** microservices are deployed and running.

   Remember to set the environment variables in the **/usr/local/etc/ocp4.config** file.

   1.1. Run the following command to load the environment variables defined in the guided exercise where you created the lab environment:

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

   1.2. Log in to OpenShift:

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_API}
Login successful.
...output omitted...
```

   1.3. Set the current project to **release-mesh**:

```
[student@workstation ~]$ oc project release-mesh
Now using project "release-mesh" on server ...output omitted...
```

   1.4. Verify that the pods for the initial version(**v1**) of the **payment** and **gateway** microservices are deployed, and in a **Running** state:

```
[student@workstation ~]$ oc get pods
NAME                          READY   STATUS    RESTARTS   AGE
gateway-v1-5484d6fb59-8vrxf   2/2     Running   0          111s
payment-v1-d74848855-whncc    2/2     Running   0          111s
```

2. Test the initial version (**v1**) of the **payment** and **gateway** microservices. Invoke the **/pay/{amount}** endpoint relative to the service mesh gateway URL, where **amount** is an integer value representing a random amount.

   You can also use the **test_app.py** script in the **/home/student/DO328/labs/release-mesh** folder to test the application. This script takes the service mesh gateway URL as an argument (**./test_app.py $GW_URL**). The script sends 50 requests and prints the version, and percentage of traffic processed by each version of the **payment** microservice. It also prints the error rate (if any) during request processing.

   Verify that you see the response only from **v1** of the **payment** microservice. Verify that all payment requests are processed by **v1** of the **gateway** microservice.

Note

You cannot directly invoke the **gateway** microservice. Inspect the log output from the **gateway** microservice to verify the version of the microservice that is processing the payments.

2.1. Run the **`oc get route`** command to gather the service mesh gateway URL.

```
[student@workstation ~]$ GW_URL=$(oc get route istio-ingressgateway \
> -n istio-system -o jsonpath='{.spec.host}')
```

2.2. Test the application. Make a request to the **`/pay/{amount}`** endpoint relative to the gateway URL (GW_URL).

```
[student@workstation ~]$ curl $GW_URL/pay/10
[payment-v1] OK. Transaction id is 6493
```

Note the response from **`v1`** of the **`payment`** microservice. Your transaction id may be different.

2.3. Test the application using the **`test_app.py`** script.

Do not add any relative endpoints to the gateway URL. It automatically generates a random amount when invoking the **`payment`** microservice.

```
[student@workstation ~]$ cd ~/DO328/labs/release-mesh
[student@workstation release-mesh]$ ./test_app.py $GW_URL
Canary Release Test
Sending 50 requests to istio-ingressgateway ...output omitted...
[payment-v1] OK. Transaction id is 9918
[payment-v1] OK. Transaction id is 8137
...output omitted...
[payment-v1] OK. Transaction id is 7685

#### Stats ####

Total requests: 50
* '[payment-v1] OK' responses: 50 (100.0%)
* Errors: 0 (0.0%)
```

Note that **`payment-v1`** is used to process all requests.

2.4. Use the **`oc logs`** command to view the logs for the **`gateway`** microservice. Get the pod name from the **`oc get pods`** command.

```
[student@workstation release-mesh]$ oc logs gateway-v1-5484d6fb59-8vrxf \
> -c gateway-v1
...output omitted...
Processing payment for $10 through gateway-v1...
Processing payment for $0 through gateway-v1...
Processing payment for $1 through gateway-v1...
Processing payment for $2 through gateway-v1...
...output omitted...
Processing payment for $48 through gateway-v1...
Processing payment for $49 through gateway-v1...
```

The logging output shows only the log message and has been trimmed to fit the width of the page. You will see time stamps, the class name, and the logging level printed before the log messages in your console.

Note that **`gateway-v1`** is processing payments for all requests.

3. The development team is ready to deploy and test **v2** of the **payment** microservice. This version has several new enhancements that must be tested in a production environment.

Deploy **v2** of the **payment** microservice. A prebuilt, publicly available container image is provided in the Quay.io registry at the URL **quay.io/redhattraining/ossm-payment:2.0**.

3.1. You can copy and create the deployment YAML resource file for **v2** from the **app-deployment.yaml** file that was originally used to deploy the microservice.

The full deployment YAML resource is also available in the **/home/student/DO328/solutions/release-mesh/payment-v2-deploy.yaml** file.

The YAML resource snippet to deploy **v2** is as follows:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    app: payment
    version: v2
  name: payment-v2
spec:
  replicas: 1
  selector:
    matchLabels:
      app: payment
      version: v2
  template:
    metadata:
      labels:
        app: payment
        version: v2
      annotations:
        sidecar.istio.io/inject: "true"
    spec:
      containers:
      - name: payment-v2
        image: quay.io/redhattraining/ossm-payment:2.0
        imagePullPolicy: Always
        ports:
...output omitted...
```

Deploy **v2** using the **oc create** command.

```
[student@workstation release-mesh]$ oc create -f payment-v2-deploy.yaml
deployment.apps/payment-v2 created
```

3.2. Verify that **v2** of the **payment** microservice is deployed and in **Running** state.

```
[student@workstation release-mesh]$ oc get pods
NAME                          READY   STATUS    RESTARTS   AGE
gateway-v1-5484d6fb59-8vrxf   2/2     Running   0          46m
payment-v1-d74848855-whncc    2/2     Running   0          46m
payment-v2-64d475cb84-wc7cc   2/2     Running   0          22s
```

4. Route **10%** of all traffic to **v2** of the **payment** microservice. **v1** of the **payment** microservice should still handle most of the traffic (**90%**).

Note

A virtual service resource for the **payment** microservice was created by the lab start script.

Use the **test_app.py** script to verify the traffic split.

4.1. Create a destination rule resource for the **payment** microservice.

The full destination rule YAML resource is also available in the **/home/student/DO328/solutions/release-mesh/payment-dest-rule.yaml** file.

The YAML resource snippet to create the destination rule is as follows:

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

Create the destination rule using the **oc create** command.

```
[student@workstation release-mesh]$ oc create -f payment-dest-rule.yaml
destinationrule.networking.istio.io/payment-dr created
```

4.2. Edit the **payment-vs** virtual service resource.

```
[student@workstation release-mesh]$ oc edit vs payment-vs
```

Split the traffic between **v1** and **v2**.

```
...output omitted...
spec:
  gateways:
  - payment-api-gw
  hosts:
  - '*'
  http:
  - route:
    - destination:
        host: payment
        port:
          number: 8080
```

```
        subset: v1
      weight: 90
    - destination:
        host: payment
        port:
          number: 8080
        subset: v2
      weight: 10
```

Save your changes.

4.3. Test the application using the **test_app.py** script.

```
[student@workstation release-mesh]$ ./test_app.py $GW_URL
...output omitted...
#### Stats ####

Total requests: 50
* '[payment-v1] OK' responses: 48 (96.0%)
* '[payment-v2] OK' responses: 2 (4.0%)
* Errors: 0 (0.0%)
```

Note that the traffic is split between **v1** and **v2** (approximately 90/10 ratio). The split may not be exact. The service mesh tries to maintain the ratio between versions as much as possible.

5. The developers are also working on a new version of the **gateway** microservice. The developers are not yet ready to deploy the new version of the **gateway** microservice to process payments. They want to test this new version with real production data and monitor the performance characteristics of the microservice.

Deploy **v2** of the **gateway** microservice. A prebuilt, publicly available container image is provided to you in the Quay.io registry at the URL **quay.io/redhattraining/ossm-gateway:2.0**.

Enable mirroring of traffic from **v1** of the **gateway** microservice to **v2** of the **gateway** microservice. **v1** of the **gateway** microservice should still exclusively process all transactions.

Use the **app_test.py** script to test the changes and verify that you do not see any errors. Inspect the log output from both versions of the **gateway** microservice to verify that all transactions sent to **v1** are mirrored to **v2**.

Note

A virtual service and destination rule resource has already been created by the lab start script.

5.1. You can copy and create the deployment YAML resource file for **v2** from the **app-deployment.yaml** file that was originally used to deploy the microservice.

The full deployment YAML resource is also available in the **/home/student/DO328/solutions/release-mesh/gateway-v2-deploy.yaml** file.

The YAML resource snippet to deploy **v2** is as follows:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
```

```
  labels:
    app: gateway
    version: v2
  name: gateway-v2
spec:
  replicas: 1
  selector:
    matchLabels:
      app: gateway
      version: v2
  template:
    metadata:
      labels:
        app: gateway
        version: v2
      annotations:
        sidecar.istio.io/inject: "true"
    spec:
      containers:
      - name: gateway-v2
        image: quay.io/redhattraining/ossm-gateway:2.0
        imagePullPolicy: Always
        ports:
...output omitted...
```

Deploy **v2** using the **oc create** command.

```
[student@workstation release-mesh]$ oc create -f gateway-v2-deploy.yaml
deployment.apps/gateway-v2 created
```

5.2. Verify that **v2** of the **gateway** microservice is deployed and in **Running** state.

```
[student@workstation release-mesh]$ oc get pods
NAME                          READY   STATUS    RESTARTS   AGE
gateway-v1-5484d6fb59-8vrxf   2/2     Running   0          64m
gateway-v2-f4dc796bd-c72zc    2/2     Running   0          49s
payment-v1-d74848855-whncc    2/2     Running   0          64m
payment-v2-64d475cb84-wc7cc   2/2     Running   0          18m
```

5.3. Edit the destination rule for the **gateway** microservice and add details for **v2**. You can get the name of the destination rule resource using the **oc get dr** command.

```
[student@workstation release-mesh]$ oc edit dr gateway-dr
```

Add the details for **v2** as follows:

```
...output omitted...
spec:
  host: gateway
  subsets:
  - labels:
      version: v1
```

```
    name: v1
  - labels:
      version: v2
    name: v2
```

Save your changes.

5.4. Edit the virtual service for the **gateway** microservice and enable mirroring. You can get the name of the virtual service resource using the **oc get vs** command.

```
[student@workstation release-mesh]$ oc edit vs gateway-vs
```

Enable mirroring as follows:

```
...output omitted...
spec:
  gateways:
  - payment-api-gw
  hosts:
  - gateway
  http:
  - mirror:
      host: gateway
      subset: v2
    route:
    - destination:
        host: gateway
        subset: v1
      weight: 100
```

Save your changes.

5.5. Test the application using the **test_app.py** script.

```
[student@workstation release-mesh]$ ./test_app.py $GW_URL
...output omitted...
#### Stats ####

Total requests: 50
* '[payment-v1] OK' responses: 44 (88.0%)
* '[payment-v2] OK' responses: 6 (12.0%)
* Errors: 0 (0.0%)
```

Note that the traffic split between **v1** and **v2** remains the same (approximately 90/10 ratio) as configured earlier.

5.6. Use the **oc logs** command to view the logs for both versions of the **gateway** microservice. Get the pod names from the **oc get pods** command.

```
[student@workstation release-mesh]$ oc logs gateway-v1-5484d6fb59-8vrxf \
> -c gateway-v1
...output omitted...
Processing payment for $0 through gateway-v1...
Processing payment for $1 through gateway-v1...
Processing payment for $2 through gateway-v1...
```

```
...output omitted...
Processing payment for $48 through gateway-v1...
Processing payment for $49 through gateway-v1...
```

```
[student@workstation release-mesh]$ oc logs gateway-v2-f4dc796bd-c72zc \
> -c gateway-v2
...output omitted...
Processing payment for $0 through gateway-v2...
Processing payment for $1 through gateway-v2...
Processing payment for $2 through gateway-v2...
...output omitted...
Processing payment for $48 through gateway-v2...
Processing payment for $49 through gateway-v2...
```

The logging output shows only the log message, and has been trimmed to fit the width of the page. Notice the time stamps, the class name, and the logging level printed before the log messages in your console.

Note that all requests sent to **`gateway-v1`** are mirrored to **`gateway-v2`**.

6. Return to the home directory.

```
[student@workstation release-mesh]$ cd ~
[student@workstation ~]$
```

Evaluation

Grade your work by running the **`lab release-mesh grade`** command from your **`workstation`** machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab release-mesh grade
```

Finish

On the **`workstation`** machine, use the **`lab`** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab release-mesh finish
```

This concludes the lab.

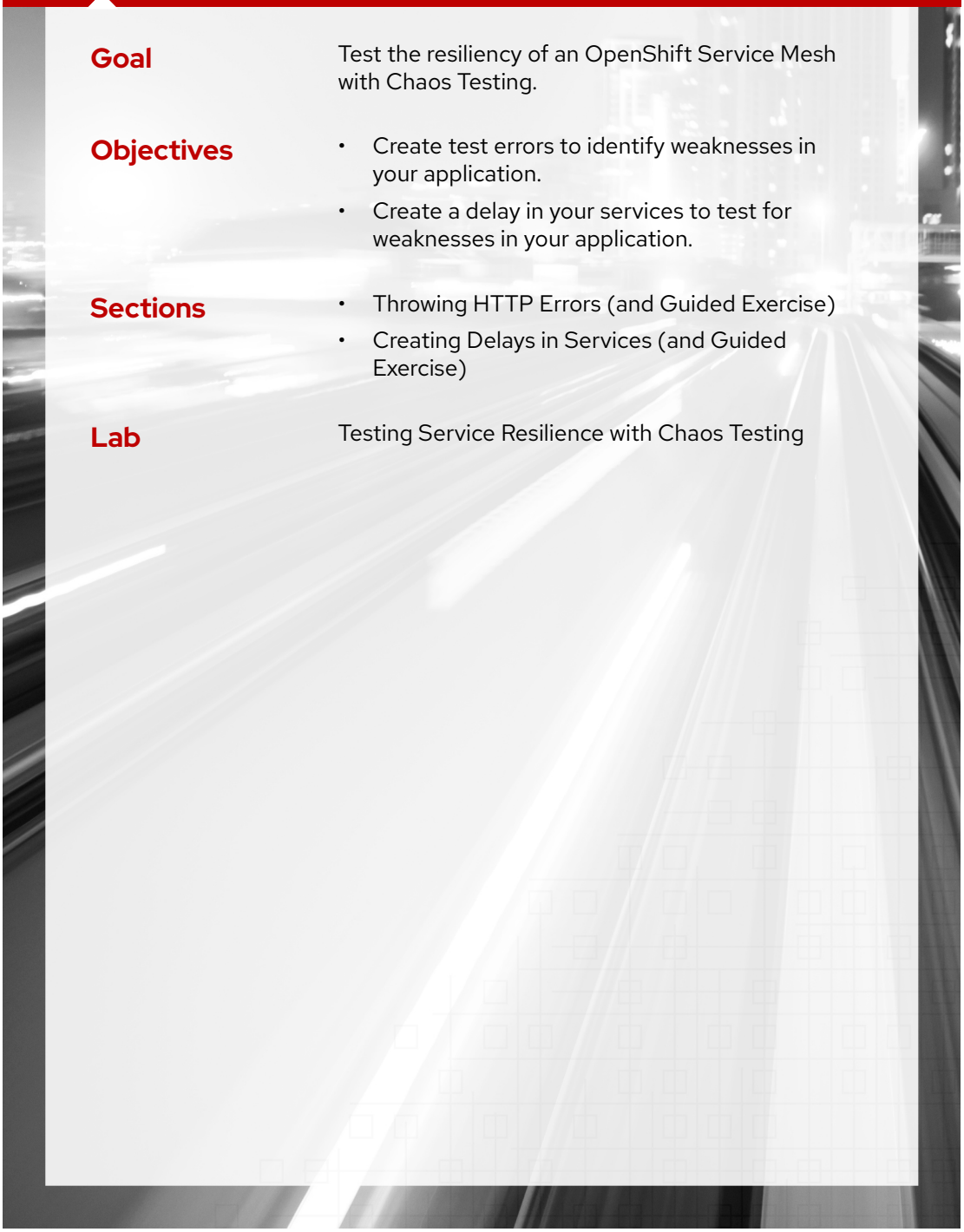
Summary

In this chapter, you learned:

- Red Hat OpenShift Service Mesh supports canary releases. You can control the percentage of traffic sent to new versions of microservices.
- You can configure Traffic routing to specific versions based on URL and HTTP header matching.
- Kiali can be used to visualize the traffic flow in the service mesh and to configure weighted routing.
- Red Hat OpenShift Service Mesh supports traffic mirroring to perform dark launches. You can use this feature to test newer versions of your microservices without impacting currently running services in production.

Chapter 6

Testing Service Resilience with Chaos Testing



Goal	Test the resiliency of an OpenShift Service Mesh with Chaos Testing.
Objectives	• Create test errors to identify weaknesses in your application. • Create a delay in your services to test for weaknesses in your application.
Sections	• Throwing HTTP Errors (and Guided Exercise) • Creating Delays in Services (and Guided Exercise)
Lab	Testing Service Resilience with Chaos Testing

Throwing HTTP Errors

Objectives

After completing this section, you should be able to create test errors to identify weaknesses in your application.

Chaos Testing

Although microservice-based applications are highly scalable, they also suffer from common problems or fallacies associated with distributed computing There are eight fallacies of distibuted computing. In this lecure, we will focus on two of the most common in general use cases.

- The network is reliable.
- There is zero latency.

These two assumptions must be addressed because network instability occurs and, if not dealt with, can create unpredictable application behavior. For example, presenting unexpected errors to the user, or silently ignoring errors without sending any notifications. To validate how applications respond to network instability, you can introduce chaos to simulate network instability.

Chaos Testing is the process of testing a microservices-based application in production or in an environment similar to production by introducing random errors to verify that the steps taken to handle these problems are correct.

Netflix coined the term Chaos Testing to refer to the techniques they used to test their systems in production, aimed at verifying that all their complex applications behaved as expected in the event of network errors. Netflix engineers resorted to these techniques because they knew that network errors and instability are inevitable.

You can use service mesh traffic management capabilities to introduce latency spikes or connection errors in your application so that you can perform chaos testing.

Throwing HTTP Errors

Use the **`HTTPFaultInjection.Abort`** object on path **`spec.http.fault.abort`** to inject errors into the **`VirtualService`**.

The Abort object requires two configuration values:

httpStatus
HTTP status code returned on abort

percentage
Percentage of total request to abort

The **`httpStatus`** value is the literal number representing the HTTP status code. If you want to return an Internal Server Error use:

```
httpStatus: 500
```

The **`percentage`** is configured using a **`double`** value, ranging from 0.0 for a 0% and 100.0 for 100%.

For example, if you want to drop 20% of the connections to **`example-svc`** from other services and return the **`Bad Request`** error, you can use a code like this in the Virtual Service:

```
apiVersion: networking.istio.io/v1alpha3
  kind: VirtualService
  metadata:
    name: example-vs
  spec:
    hosts:
    - example-svc
    http:
    - route:
      - destination:
          host: example-svc
          subset: v1
      fault: (1)
        abort: (2)
          percentage: (3)
            value: 20.0
          httpStatus: 400(4)
```

1. The **`HTTPFaultInjection`** configuration object responsible for all the faults injected into the service.
2. The **`HTTPFaultInjection.Abort`** configuration object responsible for the error injection configuration.
3. Percentage of the connections to abort.
4. The HTTP status code to return on abort.

When testing your application with HTTP errors, the percentage of failed requests can be lower than your configured value. This is because the virtual service resources contain automatic retries of failed requests.

For more information about retries, see *Configuring Retry*.

References

Istio 1.4 / Fault Injection
https://archive.istio.io/v1.4/docs/tasks/traffic-management/fault-injection/

Istio 1.4 / Virtual Service / HTTPFaultInjection.Abort
https://archive.istio.io/v1.4/docs/reference/config/networking/virtual-service/#HTTPFaultInjection-Abort

Fallacies of distributed computing
https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing

Chaos engineering
https://en.wikipedia.org/wiki/Chaos_engineering

Guided Exercise

Throwing HTTP Errors

In this exercise, you will set up Services Mesh to throw errors in an application route and verify these errors.

Outcomes

You should be able to set up a route to throw HTTP errors.

Before You Begin

To perform this exercise, ensure you have access to:

- A configured and running OpenShift cluster.
- A installed and running OpenShift Service Mesh in the OpenShift cluster.
- The OpenShift CLI (**/usr/local/bin/oc**).

As the **student** user on the **workstation** machine, use the **lab** command to prepare your system for this exercise.

This command ensures that the application is deployed in the cluster to test its behavior.

The source code is in the Git repository at `https://github.com/RedHatTraining/DO328-apps/` in the **customer**, **preference**, and **recommendation** directories.

```
[student@workstation ~]$ lab chaos-error start
```

1. Log in to OpenShift and verify that the sample project deployed successfully.

1.1. Run the following command to load the environment variables created in the Verifying OpenShift Credentials guided exercise:

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

1.2. Log in to OpenShift:

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_API}
Login successful.
...output omitted...
```

1.3. Change to project **chaos-error**:

```
[student@workstation ~]$ oc project chaos-error
Now using project "chaos-error" on server ...
```

1.4. Verify that pods are ready:

```
[student@workstation ~]$ oc get pods
NAME                              READY   STATUS    RESTARTS   AGE
customer-6948b8b959-cl5zf         2/2     Running   0          11s
preference-6d5d86cb79-9cjkv       2/2     Running   0          11s
recommendation-69db8d6c48-wrkc6   2/2     Running   0          11s
```

1.5. Save the **`ingress-gateway`** route host name with the **`/chaos`** endpoint into a variable:

```
[student@workstation ~]$ INGRESS_URL=$(oc get route -n istio-system \
> istio-ingressgateway -o jsonpath='{.spec.host}'/chaos)
```

1.6. Verify that the service responds using the **`INGRESS_URL`** variable:

```
[student@workstation ~]$ for i in {1..10};do curl $INGRESS_URL; done
customer => preference => recommendation v1 from 'f11b097f1dd0': 1
customer => preference => recommendation v1 from 'f11b097f1dd0': 2
customer => preference => recommendation v1 from 'f11b097f1dd0': 3
customer => preference => recommendation v1 from 'f11b097f1dd0': 4
customer => preference => recommendation v1 from 'f11b097f1dd0': 5
customer => preference => recommendation v1 from 'f11b097f1dd0': 6
customer => preference => recommendation v1 from 'f11b097f1dd0': 7
customer => preference => recommendation v1 from 'f11b097f1dd0': 8
customer => preference => recommendation v1 from 'f11b097f1dd0': 9
customer => preference => recommendation v1 from 'f11b097f1dd0': 10
```

2. Edit the **`recommendation VirtualService`** and configure it to always throw the HTTP error 500:

```
[student@workstation ~]$ oc edit virtualservice recommendation
```

Add this after the route section under the http block with this fault section:

```
    fault:
      abort:
        httpStatus: 500
        percentage:
          value: 50.0
```

The spec section should look like:

```
spec:
  hosts:
  - recommendation
  http:
  - route:
    - destination:
        host: recommendation
    fault:
      abort:
```

```
        httpStatus: 500
        percentage:
          value: 50.0
```

Save the file to update of the **VirtualService**.

If you have trouble editing the VirtualService, you can find the new VirtualService definition in the file **DO328/solutions/chaos-error/vs-recommendation-error.yml**. Apply this definition using the following command:

```
[student@workstation ~]$ oc replace -f DO328/solutions/chaos-error/vs-
recommendation-error.yml
virtualservice.networking.istio.io/recommendation replaced
```

3. To verify that the **recommendation** application is throwing random errors, repeat the previous service verification:

```
[student@workstation ~]$ for i in {1..10};do curl $INGRESS_URL; done
customer => preference => recommendation v1 from 'f11b097f1dd0': 11
customer => Error: 503 - preference => Error: 500 - fault filter abort
customer => Error: 503 - preference => Error: 500 - fault filter abort
customer => preference => recommendation v1 from 'f11b097f1dd0': 14
customer => preference => recommendation v1 from 'f11b097f1dd0': 15
customer => Error: 503 - preference => Error: 500 - fault filter abort
customer => Error: 503 - preference => Error: 500 - fault filter abort
customer => Error: 503 - preference => Error: 500 - fault filter abort
customer => preference => recommendation v1 from 'f11b097f1dd0': 19
customer => preference => recommendation v1 from 'f11b097f1dd0': 20
```

Note

By default, Istio retries each request twice. To clearly show how abort injections result in errors, retries are deactivated in this exercise.

If no errors are thrown in the first ten requests, then redo the test to see the errors.

Finish

On the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab chaos-error finish
```

This concludes the guided exercise.

Creating Delays in Services

Objectives

After completing this section, you should be able to create a delay in your services to test for weaknesses in your application.

Creating Delays in Services

The second distributed computing fallacy is network latency, which can spike at any moment. Similar to network errors, network latency must be tested in your application to avoid unexpected errors. During chaos testing you can inject artificial delays into services to simulate latency, verifying that the application handles these problems gracefully.

You can inject delay errors independently or in parallel with connection errors. Red Hat recommends that you perform both types of tests.

To inject delays in a **`VirtualService`** resource, use the **`HTTPFaultInjection.Delay`** object inside the **`HTTPFaultInjection`** configuration object.

The **`Delay`** object requires two configuration values:

fixedDelay
Delay to add to the connection.

percentage
Percentage of total requests into which the delay is injected.

You can declare the **`fixedDelay`** value in hours, minutes, seconds, and milliseconds (h/m/s/ms).

```
fixedDelay: 1h
```

The **`percentage`** value is a **`double`** value, ranging from 0.0 for a 0% and 100.0 for 100%.

For example, to add a 400 milliseconds delay to 10% of connections to the **`example-svc`** service, you can use the following:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: example-vs
spec:
  hosts:
  - example-svc
  http:
  - route:
    - destination:
        host: example-svc
        subset: v1
    fault: (1)
      delay: (2)
```

```
        percentage: 3
          value: 10.0
        fixedDelay: 400ms 4
```

1. The **`HTTPFaultInjection`** configuration object responsible for injecting faults into the service.
2. The **`HTTPFaultInjection.Delay`** configuration object responsible for the delay injection configuration.
3. Percentage of the connections to delay.
4. Amount of time to delay the connection.

References

Istio 1.4 / Fault Injection
https://archive.istio.io/v1.4/docs/tasks/traffic-management/fault-injection/

Istio 1.4 / Virtual Service / HTTPFaultInjection.Delay
https://archive.istio.io/v1.4/docs/reference/config/networking/virtual-service/#HTTPFaultInjection-Delay

Guided Exercise

Creating Service Delays

In this exercise, you will set up Services Mesh to add and verify delays in an application route.

Outcomes

You should be able to set up a delay in a route.

Before You Begin

To perform this exercise, ensure you have access to:

- A configured and running OpenShift cluster.
- A installed and running OpenShift Service Mesh in the OpenShift cluster.
- The OpenShift CLI (**`/usr/local/bin/oc`**).

As the **`student`** user on the **`workstation`** machine, use the **`lab`** command to prepare your system for this exercise.

This command ensures that the microservices application is deployed in the cluster to test its behavior.

The source code is in the Git repository at `https://github.com/RedHatTraining/DO328-apps/` in the **`customer`**, **`preference`**, and **`recommendation`** directories.

```
[student@workstation ~]$ lab chaos-delay start
```

1. Log in to OpenShift and verify that the sample project is successfully deployed.

   1.1. Run the following command to load the environment variables created in the Verifying OpenShift Credentials guided exercise:

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

   1.2. Log in to OpenShift:

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_API}
Login successful.
...output omitted...
```

   1.3. Change to project **`chaos-delay`**:

```
[student@workstation ~]$ oc project chaos-delay
Now using project "chaos-delay" on server ...
```

1.4. Verify that pods are ready:

```
[student@workstation ~]$ oc get pods
NAME                              READY   STATUS    RESTARTS   AGE
customer-69d5499fc4-h77bx         2/2     Running   0          11s
preference-558cf4f584-9bpd5       2/2     Running   0          11s
recommendation-c495d86d7-4h8rf    2/2     Running   0          11s
```

1.5. Save the **`ingress-gateway`** route host name with the **`/delay`** endpoint into a variable:

```
[student@workstation ~]$ INGRESS_URL=$(oc get route -n istio-system \
> istio-ingressgateway -o jsonpath='{.spec.host}'/delay)
```

1.6. Verify that the service responds using the **`INGRESS_URL`** variable:

```
[student@workstation ~]$ for i in {1..10};do curl -m 2 $INGRESS_URL; done
customer => preference => recommendation v1 from 'f11b097f1dd0': 1
customer => preference => recommendation v1 from 'f11b097f1dd0': 2
customer => preference => recommendation v1 from 'f11b097f1dd0': 3
customer => preference => recommendation v1 from 'f11b097f1dd0': 4
customer => preference => recommendation v1 from 'f11b097f1dd0': 5
customer => preference => recommendation v1 from 'f11b097f1dd0': 6
customer => preference => recommendation v1 from 'f11b097f1dd0': 7
customer => preference => recommendation v1 from 'f11b097f1dd0': 8
customer => preference => recommendation v1 from 'f11b097f1dd0': 9
customer => preference => recommendation v1 from 'f11b097f1dd0': 10
```

2. Edit the **`recommendation VirtualService`** and configure it to add a delay of 7 seconds:

```
[student@workstation ~]$ oc edit virtualservice recommendation
```

Add this after the route section under the http block with this fault section:

```
    fault:
      delay:
        percentage:
          value: 50
        fixedDelay: 7s
```

The spec section should look like:

```
spec:
  hosts:
  - recommendation
  http:
  - route:
    - destination:
        host: recommendation
    fault:
      delay:
```

```
        percentage:
          value: 50
        fixedDelay: 7s
```

Save the file to perform the update of the **`VirtualService`**.

If you have any trouble editing the VirtualService, you can find the new VirtualService definition in the file **`DO328/solutions/chaos-delay/vs-recommendation-delayed.yml`**. Apply this definition with the following command:

```
[student@workstation ~]$ oc replace -f DO328/solutions/chaos-delay/vs-
recommendation-delayed.yml
virtualservice.networking.istio.io/recommendation replaced
```

3. To verify that the **`recommendation`** service is disrupting the flow of the application, repeat the previous verification with a timeout of 2 seconds:

```
[student@workstation ~]$ for i in {1..10};do curl -m 2 $INGRESS_URL; done
customer => preference => recommendation v1 from 'f11b097f1dd0': 1
curl: (28) Operation timed out after 2001 milliseconds with 0 bytes received
curl: (28) Operation timed out after 2001 milliseconds with 0 bytes received
curl: (28) Operation timed out after 2001 milliseconds with 0 bytes received
customer => preference => recommendation v1 from 'f11b097f1dd0': 5
curl: (28) Operation timed out after 2001 milliseconds with 0 bytes received
curl: (28) Operation timed out after 2001 milliseconds with 0 bytes received
customer => preference => recommendation v1 from 'f11b097f1dd0': 8
customer => preference => recommendation v1 from 'f11b097f1dd0': 9
customer => preference => recommendation v1 from 'f11b097f1dd0': 10
```

Finish

On the **`workstation`** machine, use the **`lab`** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab chaos-delay finish
```

This concludes the guided exercise.

Lab

Testing Service Resilience with Chaos Testing

Performance Checklist

In this lab, you will simulate network issues to test application resilience and graceful handling of network issues.

Outcomes

You should be able to use OpenShift Service Mesh to simulate network failures and delays.

Before You Begin

To perform this exercise, ensure you have access to:

- A configured and running OpenShift cluster.
- A installed and running OpenShift Service Mesh in the OpenShift cluster.
- The OpenShift CLI (**/usr/local/bin/oc**).

As the **student** user on the **workstation** machine, use the **lab** command to prepare your system for this lab.

The **lab** command deploys the exchange application into your Red Hat OpenShift cluster. The source code is in the Git repository at https://github.com/RedHatTraining/DO328-apps in the **exchange-application** directory.

The exchange application consists of the following services:

- Frontend
- Currency
- Exchange
- History

You can examine the services in the **chaos-mesh** project. The application is available using the **istio-ingressgateway** route at the **/frontend** endpoint.

```
[student@workstation ~]$ lab chaos-mesh start
```

1. Log in to OpenShift and verify that the application is ready.

   1.1. Run the following command to load the environment variables created in the Verifying OpenShift Credentials guided exercise:

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

1.2. Log in to OpenShift:

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_API}
Login successful.
...output omitted...
```

1.3. Change to project **chaos-mesh**:

```
[student@workstation ~]$ oc project chaos-mesh
Now using project "chaos-mesh" on server ...
```

1.4. Verify that pods are in the **Running** state:

```
[student@workstation ~]$ oc get pods
NAME                        READY   STATUS    RESTARTS   AGE
currency-566cddc8c6-jtd5g   2/2     Running   0          13s
exchange-66b78bf65c-s72gv   2/2     Running   0          13s
frontend-5648fbb85f-td5dg   2/2     Running   0          13s
history-54b5c9d476-rk4pd    2/2     Running   0          13s
```

1.5. Save the **ingress-gateway** route hostname with the **/frontend** endpoint into a variable:

```
[student@workstation ~]$ FRONTEND=$(oc get route -n istio-system \
> istio-ingressgateway -o jsonpath='{.spec.host}'/frontend)
```

1.6. Verify that the application responds using the **FRONTEND** variable:

```
[student@workstation ~]$ firefox $FRONTEND
```

1.7. Verify that the **Historical Data** page of the application works. Open the page:

```
[student@workstation ~]$ firefox $FRONTEND/history
```

On the **Historical Data** page, click **Submit**.

1.8. Verify that the **Exchange** page of the application works. Open the page:

```
[student@workstation ~]$ firefox $FRONTEND/exchange
```

On the **Exchange** page, click **Submit**.

2. Introduce a delay fault to the **exchange-vservice** virtual service resource with the following parameters:

- Delay time: **10s**.
- The fault should influence **20%** of all requests to the **exchange-vservice** virtual service.

With the delay fault in place, test how the **frontend** application responds to large delay times.

3. Introduce an abort fault to the **exchange-vservice** virtual service resource with the following parameters:

   - HTTP error code: **500**.

     The fault should influence **30%** of all requests to the **exchange-vservice** virtual service.

   With the abort fault in place, test how the **frontend** application reacts to HTTP errors.

4. Optionally, update the **frontend** deployment to use the **quay.io/redhattraining/ossm-frontend:3.0** image.

   With both the delay and abort faults in place, test how the new application handles the injected network issues.

Evaluation

Grade your work by running the **lab chaos-mesh grade** command from your **workstation** machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab chaos-mesh grade
```

Finish

On the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab chaos-mesh finish
```

This concludes the lab.

Solution

Testing Service Resilience with Chaos Testing

Performance Checklist

In this lab, you will simulate network issues to test application resilience and graceful handling of network issues.

Outcomes

You should be able to use OpenShift Service Mesh to simulate network failures and delays.

Before You Begin

To perform this exercise, ensure you have access to:

- A configured and running OpenShift cluster.
- A installed and running OpenShift Service Mesh in the OpenShift cluster.
- The OpenShift CLI (**/usr/local/bin/oc**).

As the **student** user on the **workstation** machine, use the **lab** command to prepare your system for this lab.

The **lab** command deploys the exchange application into your Red Hat OpenShift cluster. The source code is in the Git repository at https://github.com/RedHatTraining/DO328-apps in the **exchange-application** directory.

The exchange application consists of the following services:

- Frontend
- Currency
- Exchange
- History

You can examine the services in the **chaos-mesh** project. The application is available using the **istio-ingressgateway** route at the **/frontend** endpoint.

```
[student@workstation ~]$ lab chaos-mesh start
```

1. Log in to OpenShift and verify that the application is ready.

   1.1. Run the following command to load the environment variables created in the Verifying OpenShift Credentials guided exercise:

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

1.2. Log in to OpenShift:

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_API}
Login successful.
...output omitted...
```

1.3. Change to project **chaos-mesh**:

```
[student@workstation ~]$ oc project chaos-mesh
Now using project "chaos-mesh" on server ...
```

1.4. Verify that pods are in the **Running** state:

```
[student@workstation ~]$ oc get pods
NAME                         READY   STATUS    RESTARTS   AGE
currency-566cddc8c6-jtd5g    2/2     Running   0          13s
exchange-66b78bf65c-s72gv    2/2     Running   0          13s
frontend-5648fbb85f-td5dg    2/2     Running   0          13s
history-54b5c9d476-rk4pd     2/2     Running   0          13s
```

1.5. Save the **ingress-gateway** route hostname with the **/frontend** endpoint into a variable:

```
[student@workstation ~]$ FRONTEND=$(oc get route -n istio-system \
> istio-ingressgateway -o jsonpath='{.spec.host}'/frontend)
```

1.6. Verify that the application responds using the **FRONTEND** variable:

```
[student@workstation ~]$ firefox $FRONTEND
```

1.7. Verify that the **Historical Data** page of the application works. Open the page:

```
[student@workstation ~]$ firefox $FRONTEND/history
```

On the **Historical Data** page, click **Submit**.

1.8. Verify that the **Exchange** page of the application works. Open the page:

```
[student@workstation ~]$ firefox $FRONTEND/exchange
```

On the **Exchange** page, click **Submit**.

2. Introduce a delay fault to the **exchange-vservice** virtual service resource with the following parameters:

   - Delay time: **10s**.
   - The fault should influence **20%** of all requests to the **exchange-vservice** virtual service.

   With the delay fault in place, test how the **frontend** application responds to large delay times.

2.1. Edit the **exchange-vservice** virtual service resource. You can use the **~/DO328/solutions/chaos-mesh/vservice-delay.yaml**. Alternatively, add the delay fault manually:

```
[student@workstation ~]$ oc edit virtualservice exchange-vservice
```

Add the following fault section to the first object in the **.spec.http** path:

```
  http:
  - match:
    - uri:
        prefix: /exchange
    rewrite:
      uri: /
    route:
    - destination:
        host: exchange
        port:
          number: 8080
    fault:
      delay:
        fixedDelay: 10s
        percentage:
          value: 20
```

Save the file to update the virtual service.

2.2. Retest the application. Open the **Historical Data** page:

```
[student@workstation ~]$ firefox $FRONTEND/history
```

Refresh the **Historical Data** page until you encounter the delay.

The application waits until the delay is resolved. This is suboptimal in case of a non-responsive service, and is considered a bug in the application.

3. Introduce an abort fault to the **exchange-vservice** virtual service resource with the following parameters:

- HTTP error code: **500**.

The fault should influence **30%** of all requests to the **exchange-vservice** virtual service.

With the abort fault in place, test how the **frontend** application reacts to HTTP errors.

3.1. Edit the **exchange-vservice** virtual service resource. You can use the **~/DO328/solutions/chaos-mesh/vservice-abort.yaml**. Alternatively, add the delay fault manually:

```
[student@workstation ~]$ oc edit virtualservice exchange-vservice
```

Add the following abort section into the already existing fault section:

```
    fault:
      delay:
        fixedDelay: 10s
        percentage:
          value: 20
      abort:
        percentage:
          value: 30
        httpStatus: 500
```

3.2. Retest the application. Open the **`Historical Data`** page:

```
[student@workstation ~]$ firefox $FRONTEND/history
```

3.3. Open the Firefox developer console. Right-click anywhere on the web page. Then, click **Inspect Element**.

In the **Inspector** window, click **Console**.

3.4. With the developer console open, refresh the page until you see the following error:

```
SyntaxError: "JSON.parse: unexpected keyword at line 1 column 1 of the JSON data"
```

The application does not inform the user about any network errors. It fails silently and prints errors only into the developer console.

This behavior leads to user confusion and is considered a bug.

4. Optionally, update the **`frontend`** deployment to use the **`quay.io/redhattraining/ossm-frontend:3.0`** image.

With both the delay and abort faults in place, test how the new application handles the injected network issues.

4.1. Update the **`frontend`** deployment:

```
[student@workstation ~]$ oc edit deployment frontend
```

Change the image version in the **`.spec.template.spec.containers[0].image`** path:

```
    spec:
      containers:
      - env:
        - name: REACT_APP_GW_ENDPOINT
          valueFrom:
            configMapKeyRef:
              key: GW_ADDR
              name: frontend-cm
       image: quay.io/redhattraining/ossm-frontend:3.0
```

Save the file to update the deployment.

4.2. Verify that the **`frontend`** pod is in the **`Running`** state:

```
[student@workstation ~]$ oc get pods
NAME                        READY   STATUS    RESTARTS   AGE
currency-c6879ff94-d7xdr    2/2     Running   0          151m
exchange-b8bb857cd-7xl6l    2/2     Running   0          151m
frontend-847f8bb99f-h4ssf   2/2     Running   0          40s
history-548b7f4954-jdjr8    2/2     Running   0          151m
```

4.3. Retest the application. Open the **Historical Data** page:

```
[student@workstation ~]$ firefox $FRONTEND/history
```

Note the following:

- Because of time-out, no request takes longer than 3 seconds. After 3 seconds, the request is discarded and another request is issued.

  The time-out pattern addresses long delays.

- When the exchange service returns a **5xx** response, requests are re-executed up to 3 times.

  The retry pattern addresses service and network unreliability.

- When the application encounters more than 3 errors, the last error is propagated to the user.

Evaluation

Grade your work by running the **lab chaos-mesh grade** command from your **workstation** machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab chaos-mesh grade
```

Finish

On the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab chaos-mesh finish
```

This concludes the lab.

Summary

In this chapter, you learned:

- How to inject random errors into services to test the application resilience to network errors.
- How to inject random delays into network connections to test applications behavior when faced with network latency.
- How to use errors and delays to perform Chaos Testing.

Chapter 7

Building Resilient Services

Goal

Leverage OpenShift Service Mesh strategies for creating resilient services.

Objectives

- Describe the strategies for creating resilient services with Service Mesh.
- Configure time-outs to maintain service reliability.
- Configure a service retry to maintain service reliability.
- Configure a circuit breaker pattern to maintain service reliability.

Sections

- Describing Strategies for Resilient Services with OpenShift Service Mesh (and Quiz)
- Configuring Time-outs (and Guided Exercise)
- Configuring Retry (and Guided Exercise)
- Configuring a Circuit Breaker (and Guided Exercise)

Lab

Building Resilient Services

Describing Strategies for Resilient Services with OpenShift Service Mesh

Objectives

After completing this section, you should be able to describe the strategies for creating resilient services with Service Mesh.

Describing Strategies for Resilience

As discussed in *Chapter 6, Testing Service Resilience with Chaos Testing*, microservices-based architectures are subject to the negative effects of distributed computing: unreliable networks, transport costs, and latencies. Therefore, you should assume that applications will sometimes experience problems and outages.

Preparing for resilience is a way to make applications more reliable and ready to overcome some of these challenges. Istio takes reliability into account and includes resilience features as a part of its traffic management model. In particular, virtual services and destination rules allow you to configure flexible resilience strategies at different levels, such as the service level or the subset level.

Resilience strategies include:

Load balancing
: Use load balancing to prevent service overloads by distributing the load among several service replicas sufficient to handle the load. To achieve resiliency with load balancing, you must have at least one redundant replica. Thus, if a replica fails, then the load balancer can redistribute all traffic among the rest of the healthy replicas without overwhelming them.

Time-outs
: When making a request to a service, you might encounter errors, such as a slow-down or failure in the service or network.

  Instead of waiting indefinitely when these errors occur, establish a time-out beyond which the request is rejected. Setting a time-out helps applications release resources that are blocked waiting for a response. It also protects the whole system against cascading failures.

Retries
: Sometimes, services might be temporarily unavailable due to transient problems, such as network outages or momentary overloads. To address this situation, configure Istio to retry the initially failed request a given number of times so that a request that would otherwise fail due to a momentary problem can eventually succeed.

Circuit breakers
: When a service approaches capacity, you can stop sending traffic to it, or break the circuit. The service fails fast and you protect the service from becoming overloaded, which can cause instability.

  Istio can break the circuit statically and dynamically. You can define static connection and request limits to protect a service against high loads. The dynamic mechanism uses outlier detection, which monitors the status of each service host and stops sending traffic to hosts that become unhealthy.

Implementing Service Resilience with Load Balancing

One basic way of implementing service resilience is to distribute the load among multiple replicas of the same service. If one of the replicas experiences a failure, then the load balancer removes that replica from its pool and distributes the load among the healthy nodes. To be resilient, the service must have at least one redundant replica, which is known as the *N+1* redundancy rule.

For example, if a service receives 50 requests per second, and each replica can handle 10 requests per second, then the minimum number of replicas to be resilient is six. If one of the replicas fails, then the service still has five replicas that can handle 100% of the load.

To configure load balancing for resilience at the service level, you must use the **`DestinationRule`** configuration resource. Specifically, you must set the value of the **`spec.trafficPolicy.loadBalancer.simple`** field to one of the following algorithms:

ROUND_ROBIN
: Requests are sent to each host in turn to distribute the load evenly. This is the default algorithm.

RANDOM
: Requests are sent to hosts randomly. Under high loads, requests are distributed randomly across instances.

LEAST_CONN
: Requests are sent to a host with few connections. This algorithm picks two random hosts and chooses the host with the fewest active connections.

The following is an example of a destination rule that uses the least requested load balancer:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: my-destination-rule (1)
spec:
  host: my-svc (2)
  trafficPolicy: (3)
    loadBalancer:
      simple: LEAST_CONN (4)
```

1. Name of the destination rule.
2. Service affected by the defined policies.
3. Traffic policy defined for the **`my-svc`** service.
4. Load balancing algorithm name.

You can also define load balancing policies at the subset level, applying specific load balancers to different versions of the same service. The following example shows how to specify load balancing features both at the service and the subset level for a specific version:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: my-destination-rule
spec:
  host: my-svc
  trafficPolicy: (1)
    loadBalancer:
```

```
      simple: LEAST_CONN
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
    trafficPolicy: (2)
      loadBalancer:
        simple: RANDOM
```

(1) Traffic policy that applies the **LEAST_CONN** load balancer to the service.
(2) Traffic policy that applies the **RANDOM** load balancer to the **v2** subset, overriding the policy at the service level.

Consistent Hash Load Balancing

Istio includes a more advanced load balancing algorithm called Consistent Hash-based load balancing. This load balancer provides soft session affinity by mapping HTTP headers, cookies, or source IP to a particular service host. When hosts are added to or removed from the service, the affinity is lost.

You can use this load balancer to keep a user session on a host when you want all requests from that user to go to the same host. The following example demonstrates using the **session_id** cookie as the hash key to apply this strategy:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: my-destination-rule
spec:
  host: my-svc
  trafficPolicy:
    loadBalancer:
      consistentHash:
        httpCookie:
          name: session_id
          ttl: 0s
```

Similarly, you can use the same approach with other HTTP headers or cookies. For example, you can distribute the load based on the requested endpoint.

Because Consistent Hash-based load balancing establishes affinity between request data and specific hosts, this type of load balancing can lead to host overload when your traffic is unbalanced. For example, demanding users generating high loads, or popular endpoints receiving much more load than other endpoints may cause this overload.

References

N+1 redundancy
https://en.wikipedia.org/wiki/N%2B1_redundancy

Istio Docs: Traffic Management
https://archive.istio.io/v1.4/docs/concepts/traffic-management/

Istio Docs: LoadBalancerSettings
https://archive.istio.io/v1.4/docs/reference/config/networking/destination-rule/#LoadBalancerSettings

References

For more information, refer to the *Traffic management* section in the *Red Hat Service Mesh Guide* at
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.4/html-single/service_mesh/index

Quiz

Describing Strategies for Resilient Services with OpenShift Service Mesh

Choose the correct answers to the following questions:

1. Which three strategies can be used to implement service resiliency in OpenShift Service Mesh? (Choose three.)

a. Canary releases.
b. Load balancing.
c. Time-outs.
d. Distributed tracing.
e. Circuit breakers.

2. Your networking provider is experiencing problems that affect your cluster. These problems cause outages in the network that last for less than a second. As a result, a small percentage of requests directed to your services fail.
Which resilience strategy can make the failing requests succeed?

a. Load balancing.
b. Circuit breaker.
c. Retries.
d. Time-outs.

3. You are deploying an application on OpenShift Service Mesh. You estimate that the application will receive an average of 30 requests per second.
Assuming that the application can handle up to 10 requests per second, what is a good strategy for resilience?

a. Scale the service to 4 replicas and balance the load among them.
b. Scale the service to 3 replicas and balance the load among them.
c. Limit the load to 10 requests per second with a circuit breaker.
d. Configure retries to resend a request to the service if the request fails.

4. Which resilience strategy protects a service from overloading by stopping traffic directed to it?

a. Retries.
b. Circuit breaker.
c. Time-outs.
d. Load balancing.

Solution

Describing Strategies for Resilient Services with OpenShift Service Mesh

Choose the correct answers to the following questions:

1. Which three strategies can be used to implement service resiliency in OpenShift Service Mesh? (Choose three.)

a. Canary releases.
b. Load balancing.
c. Time-outs.
d. Distributed tracing.
e. Circuit breakers.

2. Your networking provider is experiencing problems that affect your cluster. These problems cause outages in the network that last for less than a second. As a result, a small percentage of requests directed to your services fail.
Which resilience strategy can make the failing requests succeed?

a. Load balancing.
b. Circuit breaker.
c. Retries.
d. Time-outs.

3. You are deploying an application on OpenShift Service Mesh. You estimate that the application will receive an average of 30 requests per second.
Assuming that the application can handle up to 10 requests per second, what is a good strategy for resilience?

a. Scale the service to 4 replicas and balance the load among them.
b. Scale the service to 3 replicas and balance the load among them.
c. Limit the load to 10 requests per second with a circuit breaker.
d. Configure retries to resend a request to the service if the request fails.

4. Which resilience strategy protects a service from overloading by stopping traffic directed to it?

a. Retries.
b. Circuit breaker.
c. Time-outs.
d. Load balancing.

Configuring Time-outs

Objectives

After completing this section, you should be able to configure time-outs to maintain service reliability.

Defining Time-outs

Cloud-native applications are composed of different microservices making various internal and external calls. When an application relies on different components distributed across the network, the network connection becomes one of the biggest potential problems for the stability of your application.

Both the network and external service are inherently unreliable. To improve the resilience of your applications, you can use a *time-out*.

A time-out is the amount of time that a service or an application waits for some event. OpenShift Service Mesh enables you to configure the time-out outside of your application code, in the Envoy proxy, using virtual services or HTTP headers.

Using a time-out provides:

- A simple way of mitigating cascading failures. Because your application is failing early, you stop propagating slow responses from downstream services to systems that depend on yours.
- A guarantee that a network request finishes within a limited time.
- Better resource usage, because the time-outs reduce the time an application is blocked waiting for a response.

Configuring Time-outs in OpenShift Service Mesh

Time-outs can be managed in the application code, but that approach has drawbacks including:

- Adding an additional layer of complexity that must be maintained.
- Coupling the application with the network layer.

Using OpenShift Service Mesh to manage time-outs enables you to maintain a separation of application business logic and network management.

In OpenShift Service Mesh, you can configure the time-outs using virtual services or HTTP headers without modifying your application code.

Note
The default time-out for HTTP connections in OpenShift Service Mesh is **15** seconds.

Configuring Time-outs Using Virtual Services

Virtual services allow you to configure time-outs for all traffic routed to a service. You can apply a time-out setting using the **`timeout`** field in the route rules and assigning a value measured in seconds.

The following example shows a time-out configuration in a virtual service:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: a-service-vs
spec:
  hosts:
    - example-svc
  http:
    - route:
        - destination:
          host: preference
      timeout: 1s
```

In the preceding example, Envoy waits up to 1 second on any call to the **`example-svc`** service before returning a time-out error.

Configuring Time-outs Using HTTP Headers

In OpenShift Service Mesh, you can use HTTP headers to modify Envoy behavior. The Envoy proxy can add, remove, or modify HTTP headers for incoming requests. When requests with HTTP headers modifying Envoy proxy behaviour are made from outside the mesh, the Envoy proxy ignores them.

In OpenShift Service Mesh, you can use the **`PILOT_SIDECAR_USE_REMOTE_ADDRESS`** flag to modify how Envoy determines the origin of a connection. Setting the value of **`PILOT_SIDECAR_USE_REMOTE_ADDRESS`** to **`true`**, allows you to configure time-outs using headers.

Warning
Changing Pilot settings can have unexpected consequences on the stability and behavior of your service mesh.

You can configure time-outs adding the **`x-envoy-upstream-rq-timeout-ms`** request HTTP header with a value assigned in milliseconds.

The following example shows a request to a service with time-out settings:

```
HTTP/1.1 200 OK
date: Wed, 13 May 2020 13:56:01 GMT
...output omitted...
x-envoy-upstream-rq-timeout-ms: 500
...output omitted...
```

The preceding example defines a time-out of 500 milliseconds that is only valid until the service responds to that request.

Selecting Time-outs for Resilience

Each application is different, and the time required to generate a response depends on multiple factors, such as how busy the application is or if it calls external services. There is no standard way of calculating a precise value for the time-out, but there are several things to consider when defining a time-out value:

- The value allows slow responses to arrive.
- The value stops waiting for a response that is not returned.
- A high value increases latency, especially in distributed systems.
- A high value potentially increases computing resources waiting for a dead service to respond.

Time-outs are not the only solution to increase the reliability of your applications. You can combine time-outs with more advanced strategies like retries or circuit breakers.

References

Time-outs in Istio
https://archive.istio.io/v1.4/docs/tasks/traffic-management/request-timeouts/

HTTPRoute
https://archive.istio.io/v1.4/docs/reference/config/networking/virtual-service/#HTTPRoute

HTTP Header sanitizing with Envoy
https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_conn_man/header_sanitizing

Guided Exercise

Configuring Time-outs

In this exercise, you will configure time-outs for an application deployed in OpenShift Service Mesh.

Outcomes

You should be able to configure time-outs in OpenShift Service Mesh.

The application is composed of several services calling each other. These services pass requests as follows:

- **`customer`**: the entry point of the application sending requests to the **`preference`** service.
- **`preference`**: receives requests from the **`customer`** service, and sends requests to the **`recommendation`** service.
- **`recommendation`**: receives requests from the **`preference`** service and returns a response. This is the last service in the chain of requests.

The final response from the application includes the responses from each service.

Before You Begin

To perform this exercise, ensure you have access to:

- A configured and running OpenShift cluster.
- A installed and running OpenShift Service Mesh in the OpenShift cluster.
- The OpenShift CLI (**`/usr/local/bin/oc`**).

As the **`student`** user on the **`workstation`** machine, use the **`lab`** command to prepare your system for this exercise.

```
[student@workstation ~]$ lab resilience-timeout start
```

1. This guided exercise uses scripts that are located in **`~/DO328/labs/resilience-timeout`**. Change to that directory.

```
[student@workstation ~]$ cd ~/DO328/labs/resilience-timeout
[student@workstation resilience-timeout]$
```

2. Log in to the OpenShift cluster as an unprivileged user and verify that the lab project has successfully deployed.

   2.1. Source the classroom configuration file that is accessible at **`/usr/local/etc/ocp4.config`**.

```
[student@workstation resilience-timeout]$ source /usr/local/etc/ocp4.config
```

2.2. Log in to OpenShift as the **developer** user.

```
[student@workstation resilience-timeout]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_API}
Login successful.
...output omitted...
```

2.3. Change to the **resilience-timeout** project.

```
[student@workstation resilience-timeout]$ oc project resilience-timeout
Now using project "resilience-timeout" on server ...
```

2.4. Verify the status of the **resilience-timeout** project pods.

```
[student@workstation resilience-timeout]$ oc get pods
NAME                              READY   STATUS    RESTARTS   AGE
customer-f7bffdbf6-9mbzn          2/2     Running   0          50s
preference-5585d5987f-9z26q       2/2     Running   0          50s
recommendation-75c77bd445-55bb4   2/2     Running   0          50s
```

2.5. Examine the **response-times.sh** script which uses the **curl** command to make a call to the lab application and returns a custom output.

Execute the **response-times.sh** script to test the lab application response time..

```
[student@workstation resilience-timeout]$ sh response-times.sh
customer => preference => recommendation v1 from 'f11b097f1dd0': 1
HTTP code: 200
Time: 0,032099s
```

3. Configure a delay of **3** seconds in the **recommendation-vs** virtual service.

3.1. Examine the **recommendation-delay.yaml** file that configures a **3** seconds delay.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
...output omitted...
  http:
    - fault:
        delay:
          percent: 100
          fixedDelay: 3s
...output omitted...
```

Use the **oc replace** command to replace the virtual service configuration with the new one.

```
[student@workstation resilience-timeout]$ oc replace -f recommendation-delay.yaml
virtualservice.networking.istio.io/recommendation-vs replaced
```

4. Verify the response time of the application.

4.1. Execute the **`response-times.sh`** script to verify the increased response time.

```
[student@workstation resilience-timeout]$ sh response-times.sh
customer => preference => recommendation v1 from 'f11b097f1dd0': 2
HTTP code: 200
Time: 3,063155s
```

5. Configure a route time-out of **`0.5`** seconds in the **`preference-vs`** virtual service.

5.1. Examine the **`route-timeout.yaml`** file that configures the route time-out.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
...output omitted...
  http:
    - route:
        ...output omitted...
      timeout: 0.5s
```

Use the **`oc replace`** command to replace the virtual service configuration with the new one.

```
[student@workstation resilience-timeout]$ oc replace -f route-timeout.yaml
virtualservice.networking.istio.io/preference-vs replaced
```

6. Verify the response time of the application.

6.1. Execute the **`response-times.sh`** script to verify the time-out setting.

```
[student@workstation resilience-timeout]$ sh response-times.sh
customer => Error: 504 - upstream request timeout
HTTP code: 503
Time: 1,630933s
```

The **`preference`** service only waits **`0.5`** seconds for a response from the **`recommendation`** service. The reason that the response takes more than 0.5 seconds is because of the default Istio retry policies.

7. Update the **`preference-vs`** virtual service to remove the route time-out setting.

7.1. Examine the **`removed-timeout.yaml`** file that configures the **`preference-vs`** virtual service to use the default route time-out settings.

Use the **`oc replace`** command to replace the virtual service configuration with the new one.

```
[student@workstation resilience-timeout]$ oc replace -f removed-timeout.yaml
virtualservice.networking.istio.io/preference-vs replaced
```

7.2. Execute the **`response-times.sh`** script to test the response of the lab application.

```
[student@workstation resilience-timeout]$ sh response-times.sh
customer => preference => recommendation v1 from 'f11b097f1dd0': 5
HTTP code: 200
Time: 3,034979s
```

After removing the time-out in the **preference-vs** virtual service, the response time returns to values close to 3 seconds.

8. Verify that the application applied a timeout of **0.5** seconds per request.

8.1. Set the **PILOT_SIDECAR_USE_REMOTE_ADDRESS** environment variable in the **istio-pilot** deployment.

By default the Envoy proxy discards any HTTP header that modifies its behaviour. Enabling the **PILOT_SIDECAR_USE_REMOTE_ADDRESS** environment variable allows you to modify the time-out settings using HTTP headers.

```
[student@workstation resilience-timeout]$ oc set env deployment/istio-pilot \
> PILOT_SIDECAR_USE_REMOTE_ADDRESS=true -n istio-system
deployment.apps/istio-pilot updated
```

Wait a few seconds until OpenShift redeploys the **istio-pilot** container with the new environment variable. Use the **oc** command to follow the progress.

```
[student@workstation resilience-timeout]$ oc get pods -n istio-system
NAME                                READY   STATUS        RESTARTS   AGE
...output omitted...
istio-ingressgateway-7f6fcf4bc9-9ng74   1/1     Running       0          42h
istio-pilot-5bbc676f7c-lj29x        2/2     Running       0          7s
istio-pilot-6b5f69bcc7-4h6v8        2/2     Terminating   0          42h
istio-policy-7cb97db7c8-cf55c       2/2     Running       0          42h
...output omitted...
```

Warning

Changing pilot settings can have unexpected consequences on the stability and behavior of your service mesh.

8.2. Examine the **headers-timeout.yaml** file that configures the **customer** virtual service to add a header.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
...output omitted...
  http:
    - headers:
        request:
          set:
            x-envoy-upstream-rq-timeout-ms: "500"
...output omitted...
```

Use the **oc replace** command to replace the virtual service configuration with the new one.

```
[student@workstation resilience-timeout]$ oc replace -f headers-timeout.yaml
virtualservice.networking.istio.io/preference-vs replaced
```

8.3. Execute the **response-times.sh** script to verify the time-out setting.

```
[student@workstation resilience-timeout]$ sh response-times.sh
customer => Error: 504 - upstream request timeout
HTTP code: 503
Time: 1,563069s
```

9. Return to the home directory.

```
[student@workstation resilience-timeout]$ cd ~
[student@workstation ~]$
```

Finish

On the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab resilience-timeout finish
```

This concludes the guided exercise.

Configuring Retry

Objectives

After completing this section, you should be able to configure a service retry to maintain service reliability.

Defining the Retry Pattern

The *retry pattern* is a behavioral design pattern that focuses on reducing transient, short-lived communication failures. In a cloud-native environment, microservices often rely on networks to communicate with other microservices. Because networks can be unreliable, a microservice might encounter a number of issues during a communication request, such as:

- A request is lost, mishandled, or dropped due to an overloaded network.
- A target service experiences a temporary failure, for example, due to storage becoming temporarily disconnected.
- A subset of target service pods experience a failure.
- A request response takes longer than expected, resulting in the source service experiencing a time-out.

When a request fails due to any of the errors described above, a repeated request with identical parameters might still succeed. The retry pattern prevents propagation of such transient errors into the application.

OpenShift Service Mesh enables you to implement the retry pattern without changing the application code. Consequently, you can easily change the retry configuration at runtime, without recompiling or redeploying your application.

Configuring Retries in OpenShift Service Mesh

Implementing the retry pattern in the logic of the application is possible, but has several drawbacks:

- The application code contains non-business logic. The application code becomes less focused, and thus more difficult to understand and maintain.
- Depending on the implementation, changing the parameters of the retry configuration might require redeploying the application.

Implementing the retry pattern using OpenShift Service Mesh provides benefits, including:

- Envoy proxies in OpenShift Service Mesh provide advanced configuration integrated with the Red Hat OpenShift platform. For example, you can enable automatic outlier detection in case of multiple **`50x`** HTTP status codes, and provide routing logic for subsequent retries.
- Both administrators and developers can change the configuration of retries and other resiliency features. This encourages the DevOps approach towards developing and maintaining an application.

Configuring Retry Using Virtual Service

You can configure the retry pattern in the virtual service resource, for example:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: example-vs
spec:
  hosts:
  - example-svc
  http:
  - route:
    - destination:
        host: example-svc
        subset: v1
    retries: (1)
      attempts: 3 (2)
      perTryTimeout: 2s (3)
      retryOn: 5xx,retriable-4xx (4)
```

1. The **`HTTPRetry`** object responsible for configuring retries.
2. The number of times to resend a request.
3. A time-out value for each retry request. Valid values are in milliseconds **`ms`**, seconds **`s`**, minutes **`m`**, or hours **`h`**.
4. A policy that specifies conditions that cause failed requests to retry. The value is a list of comma-separated values.

Virtual services retry failed requests twice by default. To disable the retry configuration, set the attempts parameter of the retry configuration to **`0`**. For example:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: vs-with-no-retry
spec:
  hosts:
  - example-svc
  http:
  - retries:
      attempts: 0
    route:
    - destination:
        host: example-svc
```

Selecting Retry Policies

The virtual service resource enables you to select one or more retry policies. The Envoy proxy evaluates each failed request using the retry policies. If the Envoy proxy matches a request with any of the selected policies, it retries that request.

Selecting retry policies is important for optimal behavior of your service mesh. OpenShift Service Mesh contains, among others, the following retry policies:

5xx
: This policy matches any response that contains the **`5xx`** response code. Additionally, this policy matches any requests that do not get a response, such as due to a disconnect, reset, or a read time-out.

  Note that setting the **`x-envoy-upstream-rq-timeout-ms`** header overrides the configuration time-out. If a request violates a value set in this header, the response contains the **`504`** response code, but will not be matched by the **`5xx`** policy.

gateway-error
: This policy matches responses that contain the **`502`**, **`503`**, or **`504`** response codes.

reset
: This policy matches requests without any response due to disconnect, reset, or read time-out.

retriable-4xx
: This policy matches request responses that contain the **`409`** response code.

Note that the list above is not exhaustive. When choosing a retry policy, it is a good practice to first analyze all failed requests in your application and choose the most specific policy for that case. For example, an application pod might take a long time to start, and responds to first requests with a time-out. You can mitigate the issue using the **`reset`** retry policy.

Selecting Retry Parameters for Resiliency

The configuration of the retry communication pattern requires carefully considering the parameters of your environment, such as the latency requirements, network layout, application complexity, and others.

There are no standard values for the retry configuration that are suitable for every environment. The following non-exhaustive list contains some of the considerations for selecting retry parameters:

- An incorrect retry policy can substantially impact application performance. For example, if the back end application is incorrectly configured, and any requests result in a disconnect response, then retries only increase the overall number of retries with no benefits to the end-user.

- Increasing the number of retries increases the potential probability for success at the cost of performance. A higher number of retries results in larger network saturation and can cause issues in busy environments.

- Increasing the time-out value helps to reduce the load for compute-intensive services that can take longer to respond. However, increasing the time-out values also increases the overall latency of your system.

An incorrect retry setting makes your environment less resilient, and can accelerate performance issues in your environment. Other resilience patterns, such as the circuit breaker, can mitigate worst-case retry scenarios. Additionally, Red Hat recommends implementing monitoring to alert you to possible issues as soon as they occur.

References

Retry Concepts
https://archive.istio.io/v1.4/docs/concepts/traffic-management/#retries

HTTP Retry Reference Documentation
https://archive.istio.io/v1.4/docs/reference/config/networking/virtual-service/#HTTPRetry

Retry Policies
https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/router_filter#x-envoy-retry-on

Envoy Proxy Outlier Detection
https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/upstream/outlier#arch-overview-outlier-detection

Retries in Envoy Proxy
https://www.envoyproxy.io/docs/envoy/latest/faq/load_balancing/transient_failures.html?highlight=retry#retries

409 Conflict - HTTP | MDN
https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/409

Guided Exercise

Configuring Retry

In this exercise, you will configure the retry settings of an Envoy proxy.

Outcomes

You should be able to configure retries for a service in OpenShift Service Mesh without changing the application code.

Before You Begin

To perform this exercise, ensure you have access to:

- A configured and running OpenShift cluster.
- A installed and running OpenShift Service Mesh in the OpenShift cluster.
- The OpenShift CLI (**/usr/local/bin/oc**).

As the **student** user on the **workstation** machine, use the **lab** command to prepare your system for this exercise.

The **lab** command deploys the **customer**, **preference**, and **recommendation** services into your Red Hat OpenShift cluster. The source code is in the Git repository at https://github.com/RedHatTraining/DO328-apps in the **customer**, **preference**, and **recommendation** directories.

```
[student@workstation ~]$ lab resilience-retry start
```

1. Log in to the OpenShift cluster and verify that the lab project is successfully deployed.

   1.1. Source the classroom configuration file that is accessible at **/usr/local/etc/ocp4.config**.

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

   1.2. Log in to OpenShift as the **developer** user.

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_API}
Login successful.
...output omitted...
```

   1.3. Change to the **resilience-retry** project.

```
[student@workstation ~]$ oc project resilience-retry
Now using project "resilience-retry" on server ...
```

   1.4. Verify that pods are in the **Running** state:

```
[student@workstation ~]$ oc get pods
NAME                              READY   STATUS    RESTARTS   AGE
customer-f7bffdbf6-9mbzn          2/2     Running   0          50s
preference-5585d5987f-9z26q       2/2     Running   0          50s
recommendation-75c77bd445-55bb4   2/2     Running   0          50s
```

1.5. Save the **`ingress-gateway`** route host name with the **`/mtls`** endpoint into a variable:

```
[student@workstation ~]$ INGRESS_URL=$(oc get route -n istio-system \
> istio-ingressgateway -o jsonpath='{.spec.host}'/resilience-retry)
```

1.6. Verify that the service responds using the **`INGRESS_URL`** variable:

```
[student@workstation ~]$ curl $INGRESS_URL
customer => preference => recommendation v1 from 'f11b097f1dd0': 1
```

Note that you may encounter an error. This error is injected in the **`recommendation-vs`** virtual service, with a **`90%`** probability of returning the **`500`** HTTP code. The error injection simulates an environment where retries are useful.

2. Configure a retry policy for the **`preference-vs`** virtual service with the following parameters:

- The Envoy proxy executes **`10`** or fewer retry attempts.
- Each retry waits at most **`1s`** before timing out.
- The Envoy proxy issues retries only when the response contains a **`500`** HTTP code.

2.1. Edit the **`preference-vs`** virtual service:

```
[student@workstation ~]$ oc edit virtualservice preference-vs
```

2.2. Add the **`retries`** configuration. View the **`~/DO328/solutions/resilience-retry/preference-retry.yml`** file to see the solution.

```
spec:
  hosts:
  - preference
  http:
  - route:
    - destination:
        host: preference
        port:
          number: 8080
    retries:
      attempts: 10
      perTryTimeout: 1s
      retryOn: 5xx
```

3. Verify the retry configuration.

3.1. Change to the **~/DO328/labs/resilience-retry** directory:

```
[student@workstation ~]$ cd ~/DO328/labs/resilience-retry
[student@workstation resilience-retry]$
```

3.2. Examine the **test-retries.sh** file. Then, execute it.

```
[student@workstation resilience-retry]$ sh test-retries.sh
Executing 10 requests:

customer => preference => recommendation v1 from 'f11b097f1dd0': 6
customer => preference => recommendation v1 from 'f11b097f1dd0': 7
customer => Error: 503 - preference => Error: 500 - fault filter abort

customer => preference => recommendation v1 from 'f11b097f1dd0': 8
customer => preference => recommendation v1 from 'f11b097f1dd0': 9
customer => preference => recommendation v1 from 'f11b097f1dd0': 10
customer => preference => recommendation v1 from 'f11b097f1dd0': 11
customer => preference => recommendation v1 from 'f11b097f1dd0': 12
customer => preference => recommendation v1 from 'f11b097f1dd0': 13
customer => preference => recommendation v1 from 'f11b097f1dd0': 14

Done
```

4. Set the **recommendation-vs** virtual service to have a **99%** probability of returning a **500** HTTP code.

4.1. Edit the **recommendation-vs** virtual service:

```
[student@workstation resilience-retry]$ oc edit virtualservice recommendation-vs
```

4.2. Change the percentage value to **99**.

```
spec:
  hosts:
  - recommendation
  http:
  - fault:
      abort:
        httpStatus: 500
        percentage:
          value: 99
    route:
    - destination:
        host: recommendation
        port:
          number: 8080
```

5. Change the retry value of the **preference-vs** virtual service to retry **100** times, with a **2s** time-out limit.

5.1. Edit the **preference-vs** virtual service:

```
[student@workstation resilience-retry]$ oc edit virtualservice preference-vs
```

5.2. Change the number of retries and time-out settings:

```
spec:
  hosts:
  - preference
  http:
  - retries:
      attempts: 100
      perTryTimeout: 2s
      retryOn: 5xx
    route:
    - destination:
        host: preference
        port:
          number: 8080
```

6. Verify the responsiveness of the service:

```
[student@workstation resilience-retry]$ sh test-retries.sh
Executing 10 requests:

customer => preference => recommendation v1 from 'f11b097f1dd0': 15
customer => preference => recommendation v1 from 'f11b097f1dd0': 16
customer => preference => recommendation v1 from 'f11b097f1dd0': 17
customer => preference => recommendation v1 from 'f11b097f1dd0': 18
customer => preference => recommendation v1 from 'f11b097f1dd0': 19
customer => preference => recommendation v1 from 'f11b097f1dd0': 20
customer => preference => recommendation v1 from 'f11b097f1dd0': 21
customer => preference => recommendation v1 from 'f11b097f1dd0': 22
customer => preference => recommendation v1 from 'f11b097f1dd0': 23
customer => preference => recommendation v1 from 'f11b097f1dd0': 24

Done
```

The service responds properly even when 99% of the requests fail. Note that you can still see a reduced number of time-outs or errors.

The large number of retries generates large load on your Red Hat OpenShift cluster. The script finishes correctly, but takes a long time to complete

Consequently, the retry value considerably increases the overall latency of your application. Such a high retry configuration is dangerous in case the service is faulty, or incorrectly configured.

7. Change into the home directory:

```
[student@workstation resilience-retry]$ cd ~
[student@workstation ~]$
```

Finish

On the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab resilience-retry finish
```

This concludes the guided exercise.

Configuring a Circuit Breaker

Objectives

After completing this section, you should be able to configure a circuit breaker pattern to maintain service reliability.

Describing the Circuit Breaker

When a service experiences transient errors, those errors tend to occur continuously. **`Circuit Breaker`** uses this knowledge to temporarily avoid directing requests to a failing host. When a request is about to reach a failing host, the circuit breaks, sending a failure to the client without the need to wait for the host to respond. This ban is temporary, so the host receives new requests when normal function is restored.

This behavior has two benefits. First, as requests do not reach the failing hosts, services are more responsive, even if the host is slow. Second, the service in the host stops receiving requests for some time, allowing the service to recover from overload and resolve pending requests.

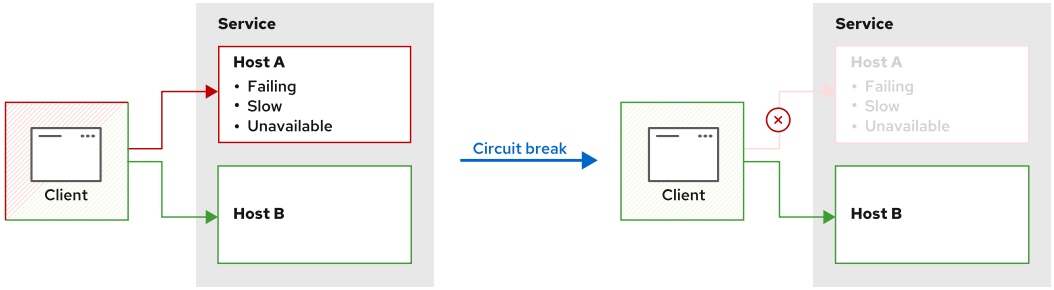


Figure 7.1: The Circuit Breaker

Circuit Breaker classifies host failures as two kinds.

Local origin
: Local failures are service errors (usually HTTP codes above 500) generated by the service.

Gateway origin
: Gateway failures arise when the service is unreachable or unresponsive, hence it cannot be used.

A Circuit Breaker identifies both kinds of failures and stops sending requests to the failing host, forwarding requests only to healthy hosts.

Detecting failing hosts, whether failures are of local origin or gateway failures, and marking them for eviction is called *Outlier Detection*.

Selecting Circuit Breakers for Resilience

Circuit Breakers are useful to protect services prone to transient failures. Compute-intensive services, for example, receive more requests than they can respond to and may experience transient failures more often. Circuit Breakers redirect requests from the host as it starts failing or timing out, so the service has time recover from the increased load.

Other common examples of selecting Circuit Breakers for resilience are services that need to process requests sequentially. Those services usually store pending requests in a queue that they process in order. If this queue becomes too big, the service takes too much time to respond, degrading the service. Circuit Breakers detect those time-outs and give the host time to empty the queue.

Configuring Circuit Breakers in OpenShift Service Mesh

OpenShift Service Mesh implements Circuit Breakers at the host (network) level, not at the service level. That means OpenShift Service Mesh evicts failing hosts, not failing services nor subsets. This behavior allows services to keep functioning even if some subset or some hosts fail.

Note

Istio terminology, and consequently OpenShift Service Mesh terminology,can be confusing when referring to a **`host`**. In many situations, like in the **`DestinationRule`** resource, **`host`** refers to a service as an entry in the Kubernetes service registry. However, in the context of Circuit Breakers, **`host`** refers to a physical or virtual workload, usually a container.

Managing Unhealthy Hosts

To enable a Circuit Breaker, include an **`outlierDetection`** entry in the **`DestinationRule`** resource related to the service:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: myDestinationRule
spec:
  host: myService (1)
  trafficPolicy: (2)
    outlierDetection:
      consecutiveErrors: 1 (3)
      interval: 1s (4)
      baseEjectionTime: 3m (5)
      maxEjectionPercent: 100 (6)
```

1. **`host`** does not refer to the physical host, but the service name. See references for details.
2. The **`outlierDetection`** entry belongs to a **`trafficPolicy`** object.
3. Defines how many errors are allowed before evicting the host.
4. The time interval between checking error counts.
5. The *minimum* amount of host ejection time.
6. The maximum percentage of evicted hosts belonging to a service at any time.

The value for **`baseEjectionTime`** indicates the minimum eviction time for the host, not the actual time. The first time that OpenShift Service Mesh evicts the host, the eviction lasts for approximately the minimum time. Subsequent evictions multiply the **`baseEjectionTime`** by the number of times the host is evicted. For example, if **`baseEjectionTime`** is five seconds, then the first time the host is evicted, the eviction lasts five seconds. The second time that same host is evicted, the eviction lasts ten seconds. The third time, the eviction lasts fifteen seconds. And so on.

The **`maxEjectionPercent`** value limits the percentage of hosts that can simultaneously be in the evicted state. If the current percentage of evicted hosts is higher than this limit, OpenShift Service Mesh evicts no other hosts, even if they fail or are unavailable. This limit is useful to avoid the eviction of all the hosts for a service, making the service unavailable even if some hosts can respond. The default value for **`maxEjectionPercent`** is 10%.

Configuring Connection Limits

Another technique to protect hosts from failing is limiting the number of simultaneous connections to the host. If hosts are prone to time-out or fail when they receive too many requests, limiting the number of connections helps prevent the host from crashing.

OpenShift Service Mesh enables applying those limits using a **`connectionPool`** entry in the **`DestinationRule`**:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: myDestinationRule
spec:
  host: myService
  trafficPolicy:
    connectionPool:
      tcp: (1)
        maxConnections: 1 (2)
        connectTimeout: 30ms (3)
      http: (4)
        http1MaxPendingRequests: 1 (5)
        maxRequestsPerConnection: 1 (6)
```

(1)(4) Connection pool settings are divided into **`HTTP`** and **`TCP`** settings for clarity.
(2) Maximum number of simultaneous connections established to the host.
(3) Maximum time to establish the connection.
(5) Maximum number of requests pending service by the host.
(6) Maximum number of requests permitted on a single connection. OpenShift Service Mesh reuses connections until reaching this limit, or the **`tcp.tcpKeepalive`** time is consumed.

See the Istio reference documentation about **`DestinationRule`** resources for the complete list of supported entries.

When limiting the connections to a host, if the threshold is exceeded it will generate service failures, specifically, gateway failures. Those limits can be applied simultaneously with a Circuit Breaker. Failures generated by the connection limit are used by the Circuit Breaker to break the circuit and start eviction policies. However, connection limits and Circuit Breaker are independent traffic policies, and developers can use one of them without the other.

Connection pools apply to every host in the service. That means each host has a connection pool independent from other host pools. If the host depletes its connection pool, OpenShift Service Mesh establishes no more connections to that host, but continues using the rest of the hosts.

References

Istio documentation on cirtuit breakers
https://archive.istio.io/v1.4/docs/concepts/traffic-management/#circuit-breakers

Istio documentation on outlier detection
https://archive.istio.io/v1.4/docs/reference/config/networking/destination-rule/#OutlierDetection

Istio documentation on connection pools
https://archive.istio.io/v1.4/docs/reference/config/networking/destination-rule/#ConnectionPoolSettings

Envoy proxy documentation on cirtuit breaking
https://www.envoyproxy.io/docs/envoy/v1.14.1/intro/arch_overview/upstream/circuit_breaking

Envoy proxy documentation on outlier detection
https://www.envoyproxy.io/docs/envoy/v1.14.1/intro/arch_overview/upstream/outlier

Istio reference documentation for `DestinationRule` resources
https://archive.istio.io/v1.4/docs/reference/config/networking/destination-rule/#DestinationRule

Guided Exercise

Configuring a Circuit Breaker

In this exercise, you will configure a connection pool for protecting a service from failing, and a Circuit Breaker to manage unhealthy hosts.

This exercise installs a basic greeting service that fails when it is stressed. First, you add a connection pool to avoid sending requests to the service that fails Second, you add a new version of the service. Finally, you deploy the new version beside the old and configure Circuit Breaker to avoid service failures when the original version of the service is stressed.

Outcomes

You should be able to configure Circuit Breaker and connection pools in OpenShift Service Mesh.

Before You Begin

To perform this exercise, ensure you have access to:

- A configured and running OpenShift cluster.
- A installed and running OpenShift Service Mesh in the OpenShift cluster.
- The OpenShift CLI (**`/usr/local/bin/oc`**).

As the **`student`** user on the **`workstation`** machine, use the **`lab`** command to prepare your system for this exercise.

```
[student@workstation ~]$ lab resilience-break start
```

1. Validate that the service is working as expected. Include a **`connectionPool`** entry to limit the number of simultaneous connections to the host.

   1.1. Log in your Red Hat OpenShift cluster using the developer credentials and make sure that you use the **`resilience-break`** project:

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_API}
Login successful.
...output omitted...
[student@workstation ~]$ oc project resilience-break
Now using project "resilience-break" on server ...output omitted...
```

   1.2. The service is exposed in the default gateway, so retrieve the URL using the following command:

```
[student@workstation ~]$ GATEWAY_URL=$(oc get route istio-ingressgateway \
> -n istio-system -o template --template '{{ .spec.host }}')
```

1.3. Test the service to ensure that it installed and is functioning correctly.

```
[student@workstation ~]$ curl -w "%{http_code}\n" $GATEWAY_URL
Hello World!
200
```

1.4. The installed service fails when it receives too many requests. Use the **parallel.sh** script provided in the labs folder to perform twenty parallel requests to the service.

```
[student@workstation ~]$ cd DO328/labs/resilience-break
[student@workstation resilience-break]$ ./parallel.sh \
> "curl -w '%{http_code}\n' $GATEWAY_URL"
Hello World!
200
Hello World!
200
503
503
...output omitted...
```

These results prove that the service is unable to respond to the increased load. The service correctly responds a few times, providing the **Hello World!** text and the **200** HTTP code. Eventually the service fails and returns the **503** failure HTTP code.

1.5. Configure a connection pool to reduce the number of connections allowed to the service. Limit the number of concurrent connections, the number of requests per connections, and the number of requests pending to one.

You can edit the **vertx-greet DestinationRule** resource from the Red Hat OpenShift console, or run the following command in your terminal:

```
[student@workstation resilience-break]$ oc edit DestinationRule vertx-greet
```

Add the needed **trafficPolicy** entry to the resource. The **DestinationRule** must look like the following:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  ...output omitted...
  name: vertx-greet
  ...output omitted...
spec:
  host: vertx-greet
  trafficPolicy:
    connectionPool:
      http:
        http1MaxPendingRequests: 1
        maxRequestsPerConnection: 1
      tcp:
        maxConnections: 1
```

You can see the complete destination rule at **~/DO328/solutions/resilience-break/dr-connection-pool.yml**.

1.6. Verify that the pool limits pending requests, connections, and requests per connection to 1.

```
[student@workstation resilience-break]$ ./parallel.sh \
> "curl -w '%{http_code}\n' $GATEWAY_URL"
Hello World!
200
upstream connect error or disconnect/reset before headers. reset reason:
 overflow503
503
503
...output omitted...
```

The connection pool drops some of the connections, but the service still fails.

2. Deploy a new version of the service that accepts more requests.

2.1. Use the file **`deployment-v2.yaml`** provided in the **`solutions`** folder to create the new **`Deployment`** resource.

```
[student@workstation resilience-break]$ oc create \
> -f ~/DO328/solutions/resilience-break/deployment-v2.yaml
deployment.apps/vertx-greet-v2 created
```

2.2. Verify that a new pod is deployed.

```
[student@workstation resilience-break]$ oc get pods
vertx-greet-v1-5bcf556987-ssbft   2/2     Running   0          6m11s
vertx-greet-v2-6794b4bd67-qcgb5   2/2     Running   0          36s
```

Pod names are generated automatically and yours may differ.

2.3. Validate that the new pod is receiving traffic and responding as expected.

```
[student@workstation resilience-break]$ ./parallel.sh \
> "curl -w '%{http_code}\n' $GATEWAY_URL"
Hello World!
200
upstream connect error or disconnect/reset before headers. reset reason:
 overflow503
upstream connect error or disconnect/reset before headers. reset reason:
 overflow503
Hello from v2
200
Hello from v2
200
upstream connect error or disconnect/reset before headers. reset reason:
 overflow503
...output omitted...
```

3. Remove the connection pool and configure a Circuit Breaker so that OpenShift Service Mesh evicts the failing host after two failures occur during a 2 second interval. Evict hosts for a minimum of 10 seconds and ensure that OpenShift Service Mesh can evict all hosts.

3.1. Update the **DestinationRule** resource to replace the **connectionPool** entry with an **outlierDetection** entry configured with appropriate values. Again, use the Red Hat OpenShift console, or the **oc edit DestinationRule vertx-greet** command to make this update.

The **DestinationRule** must look like the following:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: vertx-greet
spec:
  host: vertx-greet
  trafficPolicy:
    outlierDetection:
      baseEjectionTime: 10.000s
      consecutiveErrors: 2
      interval: 2.000s
      maxEjectionPercent: 100
```

You can see the complete destination rule at **~/DO328/solutions/resilience-break/dr-outlier-detection.yml**.

3.2. Verify the Circuit Breaker settings evict the initial service when it receives a few requests.

You can use the same **parallel.sh** script to generate a burst of requests, but the eviction might not display in the output. Use the provided **sequential.sh** script in the labs folder to emulate intense traffic to the service.

```
[student@workstation resilience-break]$ ./sequential.sh \
> "curl -w '%{http_code}\n' $GATEWAY_URL;"
Hello from v2
200
Hello from v2
200
Hello from v2
200
Hello World!
200
Hello from v2
200
Hello from v2
200
Hello from v2
200
Hello from v2
200
Hello from v2
200
Hello from v2
200
Hello from v2
```

```
200
Hello from v2
...output omitted...
```

Note that the initial version provides some responses, but the Circuit Breaker eventually evicts the host. After eviction, no responses are received from this version of the service for approximately 10 seconds. The host eventually restores and starts sending responses again.

Keep this script running for some time, and then review the output. Notice that the host takes more and more time to recover after each eviction. Press **Ctrl+C** to stop the process.

Note also that there are no service failures, because the connection pool has been removed.

3.3. Change the current working directory to the home folder before finishing the exercise:

```
[student@workstation resilience-break]$ cd ~
[student@workstation ~]$
```

Finish

On the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab resilience-break finish
```

This concludes the guided exercise.

Lab

Building Resilient Services

Performance Checklist

In this lab, you will apply different resilience strategies to improve the reliability of an application.

Outcomes

You should be able to implement retry policies, limit connections to services, and add Circuit Breakers to improve the resilience of your applications.

Before You Begin

To perform this exercise, ensure you have access to:

- A configured and running OpenShift cluster.
- A installed and running OpenShift Service Mesh in the OpenShift cluster.
- The OpenShift CLI (**/usr/local/bin/oc**).

As the **student** user on the **workstation** machine, use the **lab** command to prepare your system for this lab.

This command deploys unreliable versions of the currency exchange application and the news application. The command also includes the Financial application into the Red Hat OpenShift Service Mesh. The source code is in the Git repository at https://github.com/RedHatTraining/DO328-apps in the **exchange-application**, and **python-flask-gossip** directories.

```
[student@workstation ~]$ lab resilience-mesh start
```

1. This activity uses scripts that are located in **~/DO328/labs/resilience-mesh**. Change to that directory.

```
[student@workstation ~]$ cd ~/DO328/labs/resilience-mesh
[student@workstation resilience-mesh]$
```

2. Log in to the OpenShift cluster as an unprivileged user and verify that the lab projects are successfully deployed.
3. Configure a circuit breaker in the external **news** service with the following characteristics:
   - A maximum of **2** consecutive errors
   - An interval between ejection sweep analysis of **5 seconds**
   - A minimum ejection duration of **10 seconds**
   - A maximum ejection percent of **100**

Name the destination rule resource **`news-circuit`** and the service entry resource **`news-se`**.

4. Test the circuit breaker configuration.
5. Configure a retry policy in the **`currencies`** service with the following characteristics.
   - Retry policy of **4** attempts
   - Retry on any **5xx** error
   - Timeout between retries of **1** second

   Name the virtual service resource **`currency-retries`**.
6. Test the retry policy applied to the **`currencies`** service.
7. Configure the the **`frontend`** service with the following connection limits:
   - A maximum of **5** pending HTTP requests
   - A maximum of **10** requests per connection
   - A maximum of **5** HTTP1/TCP connections

   Name the destination rule resource **`frontend-pool`**.
8. Test the connection limits to the **`frontend`** service.
9. Return to the home directory.

```
[student@workstation resilience-mesh]$ cd ~
[student@workstation ~]$
```

Evaluation

Grade your work by running the **`lab resilience-mesh grade`** command from your **`workstation`** machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab resilience-mesh grade
```

Finish

On the **`workstation`** machine, use the **`lab`** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab resilience-mesh finish
```

This concludes the lab.

Solution

Building Resilient Services

Performance Checklist

In this lab, you will apply different resilience strategies to improve the reliability of an application.

Outcomes

You should be able to implement retry policies, limit connections to services, and add Circuit Breakers to improve the resilience of your applications.

Before You Begin

To perform this exercise, ensure you have access to:

- A configured and running OpenShift cluster.
- A installed and running OpenShift Service Mesh in the OpenShift cluster.
- The OpenShift CLI (**/usr/local/bin/oc**).

As the **student** user on the **workstation** machine, use the **lab** command to prepare your system for this lab.

This command deploys unreliable versions of the currency exchange application and the news application. The command also includes the Financial application into the Red Hat OpenShift Service Mesh. The source code is in the Git repository at https://github.com/RedHatTraining/DO328-apps in the **exchange-application**, and **python-flask-gossip** directories.

```
[student@workstation ~]$ lab resilience-mesh start
```

1. This activity uses scripts that are located in **~/DO328/labs/resilience-mesh**. Change to that directory.

```
[student@workstation ~]$ cd ~/DO328/labs/resilience-mesh
[student@workstation resilience-mesh]$
```

2. Log in to the OpenShift cluster as an unprivileged user and verify that the lab projects are successfully deployed.

   2.1. Source the classroom configuration file that is accessible at **/usr/local/etc/ocp4.config**.

```
[student@workstation resilience-mesh]$ source /usr/local/etc/ocp4.config
```

   2.2. Log in to OpenShift as the **developer** user.

```
[student@workstation resilience-mesh]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_API}
Login successful.
...output omitted...
```

2.3. Change to the **resilience-mesh** project.

```
[student@workstation resilience-mesh]$ oc project resilience-mesh
Now using project "resilience-mesh" on server ...
```

2.4. Verify the status of the **resilience-mesh** project pods.

```
[student@workstation resilience-mesh]$ oc get pods
NAME                          READY   STATUS    RESTARTS   AGE
currency-84d75bdb8c-8g97h     2/2     Running   0          36s
exchange-5fd7954fb5-874kj     2/2     Running   0          36s
frontend-7559c7874f-fknr9     2/2     Running   0          36s
history-548b7f4954-fc9jx      2/2     Running   0          36s
```

2.5. Verify the status of the **resilience-mesh-news** project pods.

```
[student@workstation resilience-mesh]$ oc get pods -n resilience-mesh-news
NAME                            READY   STATUS    RESTARTS   AGE
news-error-6c49447695-8ktl8     1/1     Running   0          30s
news-ok-8bbf9f9c9-v8f58         1/1     Running   0          30s
```

2.6. Save the front-end route into a variable.

```
[student@workstation resilience-mesh]$ FRONTEND=$(oc get route \
> istio-ingressgateway -n istio-system \
> -o jsonpath='{"http://"}{.spec.host}{"/frontend"}')
```

2.7. Access the lab application using the **Firefox** web browser on your **workstation** machine to check the unreliable behavior of the application.

```
[student@workstation resilience-mesh]$ firefox ${FRONTEND}
```

The Historical Data and Exchange pages rely on the **currency** service. This service returns a failure response for two of three requests.

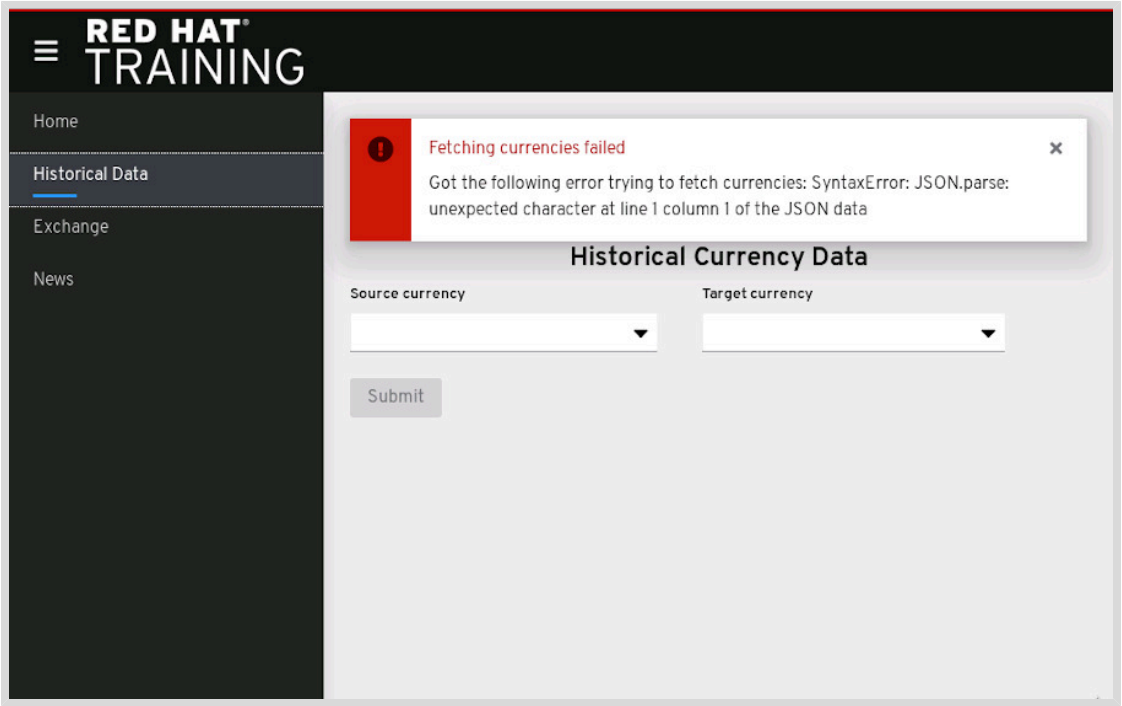


Figure Error.1: Front end displaying errors generated by the currency service

2.8. Navigate between the Historical Data and Exchange pages several times to observe this behavior.

The News page sometimes fails to fetch data from the external service. Reload the page several times to observe this behavior.

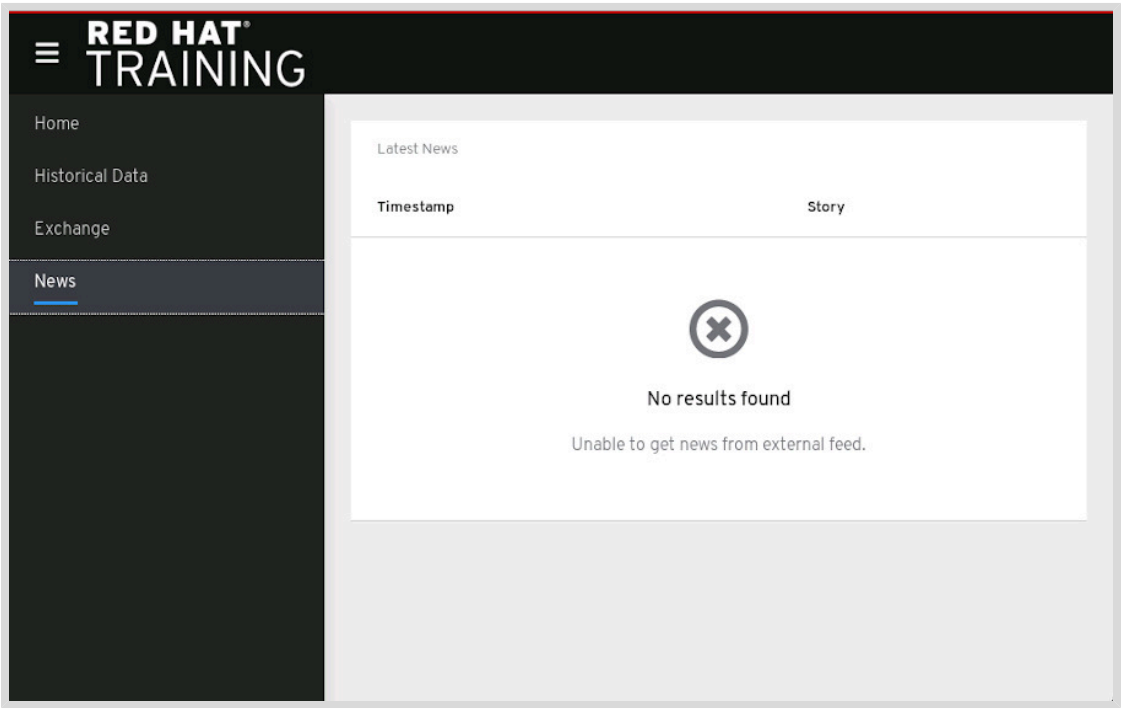


Figure Error.2: News page not receiving data from the external service

3. Configure a circuit breaker in the external **news** service with the following characteristics:

- A maximum of **2** consecutive errors
- An interval between ejection sweep analysis of **5 seconds**
- A minimum ejection duration of **10 seconds**
- A maximum ejection percent of **100**

Name the destination rule resource **news-circuit** and the service entry resource **news-se**.

3.1. Get the **news** service host name.

```
[student@workstation resilience-mesh]$ oc get route news -n resilience-mesh-news \
> -o jsonpath='{.spec.host}{"\n"}'
news-resilience-mesh-news.apps.ocp4.example.com
```

Use the news service host in the destination rule and service entry you are going to create in the following steps.

3.2. Create a service entry object YAML file, for example **service-entry.yaml**, to store the object definition.

The completed object definition is available in the **~/DO328/solutions/resilience-mesh/service-entry.yaml** file. You can use the YAML file to verify your file and fix mistakes.

3.3. Create the service entry configuration using the **oc create** command.

```
[student@workstation resilience-mesh]$ oc create -f service-entry.yaml
serviceentry.networking.istio.io/news-se created
```

3.4. Create a destination rule object YAML file, for example **circuit-breaker.yaml**, to store the object definition.

The completed object definition is available in the **~/DO328/solutions/resilience-mesh/circuit-breaker.yaml** file. You can use the YAML file to verify your file and fix mistakes.

3.5. Create the destination rule configuration using the **oc create** command.

```
[student@workstation resilience-mesh]$ oc create -f circuit-breaker.yaml
destinationrule.networking.istio.io/news-circuit created
```

4. Test the circuit breaker configuration.

4.1. Open the lab application in your browser and access the News page.

```
[student@workstation resilience-mesh]$ firefox ${FRONTEND}
```

4.2. Reload the News page several times to verify that it always displays news. It may take several seconds to OpenShift Service Mesh to propagate the new configuration.

The circuit breaker detects failures in the connections to the external news feed and removes the problematic hosts.

5. Configure a retry policy in the **currencies** service with the following characteristics.

   - Retry policy of **4** attempts
   - Retry on any **5xx** error
   - Timeout between retries of **1** second

   Name the virtual service resource **currency-retries**.

   5.1. Create a virtual service object YAML file, for example **retries.yaml**, to store the object definition.

   The completed object definition is available in the **~/DO328/solutions/resilience-mesh/retries.yaml** file. You can use the YAML file to verify your file and fix mistakes.

   5.2. Create the service entry configuration with the **oc create** command.

```
[student@workstation resilience-mesh]$ oc create -f retries.yaml
virtualservice.networking.istio.io/currency-retries created
```

6. Test the retry policy applied to the **currencies** service.

   6.1. Open the lab application in your browser and access to the Historical Data or Exchange pages.

```
[student@workstation resilience-mesh]$ firefox ${FRONTEND}
```

   6.2. Navigate through the application pages to verify that all pages are working.

7. Configure the the **frontend** service with the following connection limits:

   - A maximum of **5** pending HTTP requests
   - A maximum of **10** requests per connection
   - A maximum of **5** HTTP1/TCP connections

   Name the destination rule resource **frontend-pool**.

   7.1. Create a destination rule object YAML file, for example **connection-pool.yaml**, to store the object definition.

   The completed object definition is available in the **~/DO328/solutions/resilience-mesh/connection-pool.yaml** file. You can use the YAML file to verify your file and fix mistakes.

   7.2. Create the service entry configuration using the **oc create** command.

```
[student@workstation resilience-mesh]$ oc create -f connection-pool.yaml
destinationrule.networking.istio.io/frontend-pool created
```

8. Test the connection limits to the **frontend** service.

   8.1. Examine the **parallel-requests.sh** script. This script generates parallel connections to the front-end service and prints the HTTP code from the response.

Run the **`parallel-requests.sh`** script to send a small amount of traffic to the front-end service.

```
[student@workstation resilience-mesh]$ sh parallel-requests.sh 2
HTTP code: 200
HTTP code: 200
```

8.2. Run the **`parallel-requests.sh`** script to send a large amount of traffic to the front-end service.

```
[student@workstation resilience-mesh]$ sh parallel-requests.sh 33
HTTP code: 200
HTTP code: 200
...output omitted...
HTTP code: 200
HTTP code: 503
HTTP code: 503
HTTP code: 503
...output omitted...
```

The preceding command shows a mix of 200 and 503 errors. The connection pool settings are limiting the connections so that some responses are 503 errors.

9. Return to the home directory.

```
[student@workstation resilience-mesh]$ cd ~
[student@workstation ~]$
```

Evaluation

Grade your work by running the **`lab resilience-mesh grade`** command from your **`workstation`** machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab resilience-mesh grade
```

Finish

On the **`workstation`** machine, use the **`lab`** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab resilience-mesh finish
```

This concludes the lab.

Summary

In this chapter, you learned:

- With OpenShift Service Mesh you can implement different resilience strategies without changing your code.
- Load balancing prevents service overloads by distributing the load among several service replicas.
- Time-outs guarantee that a network request finishes within a specified time limit.
- Retries focuses on reducing transient, short-lived communication failures.
- Circuit Breaker avoids directing requests to a failing host.
- Another technique to protect hosts from failing is limiting the number of simultaneous connections to the host.

Chapter 8

Securing an OpenShift Service Mesh

Goal

Secure and encrypt services in your application with Red Hat OpenShift Service Mesh.

Objectives

- Describe how Citadel manages identities.
- Configure Mutual TLS to secure intra-service communication.
- Configure restriction on services communication in OpenShift Service Mesh.

Sections

- Describing the Role of Citadel in OpenShift Service Mesh (and Quiz)
- Configuring Mutual TLS (and Guided Exercise)
- Defining Service to Service Authorization (and Guided Exercise)

Lab

Securing an OpenShift Service Mesh

Describing the Role of Citadel in OpenShift Service Mesh

Objectives

After completing this section, you should be able to describe how Citadel manages identities.

Describing Security of the Microservice Architecture

As organizations move to cloud-native applications with DevOps principles, security practices often change. Some traditional security concepts still apply, but can become difficult to implement. For example, traditional firewall security perimeters operate on the network transport layer (L4), which becomes difficult to implement and maintain in a cloud-native environment.

Cloud-native platforms, such as Red Hat OpenShift, introduce a number of new security concepts, such as verifying and trusting your container registry, adapting network security to software defined networking, identifying and preventing unintended egress requests, and similar. As teams continue iterating upon their application, security is often deprioritized in favor of new feature development.

Deprioritizing security can lead to:

- Outages
- Data breaches
- Legal liability and noncompliance
- Slower time to production

Security is always a priority in the Red Hat ecosystem. Red Hat products and services encourage adopting *DevSecOps* principles.

DevSecOps is an evolution of DevOps principles. DevSecOps integrates security in the software development design and implementation loops from the beginning. Security becomes a responsibility of everyone involved in the software development cycle.

Security and Red Hat OpenShift Service Mesh

OpenShift Service Mesh enables developers and system administrators to abstract security away from application code and into infrastructure configuration, enabling zero-trust perimeters and deny-by-default behavior.

OpenShift Service Mesh provides, among others, the following security features:

Cryptographically Verifiable Service Identities

Red Hat OpenShift assigns each service a service account by default. However, this identity is not cryptographically verifiable. OpenShift Service Mesh provides X.509 identities with advanced security features to verify service identity.

Security features, such as internal traffic encryption, verify services using the service identities.

Enabling traffic encryption means only the origin and target services can decrypt the traffic. Consequently, this mitigates the impact of possible security breaches, and reduces avenues for attack, or attack vectors.

Service authentication

A service inside of Red Hat OpenShift cluster can communicate with any other service by using the resolvable DNS name, such as ***`svc-name.project-name.dns-domain`***.

Unrestricted communication can lead to attackers communicating with services to which they should not have access. Depending on the OpenShift cluster architecture, attackers might also compromise sensitive data to external servers.

In a large service mesh, developers might also discover that services contact legacy endpoints or other unexpected services. Because legacy endpoints often become deprecated or decommissioned, depending on legacy endpoints may cause outages.

OpenShift Service Mesh enables you to explicitly permit or deny service-to-service communication, further reducing potential attack vectors and providing clarity into the service mesh communication patterns in your cluster.

Citadel in OpenShift Service Mesh

Citadel is a crucial part of OpenShift Service Mesh, responsible for the public key infrastructure (PKI). OpenShift Service Mesh uses Citadel to provision X.509 identities for each workload, which are then used for security features, such as authentication and authorization settings, or mutual TLS.

Citadel enables the following:

- *Self-rotating PKI* ensures that Citadel automatically rotates certificates within its PKI that are about to expire. Administrators do not manually manage certificates issued by Citadel.

  The self-rotating PKI aim to reduce the maintenance costs connected with manual PKI management, and to avoid possible outages related to certificate expiry and misconfiguration.

- *Certificate-key pair injection* ensures that Citadel mounts necessary certificates into any pod that OpenShift Service Mesh manages. For example, for any new pod joining OpenShift Service Mesh, Citadel generates a certificate-key pair identity, mounts it into the pod, and stores the identity information for further security settings.

  The automatic certificate-key pair injection enables OpenShift Service Mesh to automatically assign identities for workloads. The identities are later used for authorization and authentication features.

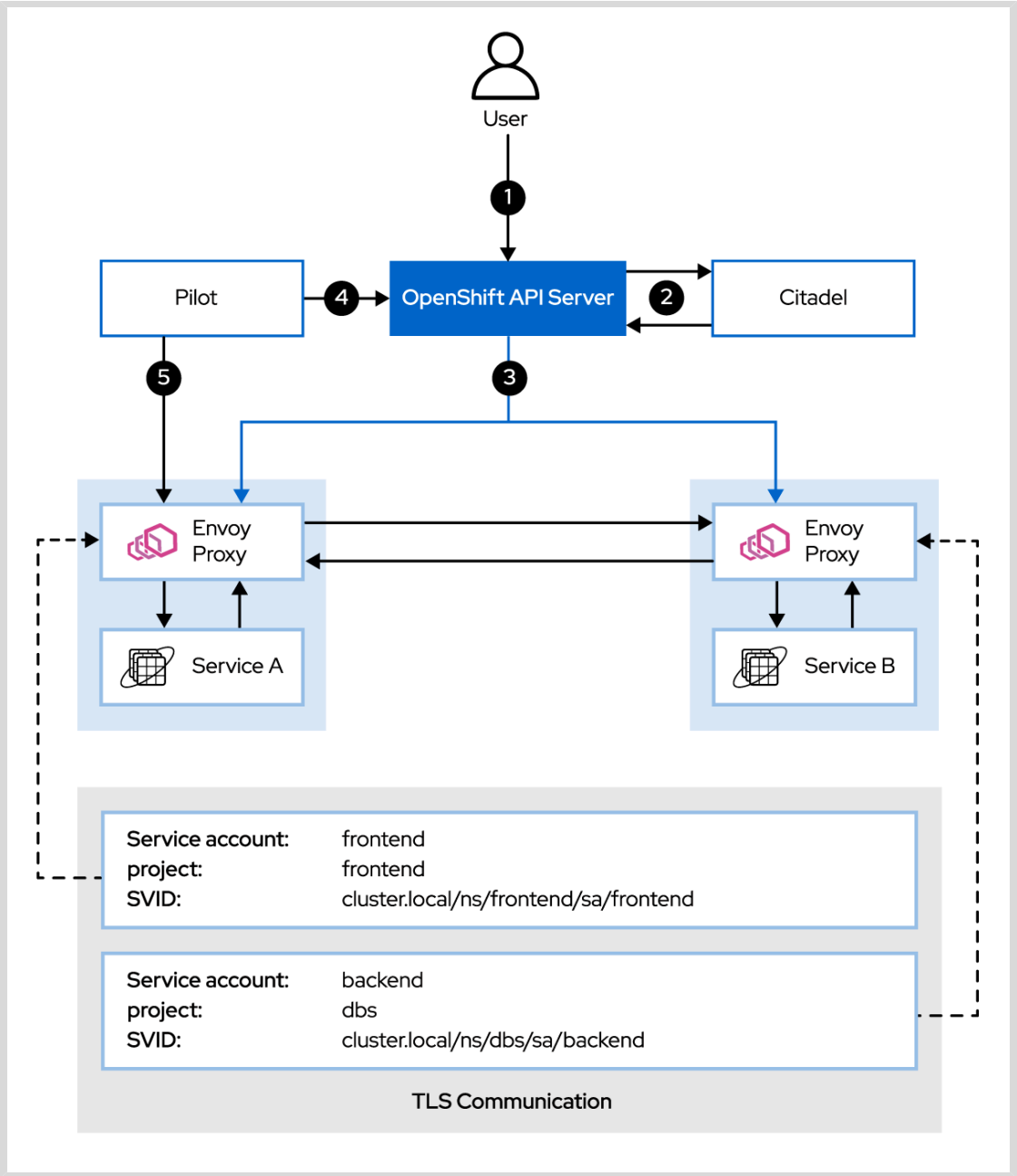
Citadel-Issued Identity Management

When you configure OpenShift Service Mesh to manage a project, Citadel generates a certificate-key pair for the default project service accounts. OpenShift Service Mesh stores the certificate-key pair, together with the certificate chain, as a secret named **`istio.`*`service_account_name`***. The secret is mounted into any pods containing the Envoy proxy container.

```
[student@workstation ~]$ oc get secret
NAME                           TYPE                                  DATA   AGE
istio.builder                  istio.io/key-and-cert                 3      98m
istio.default                  istio.io/key-and-cert                 3      98m
istio.deployer                 istio.io/key-and-cert                 3      98m
...output omitted...
```

All services in a project use the same service account by default. Therefore, all services within one project encrypt their communication with the identical private key, meaning that a malicious container injected into the project can decrypt all traffic in the project. Although the attack surface is reduced to the same project traffic, Red Hat recommends creating a service account for each microservice in a project to further reduce the attack surface.

Citadel generates a new certificate-key pair for each service account in a project. OpenShift Service Mesh mounts the certificate-key pair into pods associated with the service account. Red Hat recommends generating a unique service account for each microservice in your project so that a malicious process cannot decrypt any traffic in the project.



1. User deploys two new pods with separate service accounts.
2. Citadel generates a new certificate-key pair for each service account.
3. OpenShift Service Mesh mounts certificate-key pairs into the Envoy proxy container of each pod.
4. Pilot creates service-identity mapping for secure naming.
5. Pilot pushes secure naming information into Envoy proxy containers of each pod.

Service A and *Service B* can communicate with each other using TLS encryption based on Citadel-issued identities.

Identity Verification

When Citadel generates a certificate-key pair for a service account, it uses the *Secure Production Identity Framework for Everyone* (SPIFFE) to provide further security checks. SPIFFE is a

specification maintained by the Cloud Native Computing Foundation (CNCF). SPIFFE is designed to cryptographically verify that the certificate identity presented by a microservice matches with the microservice service account.

Each certificate is encoded with *SPIFFE Verifiable Identity Document* (SVID). The SVID takes the form of **`cluster_domain/ns/project_name/sa/service_account_name`**, and is encoded into the certificate as the **`X509v3 Subject Alternative Name`** parameter.

The role of SPIFFE is to mitigate the attack surface for identity theft. For example, if a malicious user gains access to the certificate-key pair of the *frontend* microservice, and finds an exploit to execute arbitrary code in the *gateway* microservice, then SPIFFE checks might stop the attack. Because the service account name is encoded in each certificate, setting a unique service account for both the microservices ensures that the SPIFFE check fails and Envoy proxy fails to complete the TLS handshake.

OpenShift Service Mesh provides additional security checks, referred to as *secure naming*. At run time, the Pilot component checks whether a service uses the correct service account before SPIFFE is checked. Consequently, because Pilot expects a response from one service, but receives response from a different service, if you set each a different service account for each service, then the Envoy proxy rejects the request because it originated from a service other than the expected (mapped) one. Secure naming is useful when an attacker compromises network routing in your OpenShift cluster and can reroute traffic from one service to another service.

References

What is DevSecOps?
https://www.redhat.com/en/topics/devops/what-is-devsecops

Understanding Red Hat OpenShift Service Mesh
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.4/html-single/service_mesh/index#understanding-service-mesh

Istio High-level architecture
https://archive.istio.io/v1.4/docs/concepts/security/#high-level-architecture

Istio PKI
https://archive.istio.io/v1.4/docs/concepts/security/#pki

Istio Secure Naming
https://archive.istio.io/v1.4/docs/concepts/security/#secure-naming

Understanding and creating service accounts
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.4/html-single/authentication/index#understanding-and-creating-service-accounts

Quiz

Describing the Role of Citadel in OpenShift Service Mesh

Choose the correct answers to the following questions:

1. **Which two of the following statements about issuing certificate-key pair identity are correct? (Choose two.)**
   a. Citadel generates a new certificate-key pair when a user deploys a new pod in an already existing project.
   b. Citadel generates a new certificate-key pair when a user creates a new service account in an already existing project.
   c. Citadel generates a new certificate-key pair when a user creates a new project.
   d. Citadel generates a new certificate-key pair when a user requests a new identity.

2. **Which of the following two statements about microservice security with OpenShift Service Mesh are recommended practices? (Choose two.)**
   a. Enable traffic encryption in each project. For example, using TLS and unique service accounts for each pod.
   b. Deploy each pod into a unique project so that malicious pods cannot decrypt internal traffic of other pods.
   c. Encode a custom SVID into each pod identity so that malicious containers are easily spotted.
   d. Enable deny-by-default practices, and refuse communication with any pods with identities that you cannot cryptographically verify.

3. **An application consists of four services: a front end service, a gateway service, and two back end services. The front end service serves a web page to the end user. The web page then sends requests to the gateway service, which communicates with back end services and replies directly to the web page.**
   Assuming that Red Hat OpenShift Service Mesh manages all of the services, how many service accounts should be created in the application project?
   a. One, only for the front end service, because no other pod is directly exposed to the end user.
   b. Two, for both the front end and gateway services, because they are exposed to the end user.
   c. Two, for both of the back end services, because they might contain sensitive information.
   d. Four, one for each of the services because every service must be cryptographically verifiable.

4. How does Red Hat Service Mesh store Citadel-generated identities?

a. Using OpenShift secrets, named **`istio.`*`service_account_name`***.

b. Using an X.509 certificate-key pair, mounted into each pod.

c. Pilot mounts Citadel-generated identities into each pod.

d. OpenShift cluster administrator stores each identity outside of the cluster.

Solution

Describing the Role of Citadel in OpenShift Service Mesh

Choose the correct answers to the following questions:

1. **Which two of the following statements about issuing certificate-key pair identity are correct? (Choose two.)**
   a. Citadel generates a new certificate-key pair when a user deploys a new pod in an already existing project.
   b. Citadel generates a new certificate-key pair when a user creates a new service account in an already existing project.
   c. Citadel generates a new certificate-key pair when a user creates a new project.
   d. Citadel generates a new certificate-key pair when a user requests a new identity.

2. **Which of the following two statements about microservice security with OpenShift Service Mesh are recommended practices? (Choose two.)**
   a. Enable traffic encryption in each project. For example, using TLS and unique service accounts for each pod.
   b. Deploy each pod into a unique project so that malicious pods cannot decrypt internal traffic of other pods.
   c. Encode a custom SVID into each pod identity so that malicious containers are easily spotted.
   d. Enable deny-by-default practices, and refuse communication with any pods with identities that you cannot cryptographically verify.

3. **An application consists of four services: a front end service, a gateway service, and two back end services. The front end service serves a web page to the end user. The web page then sends requests to the gateway service, which communicates with back end services and replies directly to the web page.**
   Assuming that Red Hat OpenShift Service Mesh manages all of the services, how many service accounts should be created in the application project?
   a. One, only for the front end service, because no other pod is directly exposed to the end user.
   b. Two, for both the front end and gateway services, because they are exposed to the end user.
   c. Two, for both of the back end services, because they might contain sensitive information.
   d. Four, one for each of the services because every service must be cryptographically verifiable.

4. How does Red Hat Service Mesh store Citadel-generated identities?

a. Using OpenShift secrets, named **`istio.`*`service_account_name`***.

b. Using an X.509 certificate-key pair, mounted into each pod.

c. Pilot mounts Citadel-generated identities into each pod.

d. OpenShift cluster administrator stores each identity outside of the cluster.

Configuring Mutual TLS

Objectives

After completing this section, you should be able to configure Mutual TLS to secure intra-service communication.

Describing Mutual TLS in OpenShift Service Mesh

Mutual TLS (mTLS) is a security feature provided by OpenShift Service Mesh. Enabling mTLS results in encrypted traffic between Envoy proxy containers. Every service injected with Envoy proxy can perform plain text as well as TLS-encrypted requests.

Enabling mTLS reduces the attack surface of your Red Hat OpenShift cluster. For example, mTLS mitigates man-in-the-middle attacks between the Red Hat OpenShift nodes by securing internal (also called east-west) traffic. If an attacker injects a malicious container into your service mesh, due to a compromised external container registry, for example, this container cannot decrypt the encrypted network packets. Enabling mTLS also provides a cryptographically verifiable identity, which serves to further mitigate the attack vector of potential malicious containers. Using cryptographic identities to further reduce the attack surface is explored in later sections.

Red Hat OpenShift does not provide any mechanism to enable encryption for internal communication automatically. Cloud administrators who want to adhere to the DevSecOps principles, such as implementing zero-trust network perimeters, must manually provision and maintain a *Public Key Infrastracture* (PKI). The application developers must adjust the application code to communicate using TLS protocols.

OpenShift Service Mesh enables you to utilize a ready-made and automatically managed PKI infrastructure. Because TLS is used between Envoy proxies, the application code requires no modifications to utilize TLS communication. OpenShift Service Mesh manages and rotates X.509 certificates automatically. Consequently, expired certificates do not result in unresponsive services.

Mutual TLS Modes

You can enable mTLS in one of two modes:

- Permissive (default)
- Strict

The *permissive* mode enables Envoy proxies to accept either HTTP or mTLS encrypted traffic. The permissive mode is the default mTLS mode because it enables communication with services that are not injected with Envoy proxies. In the permissive mode, you can manually enable mTLS in projects that do not communicate with services outside of the OpenShift Service Mesh.

The *strict* mode forbids Envoy proxies to accept HTTP traffic. Consequently, services using plain text requests are unable to communicate with services in OpenShift Service Mesh. Note that the strict mode might also disable certain features of OpenShift Service Mesh.

Use the strict mode when you want to enforce zero-trust perimeter, and your services have no dependencies on services without the Envoy proxy sidecar container. Permissive mode is

especially useful when you are migrating some of your services to OpenShift Service Mesh, or for testing. Even when OpenShift Service Mesh uses the permissive mode, you can set your services to communicate using mTLS.

Configuring Mutual TLS

Configuring Mutual TLS Globally

You can configure global enforcement of mTLS by modifying the **`servicemeshcontrolplane`** resource. Set the **`.spec.istio.global.mtls.enabled`** property to **`true`** to enable **`strict`** enforcement mode:

```
[student@demo ~]$ oc edit servicemeshcontrolplane basic-install -n istio-system
apiVersion: maistra.io/v1
kind: ServiceMeshControlPlane
metadata:
  creationTimestamp: "2020-04-15T10:34:31Z"
  finalizers:
  - maistra.io/istio-operator
  generation: 2
  name: basic-install
  namespace: istio-system
spec:
  istio:
    global:
      mtls:
        enabled: true (1)
    grafana:
      enabled: true
```

(1) The **`.spec.istio.global.mtls.enabled`** property of the **`servicemeshcontrolplane`** resource.

The **`.spec.istio.global.mtls.enabled`** property sets all Envoy proxies to both accept and send strictly mTLS requests. The default setting is **`false`**.

Configuring Mutual TLS per Project

When you set the global mTLS mode to **`PERMISSIVE`**, you can enforce mTLS on the project level using the **`Policy`** resource:

```
apiVersion: authentication.istio.io/v1alpha1
kind: Policy
metadata:
  name: default
spec:
  peers:
  - mtls:
      mode: STRICT
```

When you create a **`Policy`** resource, you configure Envoy proxies to accept only mTLS-encrypted requests. However, without further configuration, Envoy proxies send plain text HTTP requests. To configure Envoy proxies to also strictly send mTLS requests, use the **`DestinationRule`** resource:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: dr-mtls
spec:
  host: * (1)
  trafficPolicy:
    tls:
      mode: ISTIO_MUTUAL (2)
```

(1) This destination rule applies to requests for all hosts
(2) Requests use mutual TLS

Verifying State of Mutual TLS

When you configure services to use mTLS, you might find that some services stop responding. This is often the result of conflicting configuration of services, where one service strictly accepts mTLS requests while another strictly accepts plain HTTP requests. You might also want to verify your mTLS service configuration to ensure that the configuration settings are properly applied.

You can both troubleshoot and verify mTLS configuration using the *istioctl* command-line utility. You can use the **`istioctl authn tls-check`** ***`POD_NAME`*** command to verify Envoy proxy mTLS settings.

Verifying Service Identity

Any service using mTLS contains a set of X.509 certificates. The Citadel stores the certificates as a secret with the name of **`istio.`*`service_account_name`***, for example:

```
[student@demo ~]$ oc get secrets -n istio-system | grep istio\\.
istio.builder                          istio.io/key-and-cert    3    45h
istio.default                          istio.io/key-and-cert    3    45h
istio.deployer                         istio.io/key-and-cert    3    45h
istio.grafana                          istio.io/key-and-cert    3    45h
istio.istio-citadel-service-account    istio.io/key-and-cert    3    45h
...output omitted...
```

You can examine each certificate using the **`openssl`** utility:

```
[student@demo ~]$ oc get secret -n istio-system \
> istio.istio-ingressgateway-service-account \
> -o json | jq -r '.data["cert-chain.pem"]' | \
> base64 --decode | openssl x509 -in /dev/stdin -text -noout
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            8a:da:c5:65:3b:52:b8:90:d6:1b:f1:7d:57:0d:3b:9e
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: O = cluster.local
        Validity
            Not Before: Apr 15 10:35:06 2020 GMT
            Not After : Jul 14 10:35:06 2020 GMT
```

```
        Subject:
        Subject Public Key Info:
...output omitted...
```

OpenShift Service Mesh mounts the secret into each Envoy proxy at **/etc/certs**:

```
[student@demo ~]$ oc exec frontend-6877597c88-mscmh \
> -c istio-proxy (1) -- ls /etc/certs
cert-chain.pem (2)
key.pem (3)
root-cert.pem (4)
```

1. The Envoy proxy container is named **istio-proxy** by default.
2. Envoy certificate that is presented to other Envoy proxies in order to perform TLS handshake.
3. Envoy's private key, used for decrypting TLS traffic. Forms a certificate-key pair with **cert-chain.pem**.
4. The root certificate to verify other **cert-chain.pem** certificates. Each Citadel manages its own root certificate.

Verifying service identity is useful when you change the Citadel certificate authority to your organization's certificate authority, or when troubleshooting lower-level network issues, such as failure to perform TLS handshake between two services. You can also verify the SPIFFE identity encoded in the certificate.

End-to-End TLS

Configuring end-to-end TLS means the communication between an external client and a service is fully encrypted. However, to utilize all features of OpenShift Service Mesh, you must use the **ingressgateway** pods.

Using one route to expose multiple different services is difficult because a single route can have only one set of certificates.

You can solve the issue by deploying a separate ingress gateway for each service that you want to expose to external clients. Alternatively, you can create a *passthrough-terminated* TLS route for each exposed service and mount the certificates for each route into the default **ingressgateway** pod.

To secure your application using a passthrough-terminated TLS route:

1. Create a **tls** secret with the contents of your certificate-key pair.

```
[student@demo ~]$ oc -n istio-system create secret tls istio-ingressgateway-
customer-certs --key customer.key --cert customer.crt
secret/istio-ingressgateway-customer-certs created
```

2. Mount the certificates into the **istio-ingressgateway** pod.

    a. Prepare a patch that mounts the **istio-ingressgateway-customer-certs** secret into the **istio-ingressgateway** pod.

```
[student@demo ~]$ cat > gateway-patch.json << EOF
> [{
>   "op": "add",
>   "path": "/spec/template/spec/containers/0/volumeMounts/0",
```

```
>   "value": {
>     "mountPath": "/etc/istio/customer-certs",
>     "name": "customer-certs",
>     "readOnly": true
>   }
> },
> {
>   "op": "add",
>   "path": "/spec/template/spec/volumes/0",
>   "value": {
>   "name": "customer-certs",
>     "secret": {
>       "secretName": "istio-ingressgateway-customer-certs",
>       "optional": true
>     }
>   }
> }]
> EOF
```

b. Apply the patch.

```
[student@demo ~]$ oc -n istio-system patch --type=json deploy istio-ingressgateway
 -p "$(cat gateway-patch.json)"
deployment.apps/istio-ingressgateway patched
```

c. Verify certificates are mounted in the **`ingressgateway`** container.

```
[student@demo ~]$ INGRESS_POD=$(oc -n istio-system get pods -l
 istio=ingressgateway -o jsonpath='{.items..metadata.name}')
[student@demo ~]$ oc -n istio-system exec $INGRESS_POD -- ls /etc/istio/customer-
certs
tls.crt
tls.key
```

3. Create a gateway that accepts connections using the host of your exposed application, that is, the resolvable host name used for ingress on the HTTPS port

a. Prepare the gateway yaml file:

```
[student@demo ~]$ cat > gateway.yml <<EOF
> apiVersion: networking.istio.io/v1alpha3
> kind: Gateway
> metadata:
>   name: customer-gateway
> spec:
>   selector:
>     istio: ingressgateway (1)
>   servers:
>   - port: (2)
>       number: 443
>       name: https-customer
>       protocol: HTTPS
>     tls:
```

```
>       mode: SIMPLE
>       serverCertificate: /etc/istio/customer-certs/tls.crt
>       privateKey: /etc/istio/customer-certs/tls.key
>     hosts:
>     - "customer.com"
> EOF
```

1. The gateway targets the **istio-ingressgateway** pod by the label **istio=ingressgateway**. The **istio-ingressgateway** pod contains the mounted certificate-key pair.
2. The gateway accepts connections on the HTTPS port **443** using the HTTPS protocol.
3. The **SIMPLE** TLS mode means the server does not verify client certificates. Only the client verifies the server authority.
4. Provide path to the mounted certificate-key pair. The route presents this certificate to clients.
5. The host of the path. This path responds to **https://customer.com**. Ensure clients can correctly resolve this domain.

b. Apply the gateway:

```
[student@demo ~]$ oc create -f gateway.yml
gateway.networking.istio.io/customer-gateway created
```

4. Create a virtual service that responds to the **customer-gateway** gateway.

a. Prepare the virtual service yaml file:

```
[student@demo ~]$ cat > vservice.yml << EOF
> apiVersion: networking.istio.io/v1alpha3
> kind: VirtualService
> metadata:
>   name: customer-virtualservice
> spec:
>   hosts:
>   - "customer.com"
>   gateways:
>   - customer-gateway
>   http:
>   - match:
>     - uri:
>         prefix: /
>     route:
>     - destination:
>         host: customer
>         port:
>           number: 8080
> EOF
```

b. Apply the virtual service:

```
[student@demo ~]$ oc create -f vservice.yml
virtualservice.networking.istio.io/customer-virtualservice created
```

5. Create a **passthrough-terminated** TLS route in the **istio-system** project that listens to the domain that you selected earlier. The route redirects requests to the **istio-ingressgateway** service.

   a. Prepare the route yaml file:

```
[student@demo ~]$ cat > route.yml << EOF
> apiVersion: route.openshift.io/v1
> kind: Route
> metadata:
>   labels:
>     app: istio-ingressgateway
>   name: customer-https-route
>   namespace: istio-system
> spec:
>   host: customer.com
>   port:
>     targetPort: https
>   tls:
>     insecureEdgeTerminationPolicy: None
>     termination: passthrough
>   to:
>     kind: Service
>     name: istio-ingressgateway
>     weight: 100
>   wildcardPolicy: None
> EOF
```

   b. Apply the route:

```
[student@demo ~]$ oc create -f route.yml
route.route.openshift.io/customer-https-route created
```

6. Verify the secure ingress:

```
[student@demo ~]$ curl https://customer.com/
customer => preference => recommendation v1 from 'f11b097f1dd0': 2
```

References

Mutual TLS Deep-Dive
https://archive.istio.io/v1.4/docs/tasks/security/authentication/mutual-tls/

Automatic mutual TLS
https://archive.istio.io/v1.4/docs/tasks/security/authentication/auto-mtls/

Mutual TLS Migration
https://archive.istio.io/v1.4/docs/tasks/security/authentication/mtls-migration/

Gateway
https://archive.istio.io/v1.4/docs/reference/config/networking/gateway/#Server-TLSOptions

Configuring Routes
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.4/html-single/networking#configuring-routes

For more information, refer to the *Mutual Transport Layer Security (mTLS)* chapter in the *Introducing Istio Mesh for Microservices* book at https://developers.redhat.com/books/introducing-istio-service-mesh-microservices/

Guided Exercise

Configuring Mutual TLS

In this exercise, you will configure OpenShift Service Mesh to use mutual TLS (mTLS).

Outcomes

You should be able to:

- Verify mTLS state in the OpenShift Service Mesh cluster.
- Enable mTLS in a specific project using the *Policy* and *DestinationRule* resources.
- Validate mTLS state using the *istioctl* command-line client.

Before You Begin

To perform this exercise, ensure you have:

- A configured and running OpenShift cluster.
- A installed and running OpenShift Service Mesh in the OpenShift cluster.
- The OpenShift CLI (**`/usr/local/bin/oc`**).

As the **`student`** user on the **`workstation`** machine, use the **`lab`** command to prepare your system for this exercise.

This command ensures that:

- Your OpenShift cluster meets the requirements.
- Microservices used in this guided exercise are present on your cluster, and ready to use.
- You have access to the solution files on the **`workstation`** machine.

```
[student@workstation ~]$ lab secure-mtls start
```

1. Log in to OpenShift and verify that the sample project has been successfully deployed.

   1.1. Run the following command to load the environment variables created in the first guided exercise:

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

   1.2. Log in to OpenShift:

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_API}
Login successful.
...output omitted...
```

1.3. Change to project **mtls**:

```
[student@workstation ~]$ oc project mtls
Now using project "mtls" on server ...
```

1.4. Verify that the pods are ready:

```
[student@workstation ~]$ oc get pods
NAME                              READY   STATUS    RESTARTS   AGE
customer-6948b8b959-cl5zf         2/2     Running   0          11s
preference-6d5d86cb79-9cjkv       2/2     Running   0          11s
recommendation-69db8d6c48-wrkc6   2/2     Running   0          11s
```

1.5. Save the **ingress-gateway** route host name with the **/mtls** endpoint into a variable:

```
[student@workstation ~]$ INGRESS_HOST=$(oc get route -n istio-system \
> istio-ingressgateway -o jsonpath='{.spec.host}'/mtls)
```

1.6. Verify that the service responds using the **INGRESS_HOST** variable:

```
[student@workstation ~]$ curl $INGRESS_HOST
customer => preference => recommendation v1 from 'f11b097f1dd0': 1
```

The start script deploys the **customer**, **preference**, and **recommendation** services into your Red Hat OpenShift cluster. The source code is in the Git repository at `https://github.com/RedHatTraining/DO328-apps/` in the **customer**, **preference**, and **recommendation** directories.

2. Verify the state of mTLS in your service mesh using the **istioctl** utility.

2.1. Save the customer pod name into a variable:

```
[student@workstation ~]$ CUSTOMER_POD=$(oc get pods -l app=customer \
>  -o jsonpath={.items..metadata.name})
```

2.2. Examine the **istioctl** output using the **CUSTOMER_POD** variable:

```
[student@workstation ~]$ istioctl authn tls-check $CUSTOMER_POD
HOST:PORT                                  STATUS  SERVER
customer.mtls.svc.cluster.local:8080       OK      PERMISSIVE
grafana.istio-system.svc.cluster.local:3000   OK      DISABLE
...output omitted...
```

Mutual TLS is disabled from the client side. The customer pod can communicate with all of the services because most services are set to the **PERMISSIVE** mode. Services in the **PERMISSIVE** mode accept both HTTP and mTLS connections.

3. Enforce use of mTLS in the **mtls** project.

3.1. Create the **policy.yml** file:

```
[student@workstation ~]$ cat > policy.yml << EOF
> apiVersion: authentication.istio.io/v1alpha1
> kind: Policy
> metadata:
>   name: default
> spec:
>   peers:
>   - mtls:
>       mode: STRICT
> EOF
```

You can see the complete resource at **~/DO328/solutions/secure-mtls/policy.yml**.

3.2. Apply the **Policy** resource:

```
[student@workstation ~]$ oc create -f policy.yml
policy.authentication.istio.io/default created
```

4. Recheck the mTLS status in the **mtls** project.

4.1. Check the state of mTLS in your service mesh:

```
[student@workstation ~]$ istioctl authn tls-check $CUSTOMER_POD | grep mtls.svc
customer.mtls.svc.cluster.local:8080          OK    STRICT    -
preference.mtls.svc.cluster.local:8080        OK    STRICT    -
recommendation.mtls.svc.cluster.local:8080    OK    STRICT    -
...output omitted...
```

Because the **default** policy is now in the **STRICT** mode, all services within the **mtls** project accept only TLS requests. However, the services in the **mtls** project send only HTTP requests. Consequently, communication in the **mtls** project is forbidden.

4.2. Check that the service no longer responds to requests:

```
[student@workstation ~]$ curl $INGRESS_HOST
upstream connect error or disconnect/reset before headers. reset reason:
 connection termination
```

5. Resolve the mTLS conflict within the **mtls** project.

5.1. Create the **destrule.yml** file:

```
[student@workstation ~]$ cat > destrule.yml << EOF
> apiVersion: "networking.istio.io/v1alpha3"
> kind: "DestinationRule"
> metadata:
>   name: "default"
> spec:
>   host: "*.mtls.svc.cluster.local"
>   trafficPolicy:
>     tls:
```

```
>       mode: ISTIO_MUTUAL
> EOF
```

You can see the complete resource at **~/DO328/solutions/secure-mtls/dr.yml**.

5.2. Apply the destination rule:

```
[student@workstation ~]$ oc create -f destrule.yml
destinationrule.networking.istio.io/default created
```

5.3. Verify that there is no longer conflict within the **mtls** project using the **istioctl** client:

```
[student@workstation ~]$ istioctl authn tls-check $CUSTOMER_POD | grep mtls.svc
customer.mtls.svc.cluster.local:8080          OK     STRICT     ISTIO_MUTUAL
preference.mtls.svc.cluster.local:8080        OK     STRICT     ISTIO_MUTUAL
recommendation.mtls.svc.cluster.local:8080    OK     STRICT     ISTIO_MUTUAL
...output omitted...
```

Services now use the **ISTIO_MUTUAL** mode, which means sending requests using mutual TLS.

5.4. Verify that the **customer** pod responds at the **/mtls** endpoint:

```
[student@workstation ~]$ curl $INGRESS_HOST
customer => preference => recommendation v1 from 'f11b097f1dd0': 2
```

6. Secure communication between pods so that each pod can only decrypt its own communication.

6.1. Create a service account for each pod:

```
[student@workstation ~]$ oc create serviceaccount customer
serviceaccount/customer created
[student@workstation ~]$ oc create serviceaccount recommendation
serviceaccount/recommendation created
[student@workstation ~]$ oc create serviceaccount preference
serviceaccount/preference created
```

6.2. Assign the service accounts to pod deployments:

```
[student@workstation ~]$ oc set serviceaccount deployment customer customer
deployment.apps/customer serviceaccount updated
[student@workstation ~]$ oc set serviceaccount deployment recommendation \
> recommendation
deployment.apps/recommendation serviceaccount updated
[student@workstation ~]$ oc set serviceaccount deployment preference preference
deployment.apps/preference serviceaccount updated
```

6.3. Assigning a service account to a deployment terminates the current pod and creates a new pod. Wait until all pods become ready.

```
[student@workstation ~]$ oc get pods
NAME                              READY   STATUS    RESTARTS   AGE
customer-5c7b87c6c5-55vrz         2/2     Running   0          22s
preference-6d98556fc5-vzk7v       2/2     Running   0          23s
recommendation-6b886bd8d8-8r6xq   2/2     Running   0          24s
```

6.4. Save the new pod name into a variable:

```
[student@workstation ~]$ CUSTOMER_POD=$(oc get pods -l app=customer -o \
> jsonpath={.items..metadata.name})
```

6.5. Verify the **customer** pod uses a unique SVID:

```
[student@workstation ~]$ oc exec $CUSTOMER_POD -c istio-proxy (1) \
> cat /etc/certs/cert-chain.pem | \ (2)
> openssl x509 -text -noout | \ (3)
> grep "X509v3 Subject" -A 1 (4)
  X509v3 Subject Alternative Name:
    URI:spiffe://cluster.local/ns/mtls/sa/customer
```

(1) Execute the following commands in the **istio-proxy** container, which contains the certificates.
(2) Print the encoded certificate for this pod.
(3) Decode the certificate using the **openssl** utility.
(4) Find the **Subject Alternative Name**, and then print the line below it.

Repeat the step to verify the SVID for the **preference** and **recommendation** pods.

6.6. Verify that pods can communicate using the unique identities:

```
[student@workstation ~]$ curl $INGRESS_HOST
customer => preference => recommendation v1 from 'f11b097f1dd0': 1
```

Finish

On the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab secure-mtls finish
```

This concludes the guided exercise.

Defining Service to Service Authorization

Objectives

After completing this section, you should be able to configure restriction on services communication in OpenShift Service Mesh.

Defining Authorization in Zero-Trust Perimeters

Previously, you learned about the concepts of zero-trust networks and using mutual TLS to encrypt your internal traffic. OpenShift Service Mesh also provides an authorization mechanism, which uses the strong service identity and provides developers fine-grained control over services communication.

In OpenShift Service Mesh, you can specify a deny-by-default communication pattern with targeted exceptions. For example, you expose the front end and gateway services to end users, but exposing the database directly would be a security risk. Red Hat OpenShift enables you to selectively expose only specified services. However, if an attacker breaches the initial security perimeter, internal services are left exposed and accessible to any other service.

An attacker might breach the initial network perimeter by injecting a malicious container into your cluster, for example due to a compromised container registry. In a worst case scenario, the malicious container is properly deployed into the cluster with its own Envoy proxy and a valid cryptographic identity.

OpenShift Service Mesh enables you to mitigate such a breach using the deny-by-default communication pattern. Developers can configure the OpenShift Service Mesh to allow connections to the database service only from pods with a trusted identity. Any other communication is forbidden and the malicious container does not gain full access to the database pod.

Authorization requires no modifications to the application code. Microservices focus on executing business logic while Pilot, together with Envoy proxies, ensure that authorization configuration is enforced.

Additionally, OpenShift Service Mesh enables developers to control egress calls. If the developers configure OpenShift Service Mesh with security in mind, the malicious container is unable to make any calls outside of OpenShift Service Mesh. Service to service authorization is another tool in the toolbox of a DevSecOps organization that enables you to minimize security vectors and mitigate security breaches.

Describing Authorization Workflow

Developers configure authorization using the **`AuthorizationPolicy`** custom resource definition (CRD). When you create a new **`AuthorizationPolicy`** CRD, Pilot generates an Envoy RBAC policy configuration, which propagates into Envoy proxy containers. Envoy proxies apply the authorization policies at run time, when another service sends a request.

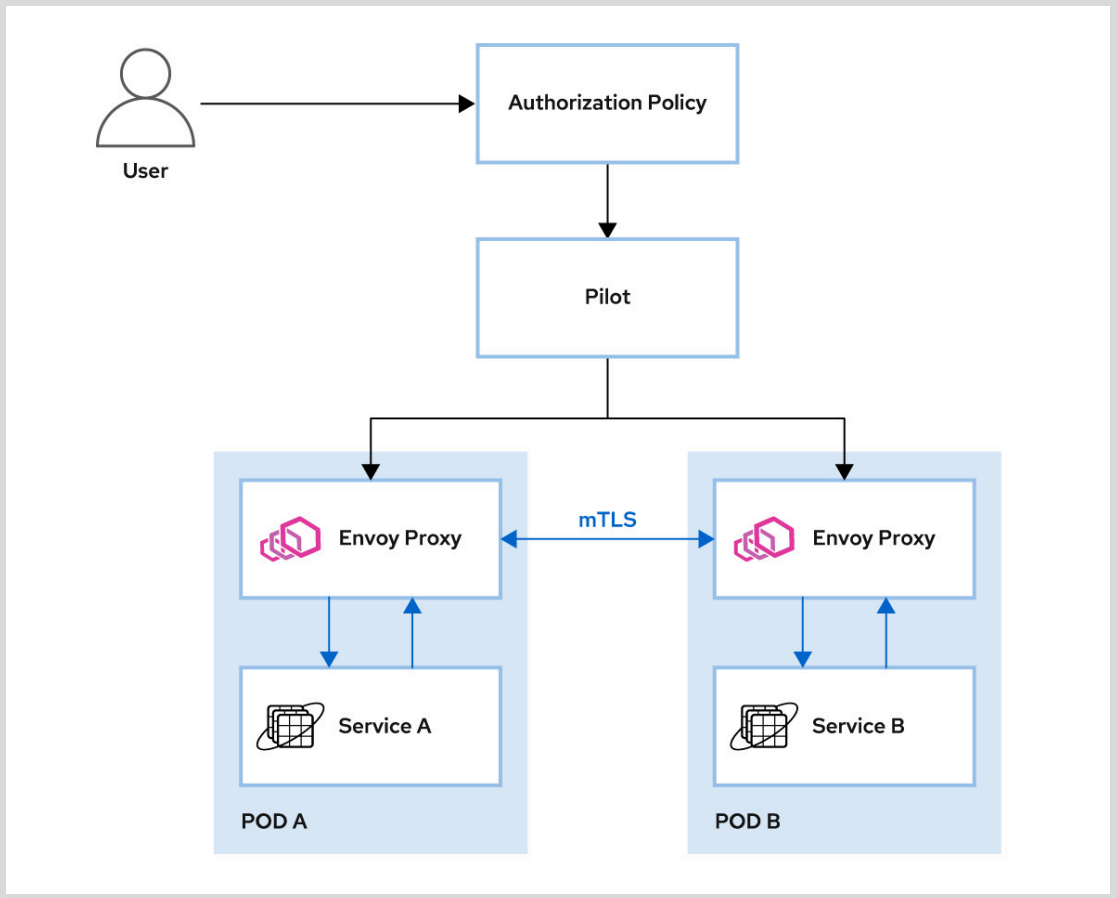


Figure 8.1: OpenShift Service Mesh Authorization Workflow

Note that mutual TLS is required for authorization policies that use identity, such as principals, namespaces, or server name indication (SNI). If you do not enable mutual TLS for identity-based conditions, then the request contains no identity and is denied by the policy.

Configuring Service Authorization

To configure OpenShift Service Mesh authorization policies, use the **`AuthorizationPolicy`** custom resource definition (CRD). The **`AuthorizationPolicy`** CRD specifies the following attributes:

- *Subject*, matched by the **`.spec.selector`** field. This is the target Envoy proxy that enforces the policy.
- *Action*, matched by the **`.spec.action`** field, specifies an allowlist (**`ALLOW`**) or a denylist (**`DENY`**) action. The default value is **`ALLOW`**.
- *Rule set*, matched by the **`spec.rules`** field, specifies the trigger for this policy, for example communication on a specified port. Each rule set has the following attributes:
  - **`source`** field, specifies the origin of the request, for example, a service identity.
  - **`to`** field, specifies the HTTP method, port, path, or other properties of the request.
  - **`when`** field, specifies additional conditions, such as the presence of an HTTP header.

For example:

```
apiVersion: "security.istio.io/v1beta1"
kind: "AuthorizationPolicy"
metadata:
  name: "test-policy"
spec:
  selector: (1)
    matchLabels:
      app: test
  action: ALLOW (2)
  rules: (3)
  - from:
    - source:
        namespaces: ["test"]
    to:
    - operation:
        methods: ["GET"]
        ports: ["8080"]
        paths: ["/"]
```

(1) This authorization policy targets all pods that match the **`app=test`** label. The Envoy proxy in matching pods will enforce this authorization policy.

(2) Action is explicitly set to **`ALLOW`**. Requests that trigger this policy are permitted. Any other requests to the matching pods will be denied, unless you specify another authorization policy.

(3) This policy matches requests from the **`test`** namespace with the **`GET`** HTTP method to the **`8080`** port matching the **`/`** endpoint.

Note the following:

- If you create no authorization policy, then all requests are permitted. However, when you specify at least one authorization policy, you enable deny-by-default behavior for the Envoy proxy of the matching workload.

  This behavior also applies when you create a single denylist policy. If you create a single authorization policy with the **`DENY`** action, all requests will be denied, and your pod will be inaccessible.

- You can create multiple authorization policies for one workload.

  Creating multiple policies for a single workload is useful for designing atomic, easy-to-maintain policies. Creating multiple policies for a single workload also enables other patterns, such as temporary permit requests used for testing or development.

  Each Envoy proxy contains a list of authorization policies. When an Envoy proxy receives a request, each policy is evaluated separately. The first policy that matches is applied. This results in a logical OR behavior.

Warning
Do not rely on the policy order of your envoy proxy.

- The scope of authorization policies is a project. The **`test-policy`** example policy above will not match workloads with the **`app=test`** label in a different project.

Combining Authorization Policy Rule Parameters

You can specify multiple rule parameters, such as the **`.spec.rules.from.source`** or **`.spec.rules.from.to.operation`** parameters. Specifying multiple matching parameters results in the logical OR behavior.

This is useful for specifying multiple trigger conditions for the policy. However, large authorization policies with multiple rule parameters are be difficult to understand and maintain. It is best practice to keep your authorization policies as small and atomic as possible in the context of your environment.

The following policy example combines multiple **`.spec.rules.from.source`** fields:

```
apiVersion: "security.istio.io/v1beta1"
kind: "AuthorizationPolicy"
metadata:
  name: "test-policy-multiple-sources"
spec:
  selector:
    matchLabels:
      app: test
  rules:
  - from:
    - source:
        namespaces: ["test"]
    - source:
        principals: ["cluster.local/ns/test2/sa/test2-service-account"]
    - source:
        ipBlocks: ["10.128.0.0/14"]
```

The example **`test-policy-multiple-sources`** authorization policy specifies no **`.spec.action`** field value. Consequently, the default **`ALLOW`** value is applied. The policy permits any requests that match at least one of the following conditions:

- Requests originate from the **`test`** namespace
- Requests identified by the **`test2-service-account`** service account in project **`test2`**
- Requests originating from an IP in the **`10.128.0.0/14`** range

The following policy example combines multiple **`.spec.rules.from.to.operation`** fields:

```
apiVersion: "security.istio.io/v1beta1"
kind: "AuthorizationPolicy"
metadata:
  name: "test-policy-multiple-operation"
spec:
  selector:
    matchLabels:
      app: test
  rules:
  - from:
    - to:
      - operation:
          methods: ["GET"]
          ports: ["8080"]
```

```
    - operation:
        methods: ["POST"]
        ports: ["8080"]
        paths: ["/api/v2/createEmployee/*"]
    - operation:
        methods: ["PUT"]
        ports: ["8080"]
        paths: ["/api/v2/updateEmployee"]
```

The **`test-policy-multiple-operation`** authorization policy permits any requests that match at least one of the following conditions:

- Any **`GET`** requests on the **`8080`** port.
- **`POST`** requests on the **`8080`** port to the **`/api/v2/createEmployee/*`** endpoints.
- **`PUT`** requests on the **`8080`** port to the **`/api/v2/updateEmployee`** endpoint.

References

Authorization Policy (Reference)
https://archive.istio.io/v1.4/docs/reference/config/security/authorization-policy/

Authorization Policy Conditions
https://archive.istio.io/v1.4/docs/reference/config/security/conditions/

Authorization Policy (Concept)
https://archive.istio.io/v1.4/docs/concepts/security/#authorization-policy

Envoy Proxy HTTP filters
https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/http_filters

Guided Exercise

Configuring Service to Service Authorization

In this exercise, you will restrict service-to-service communication within a project and across projects.

At the start of the exercise, the **`lab`** command deploys the **`customer`**, **`preference`**, and **`recommendation`** services into your Red Hat OpenShift cluster. The **`customer`** service sends requests to the **`preference`** service. The **`preference`** service sends requests the **`recommendation`** service.

You will:

- Restrict each service to accept incoming requests from one and only one other service, as per the service workflow.
- Relax the restriction by enabling the **`customer`** service to accept incoming requests from an external project, to simulate developer environment.

Outcomes

You should be able to configure OpenShift Service Mesh to restrict access to services based on service parameters, such as namespace or service account.

Before You Begin

To perform this exercise, ensure you have:

- A configured and running OpenShift cluster.
- A installed and running OpenShift Service Mesh in the OpenShift cluster.
- The OpenShift CLI (**`/usr/local/bin/oc`**).

On the **`workstation`** machine, use the **`lab`** command to prepare your system for this exercise.

The **`lab`** command deploys the **`customer`**, **`preference`**, and **`recommendation`** services into your Red Hat OpenShift cluster. The source code is in the Git repository at https://github.com/RedHatTraining/DO328-apps in the **`customer`**, **`preference`**, and **`recommendation`** directories.

Each service deployment uses a unique service account. Mutual TLS is enabled and enforced within the **`secure-authc`** project. You can examine the full deployment file in the **`~/DO328/labs/secure-authc/app-deployment.yaml`** file. In the **`app-deployment.yaml`** file, note that each **`Deployment`** resource specifies a unique service account. Additionally, examine the **`Policy`** and **`DestinationRule`** resources.

```
[student@workstation ~]$ lab secure-authc start
```

1. Log in to OpenShift and verify that the application is ready.

   1.1. Run the following command to load the environment variables created in the first guided exercise:

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

   1.2. Log in to OpenShift:

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_API}
Login successful.
...output omitted...
```

   1.3. Change to project **`secure-authc`**:

```
[student@workstation ~]$ oc project secure-authc
Now using project "secure-authc" on server ...
```

   1.4. Verify that pods are in the **`Running`** state:

```
[student@workstation ~]$ oc get pods
NAME                              READY   STATUS    RESTARTS   AGE
customer-5f9bb4676-2g4f6          2/2     Running   0          11s
preference-748787ff89-fnxt8       2/2     Running   0          11s
recommendation-78588b5ff9-wx52s   2/2     Running   0          11s
```

   1.5. Save the **`ingress-gateway`** route host name with the **`/secure-authc`** endpoint into a variable:

```
[student@workstation ~]$ INGRESS_HOST=$(oc get route -n istio-system \
> istio-ingressgateway -o jsonpath='{.spec.host}'/secure-authc)
```

   1.6. Verify that the service responds using the **`INGRESS_HOST`** variable:

```
[student@workstation ~]$ curl $INGRESS_HOST
customer => preference => recommendation v1 from 'f11b097f1dd0': 1
```

2. Verify that there is no restriction on pod communication. Any pod in the mesh can communicate with any other pod in the **`secure-authc`** project.

   2.1. Create a new project named **`curl`**:

```
[student@workstation ~]$ oc new-project curl
Now using project "curl" on server ...
...output omitted...
```

   2.2. Add the **`curl`** project into the **`servicemeshmemberroll`** resource:

```
[student@workstation ~]$ oc patch smmr default -n istio-system \
> --type='json' -p='[{"op": "add", "path": "/spec/members/0", "value":"curl"}]'
servicemeshmemberroll.maistra.io/default patched
```

Alternatively, use the **~/DO328/labs/secure-authc/patch-smmr.sh** script.

2.3. Deploy the **sleep** application into the **curl** project:

```
[student@workstation ~]$ oc create -f ~/DO328/labs/secure-authc/sleep.yml
service/sleep created
deployment.apps/sleep created
```

The **sleep** pod is a container with the **sleep** command as its execution command. You can view the source code in the Git repository at https://github.com/RedHatTraining/DO328-apps in the **sleep** directory.

2.4. After the **sleep** pod changes into the **Ready** state, issue a request from the **sleep** pod to the **customer** service:

```
[student@workstation ~]$ oc exec $(oc get pods -o name) -- \
> curl -s customer.secure-authc.svc.cluster.local:8080
Defaulting container name to sleep.
Use 'oc describe pod/sleep-556b5fc57c-725w7 -n curl' to see all of the containers
 in this pod.
customer => preference => recommendation v1 from 'f11b097f1dd0': 2
```

The **sleep** pod can reach the **customer** service in a different namespace.

2.5. Change to the **secure-authc** project:

```
[student@workstation ~]$ oc project secure-authc
Now using project "secure-authc" on server ...
...output omitted...
```

2.6. Issue a request from the **recommendation** pod to the **customer** pod:

```
[student@workstation ~]$ oc exec $(oc get pod -l app=recommendation -o name) -- \
> curl -s customer:8080
Defaulting container name to recommendation.
Use 'oc describe pod/recommendation-78588b5ff9-wx52s -n secure-authc' to see all
 of the containers in this pod.
customer => preference => recommendation v1 from 'f11b097f1dd0': 3
```

3. Restrict access to the **customer** pod using the following information:

- The **customer** pod is accessible only by the **istio-ingressgateway-service-account** service account.
- The **customer** pod is accessible only at the port **8080**.
- The **customer** pod is accessible only using the **GET** method.

3.1. Prepare the **AuthorizationPolicy** resource. You can view the prepared policy **customer-policy.yaml** in the **~/DO328/solutions/secure-authc** directory.

```
[student@workstation ~]$ cat > customer-policy.yaml << EOF
> apiVersion: "security.istio.io/v1beta1"
> kind: "AuthorizationPolicy"
> metadata:
>   name: "get-customer"
> spec:
>   selector:
>     matchLabels:
>       app: customer
>   rules:
>   - from:
>     - source:
>         principals: ["cluster.local/ns/istio-system/sa/istio-ingressgateway-
service-account"]
>     to:
>     - operation:
>         methods: ["GET"]
>         ports: ["8080"]
> EOF
```

1. This policy targets all pods with the label **app=customer**.
2. Requests from the **istio-ingressgateway-service-account** identity are permitted.
3. Requests from the permitted source are restricted to method **GET** on port **8080**.

3.2. Create the policy:

```
[student@workstation ~]$ oc create -f customer-policy.yaml
authorizationpolicy.security.istio.io/get-customer created
```

3.3. Issue a request from the **recommendation** pod to the **customer** pod to test the policy enforcement:

```
[student@workstation ~]$ oc exec $(oc get pod -l app=recommendation -o name) -- \
> curl -s customer:8080
Defaulting container name to recommendation.
Use 'oc describe pod/recommendation-78588b5ff9-wx52s -n secure-authc' to see all
 of the containers in this pod.
RBAC: access denied
```

Note

It takes a couple seconds before the authorization policy settings propagate into the Envoy proxy of the **customer** pod.

If the request succeeds, try to issue the same request again after 5-10 seconds.

4. Repeat the previous step to restrict communication for the **preference** and **recommendation** pods.

4.1. Restrict communication for the **preference** pod using the following information:

- The **preference** pod is accessible only by the **customer** pod with identity **customer-sa**.
- The **preference** pod is accessible only at the **8080** port.
- The **preference** pod is accessible only using the **GET** method.

The solution file is provided at **~/DO328/solutions/secure-authc/preference-policy.yaml**.

4.2. Restrict communication for the **recommendation** pod using the following information:

- The **recommendation** pod is accessible only by the **preference** pod with identity **preference-sa**.
- The **recommendation** pod is accessible only at the **8080** port.
- The **recommendation** pod is accessible only using the **GET** method.

The solution file is provided at **~/DO328/solutions/secure-authc/recommendation-policy.yaml**.

5. For development and testing, make the **customer** pod accessible from any other pod in the **curl** project.

5.1. Change into the **curl** project:

```
[student@workstation ~]$ oc project curl
Now using project "curl" on server ...
...output omitted
```

5.2. Verify that pods in the **curl** project cannot communicate with the **customer** pod.

```
[student@workstation ~]$ oc exec $(oc get pods -o name) -- \
> curl -s customer.secure-authc.svc.cluster.local:8080
Defaulting container name to sleep.
Use 'oc describe pod/sleep-556b5fc57c-jvm9b -n curl' to see all of the containers
 in this pod.
RBAC: access denied
```

5.3. Prepare the **AuthorizationPolicy** resource:

```
[student@workstation ~]$ cat > curl-customer-policy.yaml << EOF
> apiVersion: "security.istio.io/v1beta1"
> kind: "AuthorizationPolicy"
> metadata:
>   name: "curl-get-customer"
>   namespace: "secure-authc"
> spec:
>   selector:
>     matchLabels:
>       app: customer
>   rules:
```

```
>   - from:
>     - source:
>         namespaces: ["curl"]
```

The **curl-customer-policy.yaml** file is available in the **~/DO328/solutions/secure-authc** directory.

5.4. Create the policy:

```
[student@workstation ~]$ oc create -f curl-customer-policy.yaml
authorizationpolicy.security.istio.io/curl-get-customer created
```

5.5. Verify that pods in the **curl** project can now communicate with the **customer** pod:

```
[student@workstation ~]$ oc exec $(oc get pods -o name) -- \
> curl -s customer.secure-authc.svc.cluster.local:8080
Defaulting container name to sleep.
Use 'oc describe pod/sleep-556b5fc57c-jvm9b -n curl' to see all of the containers
 in this pod.
customer => preference => recommendation v1 from 'f11b097f1dd0': 4
```

5.6. Verify that pods in the **curl** project cannot communicate with other pods in the **secure-authc** project:

```
[student@workstation ~]$ oc exec $(oc get pods -o name) -- \
> curl -s preference.secure-authc.svc.cluster.local:8080
Defaulting container name to sleep.
Use 'oc describe pod/sleep-556b5fc57c-jvm9b -n curl' to see all of the containers
 in this pod.
RBAC: access denied
```

Finish

On the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab secure-authc finish
```

This concludes the guided exercise.

Lab

Securing an OpenShift Service Mesh

Performance Checklist

In this lab, you will secure the provided application using the OpenShift Service Mesh authorization policies.

Outcomes

You should be able to:

- Enable and verify mutual TLS.
- Configure authorization policies based on service identity.
- Restrict service communication using authorization policies based on HTTP attributes, such as HTTP verbs, ports, and endpoints.

Before You Begin

To perform this exercise, ensure you have:

- A configured and running OpenShift cluster.
- A installed and running OpenShift Service Mesh in the OpenShift cluster.

As the **`student`** user on the **`workstation`** machine, use the **`lab`** command to prepare your system for this lab.

The **`lab`** command deploys the exchange application into your Red Hat OpenShift cluster. The source code is in the Git repository at https://github.com/RedHatTraining/DO328-apps in the **`exchange-application`** directory.

The exchange application consists of the following services:

- Frontend
- Currency
- Exchange
- History

You can examine the services in the **`secure-mesh`** project. The application is available using the **`istio-ingressgateway`** route at the **`/frontend`** endpoint.

Additionally, the **`lab`** command also deploys the dashboard application into your Red Hat OpenShift cluster. The dashboard application provides information about availability of the exchange application services in the service mesh.

The dashboard source code is in the Git repository at https://github.com/RedHatTraining/DO328-apps in the **`dashboard`** directory.

The dashboard application consists of the following services:

- Frontend

- Backend

You can examine the services in the **dashboard** project. The application is available using the **istio-ingressgateway** route at the **/dashboard** endpoint.

```
[student@workstation ~]$ lab secure-mesh start
```

Secure the exchange application according to the following specifications:

- The **frontend** service is accessible only from the **istio-ingressgateway** service, using only **GET** requests on port **3000**.
- The **exchange** service is accessible only from the **istio-ingressgateway** service on port **8080** on the following endpoints:
  - **POST /currencies**
  - **POST /exchangeRate/singleCurrency**
  - **POST /exchangeRate/historicalData**
- The **currency** and **history** services are accessible only from the **secure-mesh** project.

After you secure the exchange application, it should be accessible on the **/frontend** endpoint of the **istio-ingressgateway** route. However, it should not be accessible to any other services in the service mesh. Consequently, the dashboard application should mark all services with status **Down**, and unavailable.

1. Log in to OpenShift and verify that the application is ready.
2. Configure prerequisites for identity-based service-to-service restriction.
3. Restrict access to the **frontend** service.
4. Restrict access to the **exchange** service.
5. Restrict access to the **currency** and **history** services.

Evaluation

Grade your work by running the **lab secure-mesh grade** command from your **workstation** machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab secure-mesh grade
```

Finish

On the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab secure-mesh finish
```

This concludes the lab.

Solution

Securing an OpenShift Service Mesh

Performance Checklist

In this lab, you will secure the provided application using the OpenShift Service Mesh authorization policies.

Outcomes

You should be able to:

- Enable and verify mutual TLS.
- Configure authorization policies based on service identity.
- Restrict service communication using authorization policies based on HTTP attributes, such as HTTP verbs, ports, and endpoints.

Before You Begin

To perform this exercise, ensure you have:

- A configured and running OpenShift cluster.
- A installed and running OpenShift Service Mesh in the OpenShift cluster.

As the **`student`** user on the **`workstation`** machine, use the **`lab`** command to prepare your system for this lab.

The **`lab`** command deploys the exchange application into your Red Hat OpenShift cluster. The source code is in the Git repository at https://github.com/RedHatTraining/DO328-apps in the **`exchange-application`** directory.

The exchange application consists of the following services:

- Frontend
- Currency
- Exchange
- History

You can examine the services in the **`secure-mesh`** project. The application is available using the **`istio-ingressgateway`** route at the **`/frontend`** endpoint.

Additionally, the **`lab`** command also deploys the dashboard application into your Red Hat OpenShift cluster. The dashboard application provides information about availability of the exchange application services in the service mesh.

The dashboard source code is in the Git repository at https://github.com/RedHatTraining/DO328-apps in the **`dashboard`** directory.

The dashboard application consists of the following services:

- Frontend
- Backend

You can examine the services in the **dashboard** project. The application is available using the **istio-ingressgateway** route at the **/dashboard** endpoint.

```
[student@workstation ~]$ lab secure-mesh start
```

Secure the exchange application according to the following specifications:

- The **frontend** service is accessible only from the **istio-ingressgateway** service, using only **GET** requests on port **3000**.
- The **exchange** service is accessible only from the **istio-ingressgateway** service on port **8080** on the following endpoints:
  - **POST /currencies**
  - **POST /exchangeRate/singleCurrency**
  - **POST /exchangeRate/historicalData**
- The **currency** and **history** services are accessible only from the **secure-mesh** project.

After you secure the exchange application, it should be accessible on the **/frontend** endpoint of the **istio-ingressgateway** route. However, it should not be accessible to any other services in the service mesh. Consequently, the dashboard application should mark all services with status **Down**, and unavailable.

1. Log in to OpenShift and verify that the application is ready.

   1.1. Run the following command to load the environment variables created in the first guided exercise:

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

   1.2. Log in to OpenShift:

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
> -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_API}
Login successful.
...output omitted...
```

   1.3. Change to project **secure-mesh**:

```
[student@workstation ~]$ oc project secure-mesh
Now using project "secure-mesh" on server ...
```

   1.4. Verify that pods are in the **Running** state:

```
[student@workstation ~]$ oc get pods
NAME                        READY   STATUS    RESTARTS   AGE
currency-566cddc8c6-jtd5g   2/2     Running   0          13s
exchange-66b78bf65c-s72gv   2/2     Running   0          13s
frontend-5648fbb85f-td5dg   2/2     Running   0          13s
history-54b5c9d476-rk4pd    2/2     Running   0          13s
```

1.5. Save the **`ingress-gateway`** route host name with the **`/frontend`** endpoint into a variable:

```
[student@workstation ~]$ FRONTEND=$(oc get route -n istio-system \
> istio-ingressgateway -o jsonpath='{.spec.host}'/frontend)
```

1.6. Verify that the application responds using the **FRONTEND** variable:

```
[student@workstation ~]$ firefox $FRONTEND
```

1.7. Save the **`ingress-gateway`** route hostname with the **`/dashboard`** endpoint into a variable:

```
[student@workstation ~]$ DASHBOARD=$(oc get route -n istio-system \
> istio-ingressgateway -o jsonpath='{.spec.host}'/dashboard)
```

1.8. Verify that all the services are accessible using the **DASHBOARD** variable:

```
[student@workstation ~]$ firefox $DASHBOARD
```

2. Configure prerequisites for identity-based service-to-service restriction.

2.1. Enable mutual TLS:

```
[student@workstation ~]$ oc patch smcp basic-install -n istio-system  \
> --type merge -p '{"spec":{"istio":{"global":{"mtls":{"enabled": true}}}}}'
servicemeshcontrolplane.maistra.io/basic-install patched
```

3. Restrict access to the **`frontend`** service.

3.1. Prepare the authorization policy. You can view the complete **`get-frontend.yaml`** file in the **`~/DO328/solutions/secure-mesh`** directory.

```
[student@workstation ~]$ cat > get-frontend.yaml << EOF
> apiVersion: "security.istio.io/v1beta1"
> kind: "AuthorizationPolicy"
> metadata:
>   name: "get-frontend"
> spec:
>   selector:
>     matchLabels:
>       app: frontend
```

```
>   rules:
>   - from:
>     - source:
>         principals: ["cluster.local/ns/istio-system/sa/istio-ingressgateway-
service-account"]
>     - to:
>       - operation:
>           methods: ["GET"]
>           ports: ["3000"]
> EOF
```

3.2. Apply the authorization policy:

```
[student@workstation ~]$ oc create -f get-frontend.yaml
authorizationpolicy.security.istio.io/get-frontend created
```

3.3. Verify that the application responds using the **FRONTEND** variable:

```
[student@workstation ~]$ firefox $FRONTEND
```

3.4. Verify that the **Frontend Service** service is unavailable in the dashboard application:

```
[student@workstation ~]$ firefox $DASHBOARD
```

4. Restrict access to the **exchange** service.

4.1. Prepare the authorization policy. You can view the complete **exchange.yaml** file in the **~/DO328/solutions/secure-mesh** directory.

```
[student@workstation ~]$ cat > exchange.yaml << EOF
> apiVersion: "security.istio.io/v1beta1"
> kind: "AuthorizationPolicy"
> metadata:
>   name: "exchange"
> spec:
>   selector:
>     matchLabels:
>       app: exchange
>   rules:
>   - from:
>     - source:
>         principals: ["cluster.local/ns/istio-system/sa/istio-ingressgateway-
service-account"]
>     - to:
>       - operation:
>           methods: ["POST"]
>           ports: ["8080"]
>           paths: ["/currencies", "/exchangeRate/singleCurrency", "/exchangeRate/
historicalData"]
> EOF
```

4.2. Apply the authorization policy:

```
[student@workstation ~]$ oc create -f exchange.yaml
authorizationpolicy.security.istio.io/exchange created
```

4.3. Verify that the **Historical Data** page of the application works. Open the page:

```
[student@workstation ~]$ firefox $FRONTEND/history
```

On the **Historical Data** page, click **Submit**.

4.4. Verify that the **Exchange** page of the application works. Open the page:

```
[student@workstation ~]$ firefox $FRONTEND/exchange
```

On the **Exchange** page, click **Submit**.

4.5. Verify that the **Gateway Service** service is unavailable in the dashboard application:

```
[student@workstation ~]$ firefox $DASHBOARD
```

5. Restrict access to the **currency** and **history** services.

5.1. Add a new label, **ns-restricted**, to each of the service deployments. Alternatively, you can use the **add-labels.sh** script in the **~/DO328/solutions/secure-mesh** directory.

```
[student@workstation ~]$ oc patch deployment history --type='json' \
>  -p='[{"op": "add", "path": "/spec/template/metadata/labels/ns-restricted",
 "value":"true"}]'
deployment.apps/history patched
[student@workstation ~]$ oc patch deployment currency --type='json' \
>  -p='[{"op": "add", "path": "/spec/template/metadata/labels/ns-restricted",
 "value":"true"}]'
deployment.apps/currency patched
```

5.2. Verify that both **currency** and **history** pods are labeled using the **ns-restricted** label, and are in the **Running** state:

```
[student@workstation ~]$ oc get pods -l ns-restricted
NAME                        READY   STATUS    RESTARTS   AGE
currency-65b7dbdc75-2smhv   2/2     Running   0          28s
history-64bf7cf7d5-mnmr6    2/2     Running   0          44s
```

5.3. Prepare the authorization policy. You can view the complete **ns-restricted.yaml** file in the **~/DO328/solutions/secure-mesh** directory.

```
[student@workstation ~]$ cat > ns-restricted.yaml << EOF
> apiVersion: "security.istio.io/v1beta1"
> kind: "AuthorizationPolicy"
> metadata:
>   name: "ns-restricted"
> spec:
```

```
>   selector:
>     matchLabels:
>       "ns-restricted": "true"
>   rules:
>   - from:
>     - source:
>         namespaces: ["secure-mesh"]
> EOF
```

5.4. Apply the authorization policy:

```
[student@workstation ~]$ oc create -f ns-restricted.yaml
authorizationpolicy.security.istio.io/ns-restricted created
```

5.5. Verify that the **Historical Data** page of the application works. Open the page:

```
[student@workstation ~]$ firefox $FRONTEND/history
```

On the **Historical Data** page, click **Submit**.

5.6. Verify that the **Exchange** page of the application works. Open the page:

```
[student@workstation ~]$ firefox $FRONTEND/exchange
```

On the **Exchange** page, click **Submit**.

5.7. Verify that the **History Service** and **Currency Service** services are unavailable in the dashboard application:

```
[student@workstation ~]$ firefox $DASHBOARD
```

Evaluation

Grade your work by running the **lab secure-mesh grade** command from your **workstation** machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab secure-mesh grade
```

Finish

On the **workstation** machine, use the **lab** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab secure-mesh finish
```

This concludes the lab.

Summary

In this chapter, you learned:

- How OpenShift Service Mesh enables you to adopt DevSecOps practices.
- Configuring and troubleshooting mutual TLS (mTLS).
- Securing microservices by restricting service-to-service communication using the deny-by-default pattern.

Chapter 9

Comprehensive Review

Goal	Review tasks from *Building Resilient Microservices with Istio and Red Hat OpenShift Service Mesh*
Objectives	• Review tasks from *Building Resilient Microservices with Istio and Red Hat OpenShift Service Mesh*
Sections	• Comprehensive Review of Red Hat OpenShift Service Mesh (and Lab)
Lab	Building Resilient Microservices

Comprehensive Review

Objectives

After completing this section, you should be able to demonstrate knowledge and skills learned in *Building Resilient Microservices with Istio and Red Hat OpenShift Service Mesh* .

Reviewing Building Resilient Microservices with Istio and Red Hat OpenShift Service Mesh

Before beginning the comprehensive review for this course, you should be comfortable with the topics covered in each chapter.

You can refer to earlier sections in the textbook for extra study.

Chapter 1, Introducing Red Hat OpenShift Service Mesh

Describe the basic concepts of microservice architecture and Red Hat OpenShift Service Mesh.

- Describe the basic concepts behind a distributed architecture and Red Hat OpenShift Service Mesh.
- Describe the fundamental architecture of OpenShift Service Mesh components.

Chapter 2, Installing Red Hat OpenShift Service Mesh

Deploy Red Hat OpenShift Service Mesh on OpenShift Container Platform.

Install Red Hat OpenShift Service Mesh on Red Hat OpenShift Container Platform.

Chapter 3, Observing a Service Mesh

Trace and visualize an OpenShift Service Mesh with Jaeger and Kiali.

- Configure distributed tracing to track service traffic.
- Collect and inspect critical metrics with Prometheus and Grafana.
- Monitor and visualize service interactions with Kiali.

Chapter 4, Controlling Service Traffic

Manage and route traffic with Red Hat OpenShift Service Mesh

- Manage and route traffic with Red Hat OpenShift Service Mesh.
- Route traffic to services in a mesh, based on request headers.
- Control egress traffic to access external services.

Chapter 5, Releasing Applications with OpenShift Service Mesh

Release applications with canary and mirroring release strategies.

- Release application services with a safe canary rollout.
- Deploy a "mirror" service to test a new service with a realistic load.

Chapter 6, Testing Service Resilience with Chaos Testing

Test the resiliency of an OpenShift Service Mesh with Chaos Testing.

- Create test errors to identify weaknesses in your application.
- Create a delay in your services to test for weaknesses in your application.

Chapter 7, Building Resilient Services

Leverage OpenShift Service Mesh strategies for creating resilient services.

- Describe the strategies for creating resilient services with Service Mesh.
- Configure time-outs to maintain service reliability.
- Configure a service retry to maintain service reliability.
- Configure a circuit breaker pattern to maintain service reliability.

Chapter 8, Securing an OpenShift Service Mesh

Secure and encrypt services in your application with Red Hat OpenShift Service Mesh.

- Describe how Citadel manages identities.
- Configure Mutual TLS to secure intra-service communication.
- Configure restriction on services communication in OpenShift Service Mesh.

Lab

Building Resilient Microservices

Performance Checklist

In this review, you will deploy the "Adopt a Pup" application to OpenShift and configure Red Hat OpenShift Service Mesh to manage the traffic and security aspects of the application. You will also configure the service mesh to be more resilient against delays, timeouts, and failures.

Outcomes

You should be able to:

- Deploy the "Adopt a Pup" application to OpenShift and enable Red Hat OpenShift Service Mesh to control the incoming and outgoing traffic.
- Secure the application by configuring access control and allowing only encrypted communication between microservices.
- Configure the service mesh to perform canary releases and dark launches for testing new features of the application.
- Configure the service mesh to make the application more resilient using timeouts, retries, and circuit breakers.

Before You Begin

To perform this exercise, ensure you have access to:

- A configured and running OpenShift cluster.
- An installed and running OpenShift Service Mesh in the OpenShift cluster.
- The OpenShift CLI (**`/usr/local/bin/oc`**).
- The Istio CLI (**`istioctl`**).

As the **`student`** user on the **`workstation`** machine, use the **`lab`** command to validate the prerequisites for this exercise, and prepare the lab:

```
[student@workstation ~]$ lab comprehensive-review start
```

Instructions

The start script creates a new project named **`adoptapup`** and deploys all services except the **`adoption`** service. The **`news`** service deploys in a separate project named **`adoptapup-news`**.

The architecture of the "Adopt a Pup" application is as follows:

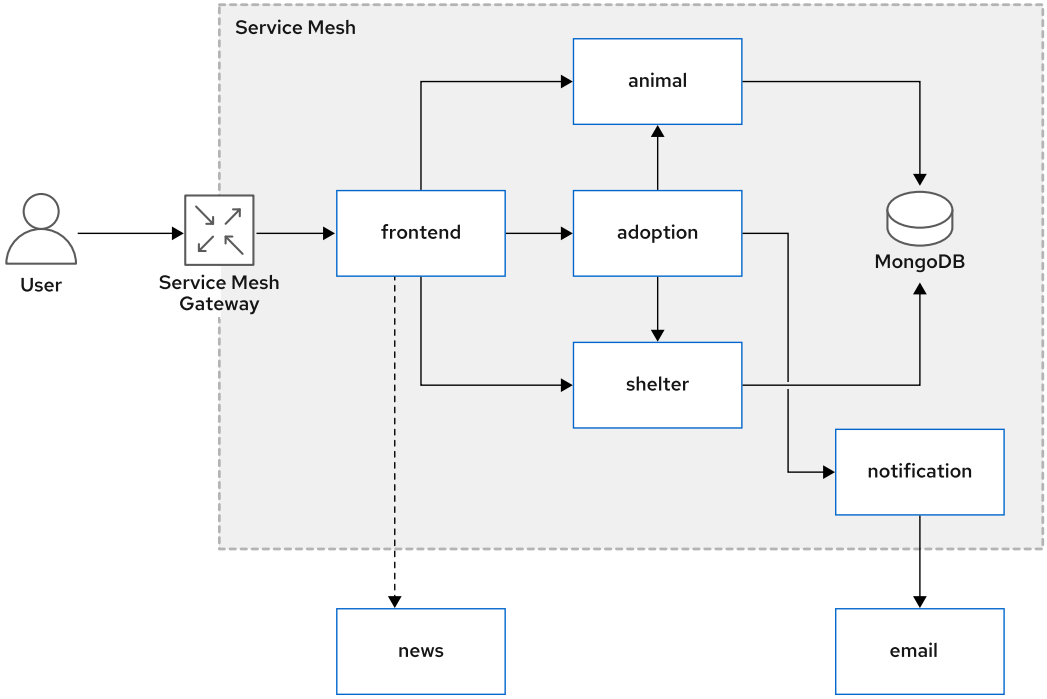


Figure 9.1: Adopt a Pup

The application consists of the following microservices:

animal
Manages a set of pups that can be adopted. Each pup belongs to a particular animal shelter.

shelter
Manages several animal shelters that take care of pups until they are adopted. A shelter can have one or more pups.

notification
Manages notifications that are sent to potential owners of pups. The notification service handles multiple notifications, such as emails and text messages. Currently, only notifications by email are supported.

email
Sends emails to users. The service mesh does not manage this service.

adoption
This service processes user requests for adoption. It is responsible for data validation and verifying that the user is eligible to adopt a pup.

news
This service periodically informs users about news regarding shelters and animals. This service is deployed in a separate project. The service mesh does not manage this service.

frontend
This service provides the web user interface for the application.

mongodb
The details for pups, shelters, and adoptions are stored in this MongoDB database.

The start script deployed these services using YAML files located in the **/home/student/DO328/labs/comprehensive-review** folder.

The start script deployed the MongoDB database and preloaded shelter and animal data into the database.

Note

Name all virtual service resources created in the **adoptapup** project as *service*-vs, where *service* is the service name, for example, **adoption-vs** and **animal-vs**.

Name all destination rule resources created in the **adoptapup** project as *service*-dr, where *service* is the service name, for example **adoption-dr** and **animal-dr**.

Name all service accounts created in the **adoptapup** project as *service*-sa, where *service* is the service name, for example **adoption-sa** and **animal-sa**.

Perform the following tasks:

1. Log in to OpenShift as the **developer** user and inspect the deployed microservices in the **adoptapup** and **adoptapup-news** projects. Verify that all the microservices, except the **adoption** microservice, are deployed and running.

   Remember to set the environment variables in the **/usr/local/etc/ocp4.config** file.

2. Verify that the data has been successfully loaded into MongoDB using the **check-mongo.sh** script in the **/home/student/DO328/labs/comprehensive-review** folder. If data is successfully loaded, you will see the raw JSON data from the database as follows:

```
[student@workstation ~]$ sh ~/DO328/labs/comprehensive-review/check-mongo.sh
Login successful.

You have access to the following projects ...output omitted...

  * adoptapup
...output omitted...
Checking if animal data is loaded into MongoDB...

{ "_id" : "d52a8d58-9024-49dd-92b6-d443c6049ffe", "animalName" : "Gus" ...output
 omitted...
...output omitted...

Checking if shelter data is loaded into MongoDB...

{ "_id" : "e038ae3c-592f-403e-9233-4b6eeab30e3c", "shelterName" : "Denver
 Doggos" ...output omitted...
...output omitted...
```

   If the data is not loaded, then run the **load-mongo.sh** script in the **/home/student/DO328/labs/comprehensive-review** folder, and then rerun the **check-mongo.sh** script to verify success.

3. Create a service mesh gateway named **adoptapup-gateway** to allow external traffic to flow into the service mesh. Configure the gateway to listen for HTTP requests on port **80**.

4. Deploy the **adoption** service in the **adoptapup** project. Ensure that it is managed by Red Hat OpenShift Service Mesh.

The container image for the adoption service is available at **quay.io/redhattraining/ossm-adoption-service:1.0**. The container serves requests on port **8080**.

Name the OpenShift deployment and service resource as **adoption**, and then add a label **app: adoption** to the relevant resources. This service listens on port **8080**.

Name the virtual service as **adoption-vs**. Configure the virtual service so that all requests to the **/adoption** endpoint, relative to the service mesh gateway URL, are routed to the **adoption** service.

Using a web browser, access the front end user interface at the **/frontend** endpoint relative to the gateway URL. Verify that you can browse shelters and animals from the navigation pane.

Warning

If you have restarted your classroom VMs, or redeployed the MongoDB database pod after running the start script, then you must run the **load-mongo.sh** script as discussed in a previous step. You must do this because the MongoDB pod uses ephemeral storage and it does not persist data between restarts.

5. Click **News** in the navigation panel. The front end fails to fetch data from the **news** service (external to the service mesh).

   Configure the service mesh to allow the front end service to fetch the latest news from the **news** service.

6. Version 2 of the **frontend** service introduces some user interface changes. The container image for version 2 is available at **quay.io/redhattraining/ossm-adopt-a-pup-webapp:2.0**.

   Deploy version 2 of the **frontend** service as follows:

   - Configure service mesh to route 80% of traffic to version 1 and the remainder to version 2.
   - The container name and deployment name should be called **frontend-v2**. Add the label **version: v2** to the relevant resources.

   Verify that the navigation panel background color is red in version 2, but not in version 1.

Note

You might have to refresh your browser several times before you can see the user interface changes. It can take some time for the page from version 2 to render as it loads the static assets (CSS and JavaScript files) for the first time.

7. Version 2 of the **animal** service focuses on improving performance. The container image for version 2 is available at **quay.io/RedHatTraining/ossm-animal-service:2.0**.

   Deploy version 2 of the **animal** service as follows:

   - The container name and deployment name should be called **animal-v2**. Add the label **version: v2** to the relevant resources.
   - Configure service mesh to mirror traffic from version 1 to version 2. Version 2 is not yet ready for production deployment, so responses to clients must still be sent exclusively from version 1.
   - Verify that traffic is mirrored and that you see the output from version 2 in the generated logs.

8. Introduce a 3-second delay for all responses from the **`shelter`** service. Verify that you see delayed responses when you click **`Our Shelters`** in the navigation panel.

9. Configure network connections for the **`adoption`** service to retry failed requests consistent with the following policies:

   - Configure the service mesh to allow **3** retry attempts. Each retry waits at most **`5 seconds`** before timing out.
   - Retries must be triggered only when the response contains an HTTP status code of 500 and above.

10. Configure a circuit breaker for the **`adoption`** service as follows:

    - Allow 3 consecutive errors within 10 seconds before breaking the circuit.
    - Allow 1 minute of recovery after the circuit breaks before activating the service to receive requests.

11. Configure connection pooling for the **`notification`** service as follows:

    - Allow only 5 concurrent connections at any given time to prevent overloading this service.
    - Allow only 1 request per connection.

12. Secure the services in the service mesh as follows:

    - Ensure that traffic sent from the **`animal`** and **`shelter`** services to the MongoDB database is encrypted. Enable mutual authentication between these services and the MongoDB database.
    - Configure the service mesh so that only the **`shelter`** service and version 1 of the **`animal`** service are authorized to communicate with the MongoDB database.

    Name your resource **`mongodb-auth-policy`**.

Evaluation

Grade your work by running the **`lab comprehensive-review grade`** command from your **`workstation`** machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab comprehensive-review grade
```

Finish

On the **`workstation`** machine, use the **`lab`** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab comprehensive-review finish
```

This concludes the lab.

Solution

Building Resilient Microservices

Performance Checklist

In this review, you will deploy the "Adopt a Pup" application to OpenShift and configure Red Hat OpenShift Service Mesh to manage the traffic and security aspects of the application. You will also configure the service mesh to be more resilient against delays, timeouts, and failures.

Outcomes

You should be able to:

- Deploy the "Adopt a Pup" application to OpenShift and enable Red Hat OpenShift Service Mesh to control the incoming and outgoing traffic.
- Secure the application by configuring access control and allowing only encrypted communication between microservices.
- Configure the service mesh to perform canary releases and dark launches for testing new features of the application.
- Configure the service mesh to make the application more resilient using timeouts, retries, and circuit breakers.

Before You Begin

To perform this exercise, ensure you have access to:

- A configured and running OpenShift cluster.
- An installed and running OpenShift Service Mesh in the OpenShift cluster.
- The OpenShift CLI (**`/usr/local/bin/oc`**).
- The Istio CLI (**`istioctl`**).

As the **`student`** user on the **`workstation`** machine, use the **`lab`** command to validate the prerequisites for this exercise, and prepare the lab:

```
[student@workstation ~]$ lab comprehensive-review start
```

Instructions

The start script creates a new project named **`adoptapup`** and deploys all services except the **`adoption`** service. The **`news`** service deploys in a separate project named **`adoptapup-news`**.

The architecture of the "Adopt a Pup" application is as follows:

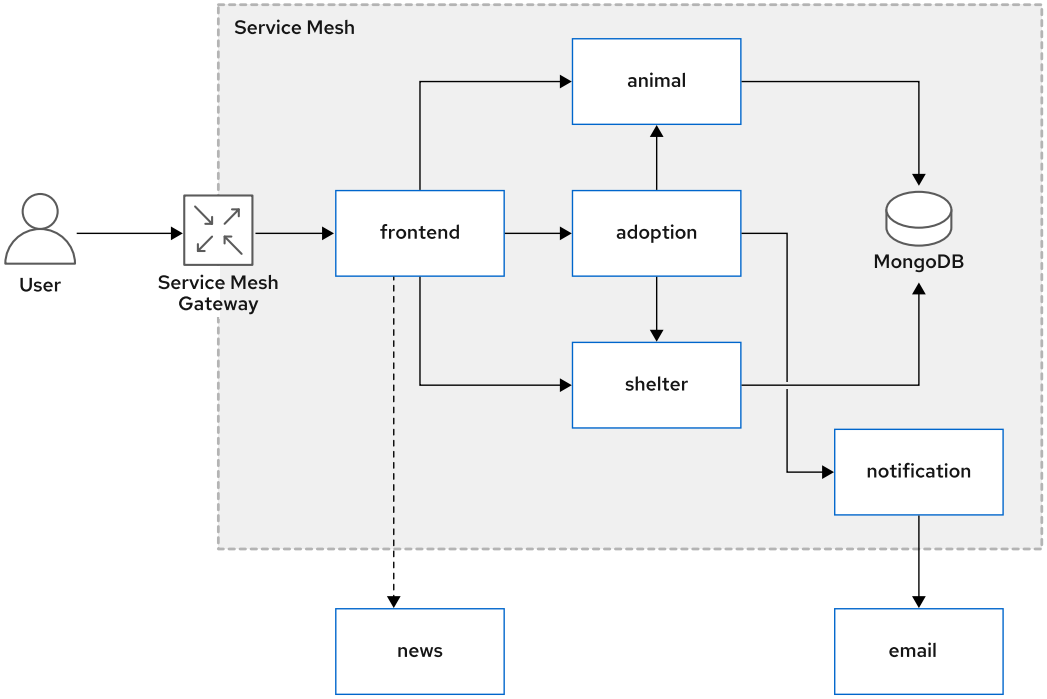


Figure Error.1: Adopt a Pup

The application consists of the following microservices:

animal
Manages a set of pups that can be adopted. Each pup belongs to a particular animal shelter.

shelter
Manages several animal shelters that take care of pups until they are adopted. A shelter can have one or more pups.

notification
Manages notifications that are sent to potential owners of pups. The notification service handles multiple notifications, such as emails and text messages. Currently, only notifications by email are supported.

email
Sends emails to users. The service mesh does not manage this service.

adoption
This service processes user requests for adoption. It is responsible for data validation and verifying that the user is eligible to adopt a pup.

news
This service periodically informs users about news regarding shelters and animals. This service is deployed in a separate project. The service mesh does not manage this service.

frontend
This service provides the web user interface for the application.

mongodb
The details for pups, shelters, and adoptions are stored in this MongoDB database.

The start script deployed these services using YAML files located in the **/home/student/DO328/labs/comprehensive-review** folder.

The start script deployed the MongoDB database and preloaded shelter and animal data into the database.

Note

Name all virtual service resources created in the **adoptapup** project as *service*-vs, where *service* is the service name, for example, **adoption-vs** and **animal-vs**.

Name all destination rule resources created in the **adoptapup** project as *service*-dr, where *service* is the service name, for example **adoption-dr** and **animal-dr**.

Name all service accounts created in the **adoptapup** project as *service*-sa, where *service* is the service name, for example **adoption-sa** and **animal-sa**.

Perform the following tasks:

1. Log in to OpenShift as the **developer** user and inspect the deployed microservices in the **adoptapup** and **adoptapup-news** projects. Verify that all the microservices, except the **adoption** microservice, are deployed and running.

   Remember to set the environment variables in the **/usr/local/etc/ocp4.config** file.

   1.1. Run the following command to load the environment variables in the **/usr/local/etc/ocp4.config** file:

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

   1.2. Log in to OpenShift:

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
>  -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_API}
Login successful.
...output omitted...
```

   1.3. If you are working with a different OpenShift project, set the current project to **adoptapup**:

```
[student@workstation ~]$ oc project adoptapup
Now using project "adoptapup" on server ...output omitted...
```

   1.4. Verify that the pods for the microservices deployed by the start script are **Running**:

```
[student@workstation ~]$ oc get pods
NAME                            READY   STATUS    RESTARTS   AGE
animal-v1-588956cfb5-88k9r      2/2     Running   0          30m
email-65bbdb599b-f75k8          1/1     Running   0          30m
frontend-v1-5f4965ff7d-ngfjz    2/2     Running   0          30m
mongodb-574f8ccb9c-98fpq        2/2     Running   0          30m
notification-5c7fd8bccf-nwndv   2/2     Running   0          30m
shelter-76dccc9cbc-rws47        2/2     Running   0          30m
```

1.5. Verify that the pod for the **news** microservice in the **adoptapup-news** project is **Running**:

```
[student@workstation ~]$ oc get pods -n adoptapup-news
NAME                    READY   STATUS    RESTARTS   AGE
news-6f7c8d4486-sn2pb   1/1     Running   0          32m
```

2. Verify that the data has been successfully loaded into MongoDB using the **check-mongo.sh** script in the **/home/student/DO328/labs/comprehensive-review** folder. If data is successfully loaded, you will see the raw JSON data from the database as follows:

```
[student@workstation ~]$ sh ~/DO328/labs/comprehensive-review/check-mongo.sh
Login successful.

You have access to the following projects ...output omitted...

  * adoptapup
...output omitted...
Checking if animal data is loaded into MongoDB...

{ "_id" : "d52a8d58-9024-49dd-92b6-d443c6049ffe", "animalName" : "Gus" ...output
 omitted...
...output omitted...

Checking if shelter data is loaded into MongoDB...

{ "_id" : "e038ae3c-592f-403e-9233-4b6eeab30e3c", "shelterName" : "Denver
 Doggos" ...output omitted...
...output omitted...
```

If the data is not loaded, then run the **load-mongo.sh** script in the **/home/student/DO328/labs/comprehensive-review** folder, and then rerun the **check-mongo.sh** script to verify success.

3. Create a service mesh gateway named **adoptapup-gateway** to allow external traffic to flow into the service mesh. Configure the gateway to listen for HTTP requests on port **80**.

3.1. Create a **gateway.yaml** file with the following contents. You can also copy the content from the **/home/student/DO328/solutions/comprehensive-review/gateway.yaml** file.

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: adoptapup-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
    - port:
        number: 80
        name: http
```

```
        protocol: HTTP
      hosts:
        - "*"
```

3.2. Create the gateway using the **oc create** command.

```
[student@workstation ~]$ oc create -f gateway.yaml
gateway.networking.istio.io/adoptapup-gateway created
```

4. Deploy the **adoption** service in the **adoptapup** project. Ensure that it is managed by Red Hat OpenShift Service Mesh.

   The container image for the adoption service is available at **quay.io/redhattraining/ossm-adoption-service:1.0**. The container serves requests on port **8080**.

   Name the OpenShift deployment and service resource as **adoption**, and then add a label **app: adoption** to the relevant resources. This service listens on port **8080**.

   Name the virtual service as **adoption-vs**. Configure the virtual service so that all requests to the **/adoption** endpoint, relative to the service mesh gateway URL, are routed to the **adoption** service.

   Using a web browser, access the front end user interface at the **/frontend** endpoint relative to the gateway URL. Verify that you can browse shelters and animals from the navigation pane.

Warning

If you have restarted your classroom VMs, or redeployed the MongoDB database pod after running the start script, then you must run the **load-mongo.sh** script as discussed in a previous step. You must do this because the MongoDB pod uses ephemeral storage and it does not persist data between restarts.

4.1. Create a YAML file named **adoption-service.yaml** with the following contents. You can also copy the YAML snippets from the **/home/student/DO328/solutions/comprehensive-review/adoption-service.yaml** file.

Start by creating the deployment resource:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: adoption
spec:
  selector:
    matchLabels:
      app: adoption
  replicas: 1
  template:
    metadata:
      labels:
        app: adoption
      annotations:
        sidecar.istio.io/inject: "true"
    spec:
      containers:
        - name: adoption
```

```
          image: quay.io/redhattraining/ossm-adoption-service:1.0
          imagePullPolicy: Always
          ports:
            - containerPort: 8080
---
```

Add the YAML snippet for creating a service:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: adoption
  name: adoption
spec:
  ports:
    - port: 8080
      protocol: TCP
      targetPort: 8080
  selector:
    app: adoption
---
```

Finally, create the virtual service as follows:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: adoption-vs
spec:
  hosts:
    - "*"
  gateways:
    - adoptapup-gateway
  http:
    - match:
        - uri:
            prefix: /adoption
      route:
        - destination:
            host: adoption
            port:
              number: 8080
```

Save your changes.

4.2. Create deployment, service, and virtual service resources.

```
[student@workstation ~]$ oc create -f adoption-service.yaml
deployment.apps/adoption created
service/adoption created
virtualservice.networking.istio.io/adoption-vs created
```

4.3. Before continuing to the next step, verify that the **`adoption`** service pod is **`Running`**.

```
[student@workstation ~]$ oc get pods
NAME                           READY   STATUS    RESTARTS   AGE
adoption-658c6fbcb4-jnnrm      2/2     Running   0          15m
...output omitted...
```

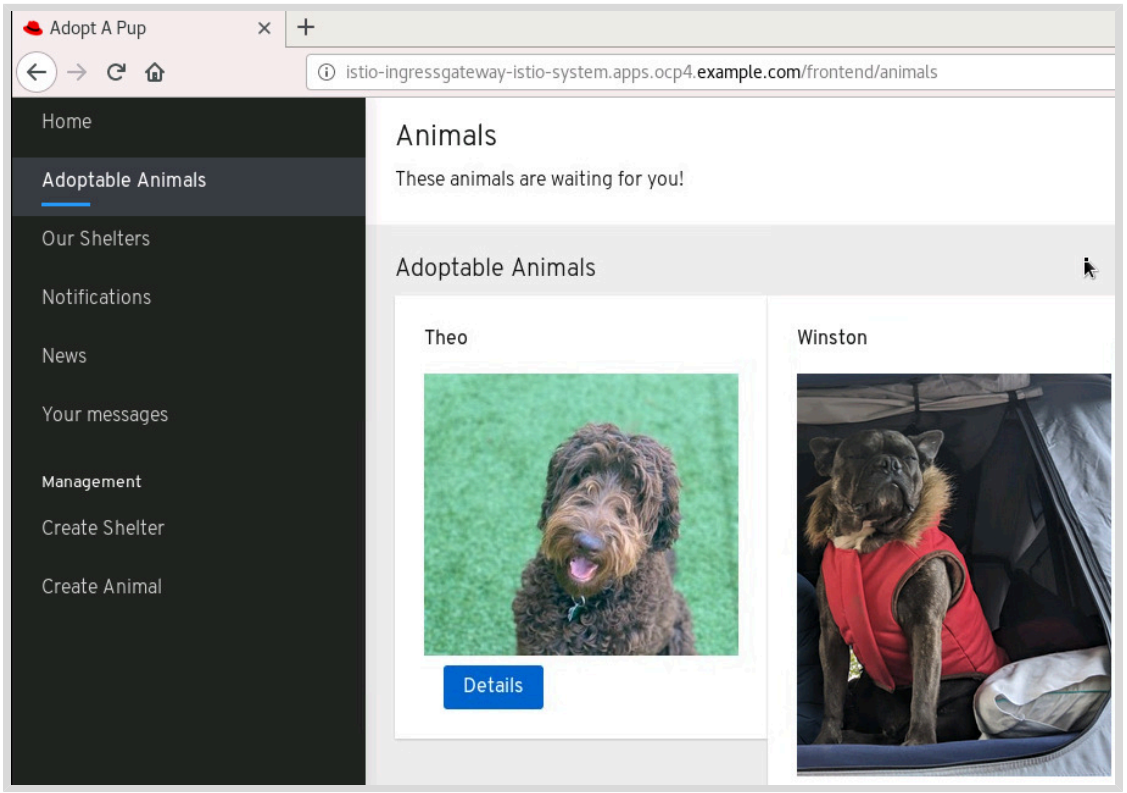
4.4. Run the **`oc get route`** command to get the service mesh gateway URL.

```
[student@workstation ~]$ GW_URL=$(oc get route istio-ingressgateway \
> -n istio-system -o jsonpath='{.spec.host}')
```

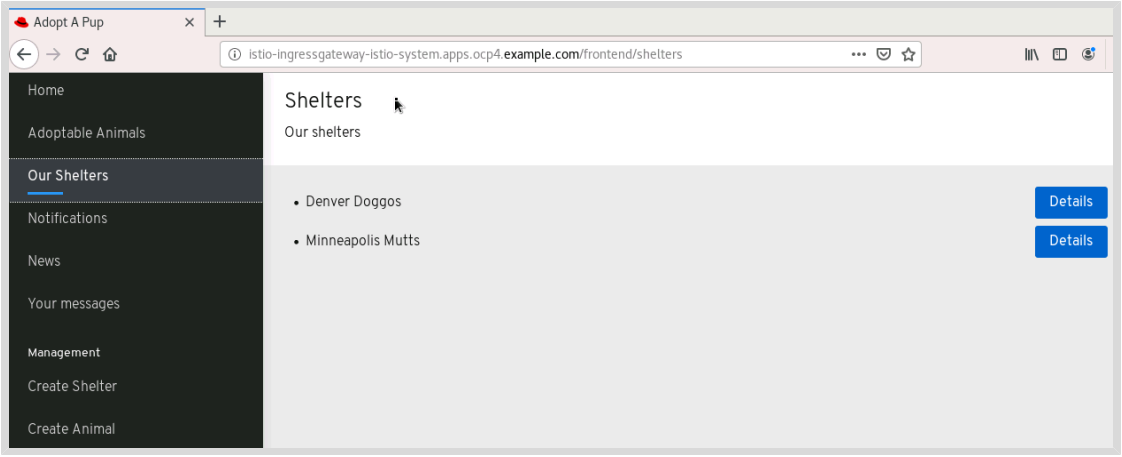
4.5. Access the front end for the "Adopt a Pup" application using the **`Firefox`** browser.

```
[student@workstation ~]$ firefox http://${GW_URL}/frontend &
```

4.6. Click **Adoptable Animals** to see a list of pups that are available for adoption.



Click **Our Shelters** to see a list of shelters. Click **Details** on the **Our Shelters** page to see details about the shelter, and to view the list of pups available in the shelter.



5. Click **News** in the navigation panel. The front end fails to fetch data from the **news** service (external to the service mesh).

   Configure the service mesh to allow the front end service to fetch the latest news from the **news** service.

   5.1. Get the route URL for the **news** service.

```
[student@workstation ~]$ oc get route news -n adoptapup-news
NAME   HOST/PORT                                  ...output omitted...
news   news-adoptapup-news.apps.ocp4.example.com   ...output omitted...
```

   5.2. The front end is unable to fetch news items from the **news** service, which runs outside the service mesh in a separate project. Create a service entry resource named **news** to allow service mesh to fetch data from the external **news** service.

   Create a file named **news-serviceentry.yaml** with the following content. Add the host name that you gathered from the **oc get route** command in the previous step.

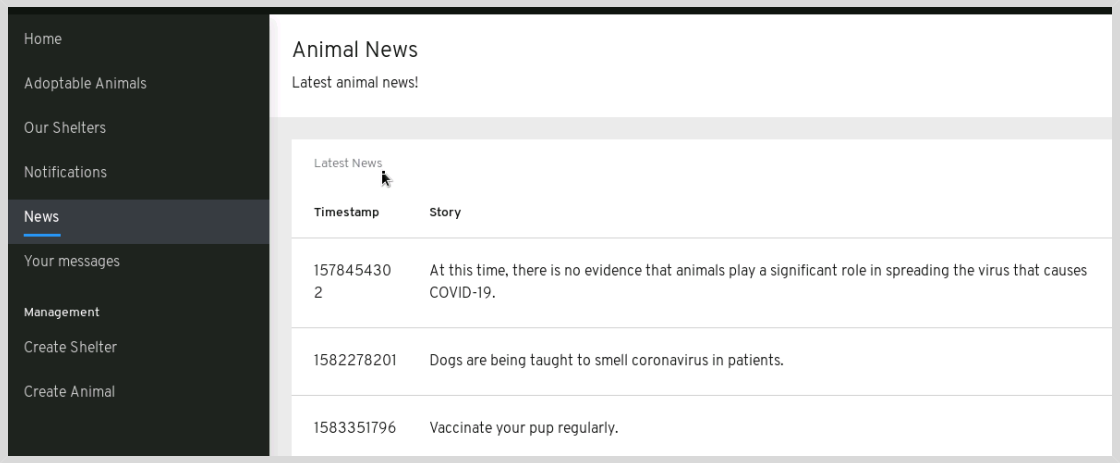
   You can also copy the YAML content from the **/home/student/DO328/solutions/comprehensive-review/news-serviceentry.yaml** file.

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: news
spec:
  hosts:
  - news-adoptapup-news.apps.ocp4.example.com
  ports:
  - name: http-80
    number: 80
    protocol: http
```

   5.3. Create a service entry resource.

```
[student@workstation ~]$ oc create -f news-serviceentry.yaml
serviceentry.networking.istio.io/news created
```

5.4. Wait for a few seconds for the configuration to propagate. Refresh the **News** page from the navigation panel. You should now see news items fetched from the **news** service.



Note

The news items are displayed randomly. Your list of items may differ from the example.

6. Version 2 of the **`frontend`** service introduces some user interface changes. The container image for version 2 is available at **`quay.io/redhattraining/ossm-adopt-a-pup-webapp:2.0`**.

Deploy version 2 of the **`frontend`** service as follows:

- Configure service mesh to route 80% of traffic to version 1 and the remainder to version 2.
- The container name and deployment name should be called **`frontend-v2`**. Add the label **`version: v2`** to the relevant resources.

Verify that the navigation panel background color is red in version 2, but not in version 1.

Note

You might have to refresh your browser several times before you can see the user interface changes. It can take some time for the page from version 2 to render as it loads the static assets (CSS and JavaScript files) for the first time.

6.1. Make a copy of the **`/home/student/DO328/labs/comprehensive-review/frontend-service-v1.yaml`** file.

```
[student@workstation ~]$ cp ~/DO328/labs/comprehensive-review/frontend-service-
v1.yaml \
> ~/frontend-service-v2.yaml
```

6.2. Delete the **`Service`** and **`VirtualService`** resource definitions (lines 41-73) from the **`frontend-service-v2.yaml`** file. Version 2 includes only the **`Deployment`** resource.

6.3. Change the **`frontend-service-v2.yaml`** file for deploying version 2 as follows:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: frontend
    version: v2
  name: frontend-v2
spec:
  selector:
    matchLabels:
      app: frontend
      version: v2
  replicas: 1
  template:
    metadata:
      labels:
        app: frontend
        version: v2
      annotations:
        sidecar.istio.io/inject: "true"
    spec:
      containers:
        - name: frontend-v2
          image: quay.io/redhattraining/ossm-adopt-a-pup-webapp:2.0
...output omitted...
```

6.4. Deploy version 2 of the **frontend** service.

```
[student@workstation ~]$ oc create -f frontend-service-v2.yaml
deployment.apps/frontend-v2 created
```

6.5. Verify that the pod for version 2 is **Running** before continuing to the next step.

```
[student@workstation ~]$ oc get pods
NAME                           READY   STATUS    RESTARTS   AGE
...output omitted...
frontend-v1-5f4965ff7d-ngfjz   2/2     Running   0          2h
frontend-v2-5449f5d8bc-c2fl8   2/2     Running   0          74s
...output omitted...
```

6.6. Create the destination rule resource for the **frontend** service. Create a file named **frontend-dest-rule.yaml** with YAML content as follows. You can also copy the YAML from the **/home/student/DO328/solutions/comprehensive-review/frontend-dest-rule.yaml** file.

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: frontend-dr
spec:
  host: frontend
  subsets:
```

```
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
```

6.7. Create the destination rule resource.

```
[student@workstation ~]$ oc create -f frontend-dest-rule.yaml
destinationrule.networking.istio.io/frontend-dr created
```

6.8. Edit the virtual service resource for the **`frontend`** service.

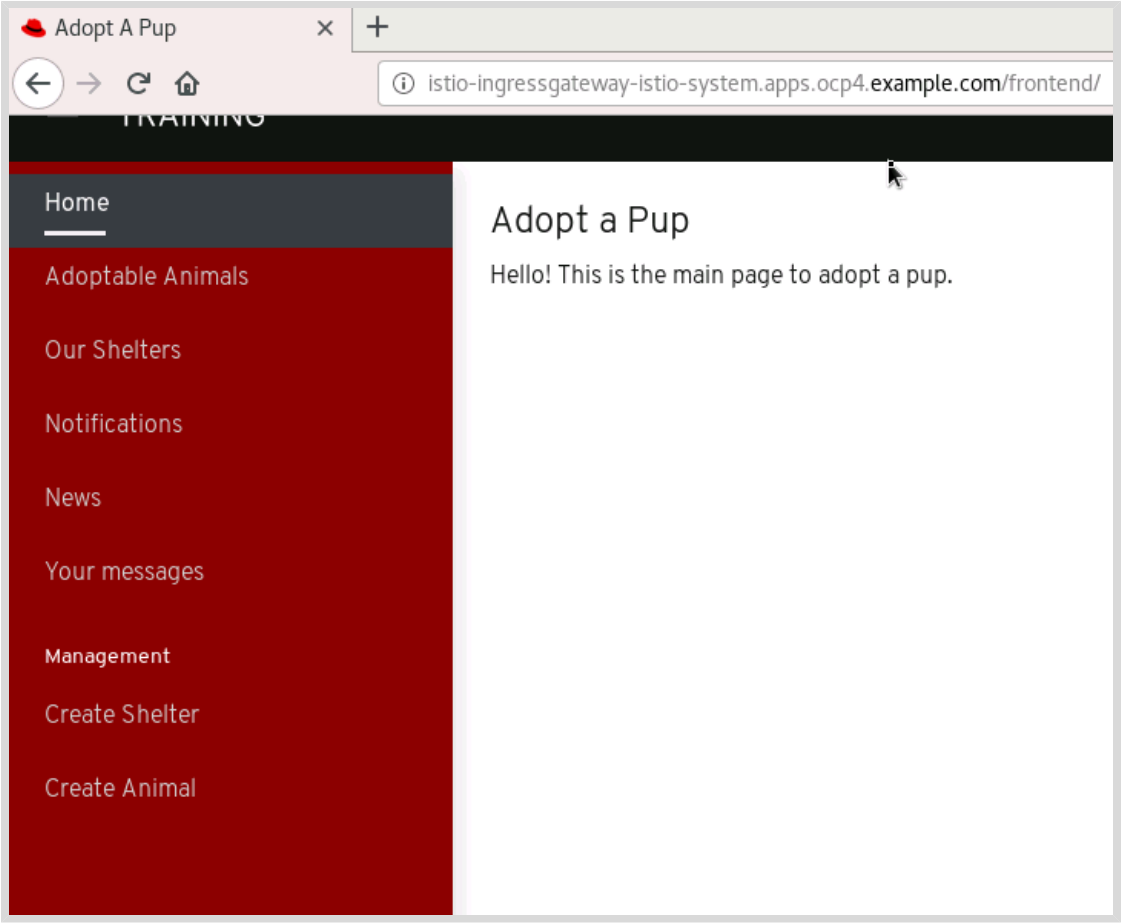
```
[student@workstation ~]$ oc edit vs frontend-vs
```

Configure weighted routing as follows:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: frontend-vs
  namespace: adoptapup
spec:
  gateways:
  - adoptapup-gateway
  hosts:
  - '*'
  http:
  - match:
    - uri:
        prefix: /frontend
    route:
    - destination:
        host: frontend
        subset: v1
        port:
          number: 3000
      weight: 80
    - destination:
        host: frontend
        subset: v2
        port:
          number: 3000
      weight: 20
```

Save your changes to the virtual service. Wait approximately 30 seconds while the deployment changes propagate through the service mesh.

6.9. Click **Home** in the navigation panel, and then refresh the page multiple times until you see the background color change to red.



7. Version 2 of the **`animal`** service focuses on improving performance. The container image for version 2 is available at **`quay.io/RedHatTraining/ossm-animal-service:2.0`**.

   Deploy version 2 of the **`animal`** service as follows:

   - The container name and deployment name should be called **`animal-v2`**. Add the label **`version: v2`** to the relevant resources.
   - Configure service mesh to mirror traffic from version 1 to version 2. Version 2 is not yet ready for production deployment, so responses to clients must still be sent exclusively from version 1.
   - Verify that traffic is mirrored and that you see the output from version 2 in the generated logs.

   7.1. Make a copy of the **`/home/student/DO328/labs/comprehensive-review/animal-service-v1.yaml`** file.

```
[student@workstation ~]$ cp ~/DO328/labs/comprehensive-review/animal-service-
v1.yaml \
> ~/animal-service-v2.yaml
```

   7.2. Delete the **`Service`** and **`VirtualService`** resource definitions (lines 28-60) from the **`animal-service-v2.yaml`** file. Version 2 includes only the **`Deployment`** resource.

7.3. Make changes to the **animal-service-v2.yaml** file for deploying version 2 as follows:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: animal
    version: v2
  name: animal-v2
spec:
  selector:
    matchLabels:
      app: animal
      version: v2
  replicas: 1
  template:
    metadata:
      labels:
        app: animal
        version: v2
      annotations:
        sidecar.istio.io/inject: "true"
    spec:
      containers:
        - name: animal-v2
          image: quay.io/redhattraining/ossm-animal-service:2.0
...output omitted...
```

7.4. Deploy version 2 of the **animal** service.

```
[student@workstation ~]$ oc create -f animal-service-v2.yaml
deployment.apps/animal-v2 created
```

7.5. Verify that the pod for version 2 is **Running** before continuing to the next step.

```
[student@workstation ~]$ oc get pods
NAME                          READY   STATUS    RESTARTS   AGE
...output omitted...
animal-v1-588956cfb5-88k9r    2/2     Running   0          3h
animal-v2-7c8fff7686-zlhfs    2/2     Running   0          10m
...output omitted...
...output omitted...
```

7.6. Create the destination rule resource for the **animal** service. Create a file named **animal-dest-rule.yaml** with YAML content as follows. You can also copy the YAML from the **/home/student/DO328/solutions/comprehensive-review/animal-dest-rule.yaml** file.

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: animal-dr
```

```
spec:
  host: animal
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
```

7.7. Create the destination rule resource.

```
[student@workstation ~]$ oc create -f animal-dest-rule.yaml
destinationrule.networking.istio.io/animal-dr created
```

7.8. Edit the virtual service resource for the **`animal`** service.

```
[student@workstation ~]$ oc edit vs animal-vs
```

Configure mirroring for the **`animal`** service as follows:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: animal-vs
  namespace: adoptapup
spec:
  gateways:
  - adoptapup-gateway
  hosts:
  - '*'
  http:
  - match:
    - uri:
        prefix: /animals
    route:
    - destination:
        host: animal
        subset: v1
        port:
          number: 8080
      weight: 100
    mirror:
      host: animal
      subset: v2
```

Save your changes to the virtual service. Wait approximately 30 seconds while the deployment changes propagate through the service mesh.

7.9. Click **Adoptable Animals** in the navigation panel, and then refresh the page multiple times to verify that the list of adoptable animals displays without errors.

7.10. Inspect the logs for version 2 and verify that requests sent to version 1 of the **`animal`** service are mirrored. You should see logs for incoming requests.

```
[student@workstation ~]$ oc logs animal-v2-7c8fff7686-zlhfs \
> -c animal-v2
...output omitted...animalservice.AnimalController : Getting 5 adoptable animals
 through animal-v2...
...output omitted...animalservice.AnimalController : Getting 5 adoptable animals
 through animal-v2...
...output omitted...animalservice.AnimalController : Getting 5 adoptable animals
 through animal-v2...
...output omitted...
```

8. Introduce a 3-second delay for all responses from the **shelter** service. Verify that you see delayed responses when you click **Our Shelters** in the navigation panel.

   8.1. Edit the virtual service resource for the **shelter** microservice.

```
[student@workstation ~]$ oc edit vs shelter-vs
```

   8.2. Add a 3-second delay to all responses as follows:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  annotations:
  name: shelter-vs
  namespace: adoptapup
spec:
  gateways:
  - adoptapup-gateway
  hosts:
  - '*'
  http:
  - match:
    - uri:
        prefix: /shelters
    route:
    - destination:
        host: shelter
        port:
          number: 8080
    fault:
      delay:
        percentage:
          value: 100
        fixedDelay: 3000ms
```

   Save your changes. Wait approximately 30 seconds for the service mesh to propagate the changes.

   8.3. Click **Our Shelters** in the navigation panel, and then refresh the page multiple times. Verify that the list of shelters displays after a 3-second delay.

   Click **Details** next to one of the shelters. Verify that shelter details display after a 3-second delay.

9. Configure network connections for the **adoption** service to retry failed requests consistent with the following policies:

- Configure the service mesh to allow **3** retry attempts. Each retry waits at most **5 seconds** before timing out.
- Retries must be triggered only when the response contains an HTTP status code of 500 and above.

9.1. Edit the virtual service resource for the **adoption** microservice.

```
[student@workstation ~]$ oc edit vs adoption-vs
```

9.2. Add configuration for retries as follows:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: adoption-vs
  namespace: adoptapup
spec:
  gateways:
  - adoptapup-gateway
  hosts:
  - '*'
  http:
  - match:
    - uri:
        prefix: /adoption
    route:
    - destination:
        host: adoption
        port:
          number: 8080
    retries:
      attempts: 3
      perTryTimeout: 5s
      retryOn: 5xx
```

Save your changes.

10. Configure a circuit breaker for the **adoption** service as follows:

- Allow 3 consecutive errors within 10 seconds before breaking the circuit.
- Allow 1 minute of recovery after the circuit breaks before activating the service to receive requests.

10.1. Create the destination rule resource for the **adoption** microservice in a file named **adoption-cb.yaml** as follows. You can also copy the YAML snippet from the **/home/student/DO328/solutions/comprehensive-review/adoption-cb.yaml** file.

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

```
  name: adoption-dr
spec:
  host: adoption
  trafficPolicy:
    outlierDetection:
      consecutiveErrors: 3
      interval: 10s
      baseEjectionTime: 1m
```

Save your changes.

10.2. Create the destination rule resource.

```
[student@workstation ~]$ oc create -f adoption-cb.yaml
destinationrule.networking.istio.io/adoption-dr created
```

11. Configure connection pooling for the **`notification`** service as follows:

- Allow only 5 concurrent connections at any given time to prevent overloading this service.
- Allow only 1 request per connection.

11.1. Create the destination rule resource for the **`notification`** microservice in a file named **`notification-pool.yaml`** as follows. You can also copy the YAML snippet from the **`/home/student/DO328/solutions/comprehensive-review/notification-pool.yaml`** file.

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: notification-dr
spec:
  host: notification
  trafficPolicy:
    connectionPool:
      tcp:
        maxConnections: 5
      http:
        maxRequestsPerConnection: 1
```

Save your changes.

11.2. Create the destination rule resource.

```
[student@workstation ~]$ oc create -f notification-pool.yaml
destinationrule.networking.istio.io/notification-dr created
```

12. Secure the services in the service mesh as follows:

- Ensure that traffic sent from the **`animal`** and **`shelter`** services to the MongoDB database is encrypted. Enable mutual authentication between these services and the MongoDB database.
- Configure the service mesh so that only the **`shelter`** service and version 1 of the **`animal`** service are authorized to communicate with the MongoDB database.

Name your resource **mongodb-auth-policy**.

12.1. Create service accounts for the **shelter** and **animal** services.

```
[student@workstation ~]$ oc create serviceaccount animal-sa
serviceaccount/animal-sa created
[student@workstation ~]$ oc create serviceaccount shelter-sa
serviceaccount/shelter-sa created
```

12.2. Assign the service accounts to pod deployments.

```
[student@workstation ~]$ oc set serviceaccount deployment animal-v1 animal-sa
deployment.apps/animal-v1 serviceaccount updated
[student@workstation ~]$ oc set serviceaccount deployment shelter shelter-sa
deployment.apps/shelter serviceaccount updated
```

12.3. Assigning a service account to a deployment terminates the current pod and creates a new pod. Verify that the pods for the **shelter** and **animal** services are **Running**.

```
[student@workstation ~]$ oc get pods
NAME                            READY   STATUS    RESTARTS   AGE
...output omitted...
animal-v1-7b6487966f-d5gdb      2/2     Running   0          54s
...output omitted...
shelter-84fcc987c9-n8n5t        2/2     Running   0          21s
```

12.4. Enable mutual authentication for the **mongodb** service.

Create a file named **mongodb-dr.yaml** with the following content. You can also copy the YAML content from the **/home/student/DO328/solutions/comprehensive-review/mongodb-dr.yaml** file.

```
apiVersion: "networking.istio.io/v1alpha3"
kind: "DestinationRule"
metadata:
  name: "mongodb-dr"
  namespace: "adoptapup"
spec:
  host: "mongodb.adoptapup.svc.cluster.local"
  trafficPolicy:
    tls:
      mode: ISTIO_MUTUAL
```

12.5. Create the destination rule resource for the **mongodb** service:

```
[student@workstation ~]$ oc create -f mongodb-dr.yaml
destinationrule.networking.istio.io/mongodb-dr created
```

12.6. Enable mutual authentication for the **shelter** service.

Create a file named **shelter-dr.yaml** with the following content. You can also copy the YAML content from the **/home/student/DO328/solutions/comprehensive-review/shelter-dr.yaml** file.

```
apiVersion: "networking.istio.io/v1alpha3"
kind: "DestinationRule"
metadata:
  name: "shelter-dr"
  namespace: "adoptapup"
spec:
  host: "shelter.adoptapup.svc.cluster.local"
  trafficPolicy:
    tls:
      mode: ISTIO_MUTUAL
```

12.7. Create the destination rule resource for the **`shelter`** service:

```
[student@workstation ~]$ oc create -f shelter-dr.yaml
destinationrule.networking.istio.io/shelter-dr created
```

12.8. Edit the destination rule resource for the **`animal`** service and enable mutual authentication.

```
[student@workstation ~]$ oc edit dr animal-dr
```

Add the following **`trafficPolicy`** attribute:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: animal-dr
  namespace: adoptapup
...output omitted...
spec:
  host: animal
  subsets:
  ...output omitted...
  trafficPolicy:
    tls:
      mode: ISTIO_MUTUAL
```

12.9. Configure authorization policies to allow only the **`animal`** and **`shelter`** services to communicate with MongoDB.

Create a file named **`mongodb-security-policy.yaml`** with the following contents. You can also copy the YAML content from the **`/home/student/DO328/solutions/comprehensive-review/mongodb-security-policy.yaml`** file.

```
apiVersion: "security.istio.io/v1beta1"
kind: "AuthorizationPolicy"
metadata:
  name: "mongodb-auth-policy"
spec:
  selector:
    matchLabels:
      app: mongodb
```

```
  rules:
  - from:
    - source:
        principals: ["cluster.local/ns/adoptapup/sa/animal-sa"]
    - source:
        principals: ["cluster.local/ns/adoptapup/sa/shelter-sa"]
    to:
    - operation:
        ports: ["27017"]
```

12.10. Create the authorization policy resources.

```
[student@workstation ~]$ oc create -f mongodb-security-policy.yaml
authorizationpolicy.security.istio.io/mongodb-auth-policy created
```

12.11. Verify that the **`animal`** and **`shelter`** services can fetch data from MongoDB. Click **Adoptable Animals** and **Our Shelters** in the navigation pane and verify that the list of animals and shelters is displayed.

Evaluation

Grade your work by running the **`lab comprehensive-review grade`** command from your **`workstation`** machine. Correct any reported failures and rerun the script until successful.

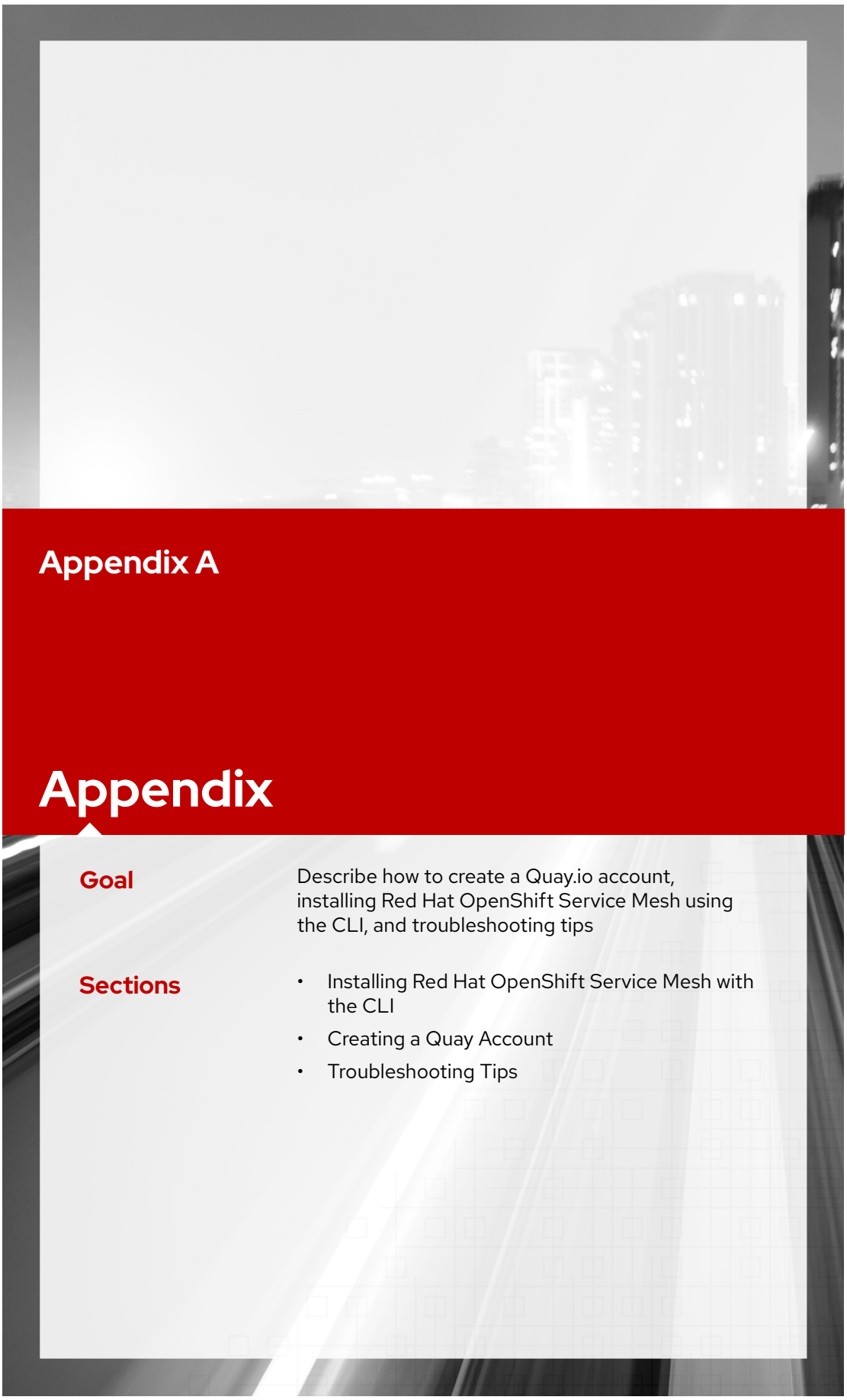
```
[student@workstation ~]$ lab comprehensive-review grade
```

Finish

On the **`workstation`** machine, use the **`lab`** command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation ~]$ lab comprehensive-review finish
```

This concludes the lab.



Appendix A

Appendix

Goal

Describe how to create a Quay.io account, installing Red Hat OpenShift Service Mesh using the CLI, and troubleshooting tips

Sections

- Installing Red Hat OpenShift Service Mesh with the CLI
- Creating a Quay Account
- Troubleshooting Tips

Installing Red Hat OpenShift Service Mesh with the CLI

Objectives

After completing this section, you should be able to install OpenShift Service Mesh on Red Hat OpenShift Container Platform with the CLI

Installing OpenShift Service Mesh Using the CLI

You can also install OpenShift Service Mesh using the **oc** CLI. To install operators, you must have **cluster-admin** privileges.

Log in Red Hat OpenShift using an account with **cluster-admin** privileges.

```
[user@host ~]$ oc login -u USER -p PASSWORD OCP4_API
```

Installing the Elasticsearch Operator

The installation of the Elasticsearch Operator involves the following steps:

1. Create a Subscription object YAML file to subscribe the **openshift-operators** namespace to the Elasticsearch Operator, for example, *elasticsearch.yaml*.

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: elasticsearch-operator
  namespace: openshift-operators (1)
spec:
  channel: "4.3" (2)
  name: elasticsearch-operator (3)
  source: redhat-operators (4)
  sourceNamespace: openshift-marketplace
  installPlanApproval: Automatic
```

(1) Namespace used to install the operator.
(2) Stream of operator versions.
(3) Name of the operator to subscribe.
(4) Source that provides the operator.

2. Create the Subscription object applying the YAML file.

```
[user@host ~]$ oc apply -f elasticsearch.yaml
subscription.operators.coreos.com/elasticsearch-operator created
```

3. Check the status of the Operator installation.

```
[user@host ~]$ oc describe sub elasticsearch-operator \
> -n openshift-operators
Name:         elasticsearch-operator
Namespace:    openshift-operators
...output omitted...
Message:               all available catalogsources are healthy
...output omitted...
```

Installing the Jaeger Operator

Installing the Jaeger Operator involves the following steps:

1. Create a Subscription object YAML file to subscribe the **openshift-operators** namespace to the Jaeger Operator, for example, *jaeger.yaml*.

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: jaeger-product
  namespace: openshift-operators
spec:
  name: jaeger-product
  source: redhat-operators
  sourceNamespace: openshift-marketplace
  channel: "stable"
  installPlanApproval: Automatic
```

2. Create the Subscription object applying the YAML file.

```
[user@host ~]$ oc apply -f jaeger.yaml
subscription.operators.coreos.com/jaeger-product created
```

3. Check the status of the Operator installation.

```
[user@host ~]$ oc describe sub jaeger-product \
> -n openshift-operators
Name:         jaeger-product
Namespace:    openshift-operators
...output omitted...
Message:               all available catalogsources are healthy
...output omitted...
```

Installing the Kiali Operator

Installing the Kiali Operator involves the following steps:

1. Create a Subscription object YAML file to subscribe the **openshift-operators** namespace to the Kiali Operator, for example, *kiali.yaml*.

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
```

```
  name: kiali-ossm
  namespace: openshift-operators
spec:
  name: kiali-ossm
  source: redhat-operators
  sourceNamespace: openshift-marketplace
  channel: "stable"
  installPlanApproval: Automatic
```

2. Create the Subscription object applying the YAML file.

```
[user@host ~]$ oc apply -f kiali.yaml
subscription.operators.coreos.com/kiali-ossm created
```

3. Check the status of the Operator installation.

```
[user@host ~]$ oc describe sub kiali-ossm \
> -n openshift-operators
Name:         kiali-ossm
Namespace:    openshift-operators
...output omitted...
Message:              all available catalogsources are healthy
...output omitted...
```

Installing the OpenShift Service Mesh Operator

Installing the OpenShift Service Mesh Operator involves the following steps:

1. Create a Subscription object YAML file to subscribe the **`openshift-operators`** namespace to the Red Hat OpenShift Service Mesh Operator, for example, *service-mesh-subscription.yaml*.

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: servicemeshoperator
  namespace: openshift-operators
spec:
  channel: '1.0'
  name: servicemeshoperator
  source: redhat-operators
  sourceNamespace: openshift-marketplace
  installPlanApproval: Automatic
```

2. Create the Subscription object applying the YAML file.

```
[user@host ~]$ oc apply -f service-mesh-subscription.yaml
subscription.operators.coreos.com/servicemeshoperator created
```

3. Check the status of the Operator installation.

```
[user@host ~]$ oc describe sub servicemeshoperator \
> -n openshift-operators
Name:         servicemeshoperator
Namespace:    openshift-operators
...output omitted...
  Message:               all available catalogsources are healthy
...output omitted...
```

Creating the OpenShift Service Mesh Control Plane

Red Hat recommends that you deploy the control plan in a separate project. The following describes how to deploy the control plane using the CLI.

1. Log in Red Hat OpenShift as a developer user.

```
[user@host ~]$ oc login -u USER -p PASSWORD RHT_OCP4_API
```

2. Create a project, for example, **`istio-system`**.

```
[user@host ~]$ oc new-project istio-system
Now using project "istio-system" on server "https://api.ocp4.example.com:6443".
...output omitted...
```

3. Create a **`ServiceMeshControlPlane`** object YAML file, for example, *istio-basic-installation.yaml*.

```
apiVersion: maistra.io/v1
kind: ServiceMeshControlPlane
metadata:
  name: basic-install (1)
  namespace: istio-system (2)
spec:
  istio:
    gateways: (3)
      istio-egressgateway:
        autoscaleEnabled: false
      istio-ingressgateway:
        autoscaleEnabled: false
  mixer:
    policy:
      autoscaleEnabled: false
    telemetry:
      autoscaleEnabled: false
  pilot: (4)
    autoscaleEnabled: false
    traceSampling: 100
  kiali: (5)
    enabled: true
  grafana: (6)
    enabled: true
  tracing: (7)
```

```
    enabled: true
    jaeger:
      template: all-in-one
```

1. Name assigned to the Control Plane.
2. Namespace where the Control Plane is deployed.
3. Istio gateways configuration. Disables autoscaling on the ingress and egress gateways.
4. Pilot configuration. Disables autoscaling and sets the percentage of trace sampling.
5. Kiali configuration. Enables Kiali to visualize traffic in the Service Mesh.
6. Grafana configuration. Enables Grafana to analyze and monitor the Service Mesh.
7. Jaeger configuration. Enables Jaeger to trace traffic in the Service Mesh.

4. Deploy the control plane.

```
[user@host ~]$ oc create -n istio-system \
> -f istio-basic-installation.yaml
servicemeshcontrolplane.maistra.io/basic-install created
```

5. Check the status of the control plane installation.

```
[user@host ~]$ oc get smcp -n istio-system
NAME            READY
basic-install   True
```

You must create a new **`ServiceMeshMemberRoll`** for each new control plane installation. To create a new **`ServiceMeshMemberRoll`**:

1. Create a **`ServiceMeshControlPlane`** object YAML file, for example, *service-mesh-member-roll.yaml*.

```
apiVersion: maistra.io/v1
kind: ServiceMeshMemberRoll
metadata:
  name: default
  namespace: istio-system
spec:
  members:
    - a-project
```

   The control plane manages projects listed as **`members`**.

2. Deploy the **`ServiceMeshMemberRoll`**.

```
[user@host ~]$ oc create -n istio-system \
> -f service-mesh-member-roll.yaml
servicemeshmemberroll.maistra.io/default created
```

Adding or Removing a Project from the Control Plane

Only projects listed on the **`ServiceMeshMemberRoll`** are managed by the service mesh. To add or remove a project from the control plane:

1. Log in Red Hat OpenShift Container.

2. Edit the **`ServiceMeshMemberRoll`** resource.

```
[user@host ~]$ oc edit smmr -n istio-system
```

3. Modify the YAML to add or remove project **`members`** and save the changes.

Creating a Quay Account

Objectives

After completing this section, you should be able to describe how to create a Quay account, and public container image repositories for labs in the course

Creating a Quay Account

You need a Quay account to create one or more *public* container image repositories for the labs in this course. If you already have a Quay account, you can skip the steps to create a new account listed in this appendix.

Important

If you already have a Quay account, ensure that you only create *public* container image repositories for the labs in this course. The lab grading scripts and instructions require unauthenticated access to pull container images from the repository.

To create a new Quay account, perform the following steps:

1. Navigate to https://quay.io using a web browser.

2. Click **Sign in** in the upper-right corner (next to the search bar).

3. On the **Sign in** page, you can log in using your Google or GitHub credentials (created in Appendix A).

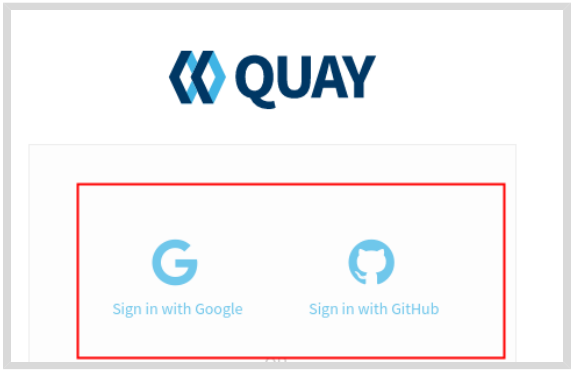


Figure A.1: Sign in using Google or GitHub credentials.

Alternatively, click **Create Account** to create a new account.

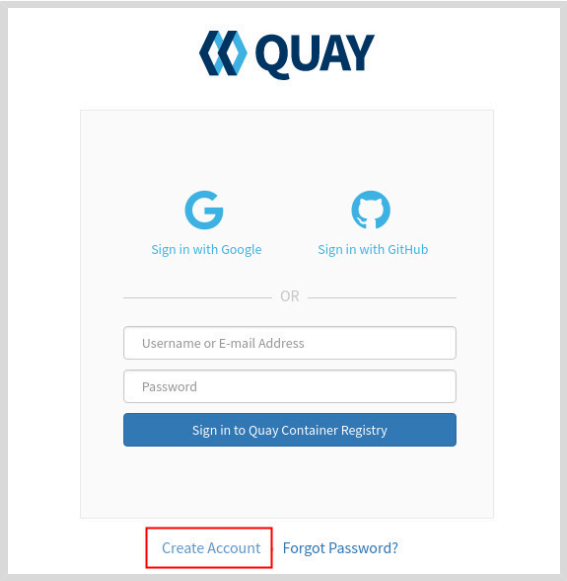


Figure A.2: Creating a new account

4. If you chose to skip the Google or GitHub log-in method and instead opted to create a new account, you will receive an email with instructions on how to activate your Quay account. Verify your email address and then sign in to the Quay website with the username and password you provided during account creation.

5. After you have logged in to Quay you can create new image repositories by clicking **Create New Repository** on the **Repositories** page.

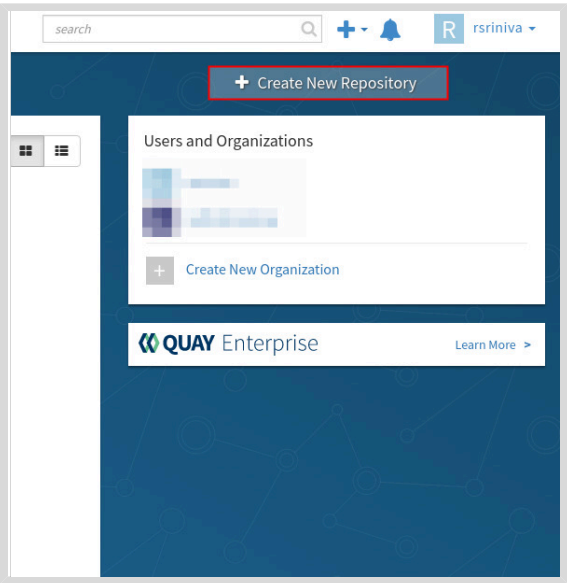


Figure A.3: Creating a new image repository

Alternatively, click the plus icon (+) in the upper-right corner (to the left of the bell icon), and then click **New Repository**.

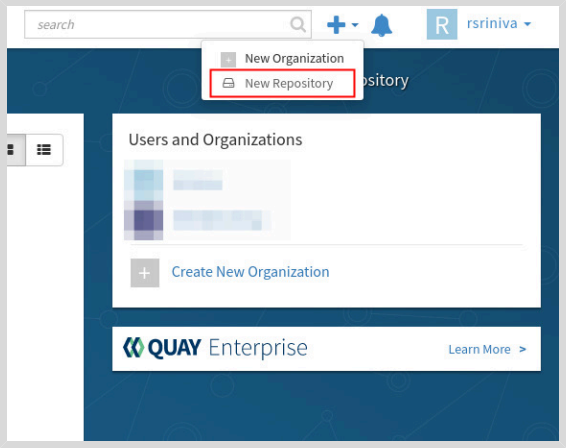


Figure A.4: Creating a new image repository

6. Enter a name for the repository as per your lab instructions. Ensure that you select the **Public**, and **Empty Repository** options.

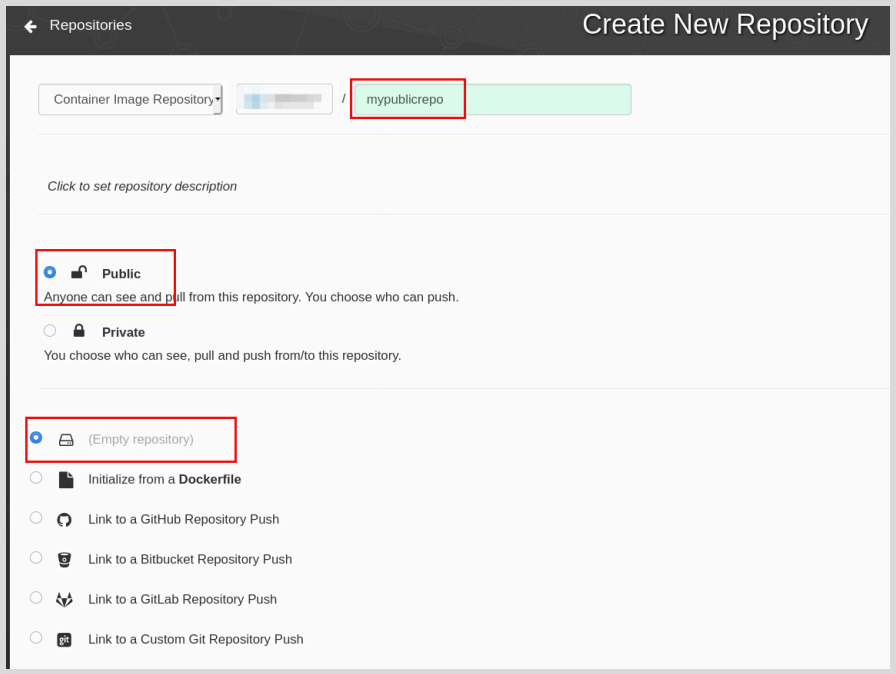


Figure A.5: Creating a new image repository

Click **Create Public Repository** to create the repository.

References

Getting Started with Quay.io
https://docs.quay.io/solution/getting-started.html

Troubleshooting Tips

Objectives

After completing this section, you should be able to troubleshoot and resolve general issues for labs in the course

OpenShift Log In Failure

Logging in as the OpenShift **`developer`** or **`admin`** user may sometimes fail with the following errors:

- **`error: EOF`**
- **`error: net/http: TLS handshake timeout`**
- **`Error from server (InternalError): Internal error occurred: unexpected response: 503`**

You may also see a lab start script printing a **`FAIL`** message when it tries to log in to OpenShift.

- If you experience this issue after you started your VMs, then wait a few minutes until all the OpenShift services are started and ready, and then try logging in again.
- Log in to the **`utility`** VM as the **`lab`** user, and inspect the status of your OpenShift nodes using the **`oc get nodes`** command.

```
[student@workstation ~]$ ssh lab@utility
[lab@utility ~]$ oc get nodes
NAME       STATUS   ROLES           AGE   VERSION
master01   Ready    master,worker   28d   v1.17.1+3f6f40d
master02   Ready    master,worker   28d   v1.17.1+3f6f40d
master03   Ready    master,worker   28d   v1.17.1+3f6f40d
worker01   Ready    worker          28d   v1.17.1+3f6f40d
worker02   Ready    worker          28d   v1.17.1+3f6f40d
worker03   Ready    worker          28d   v1.17.1+3f6f40d
```

All your nodes should be in **`Ready`** state.

Service Mesh Installation Issues

Your Red Hat OpenShift Service Mesh installation may fail due to transient network issues, or timeouts.

- Run the **`lab uninstall-mesh start`** script to clean up the failed install, and retry your installation. You can also run the **`lab install-mesh solve`** script to perform an automated install of service mesh.
- Verify that a service mesh control plane named **`basic-install`** is created by running the **`oc get smcp -n istio-system`** command.

```
NAME            READY   STATUS             ...output omitted...
basic-install   9/9     InstallSuccessful  ...output omitted...
```

- Verify that a service mesh member roll resource named **`default`** is created by running the **`oc get smmr -n istio-system`** command.

```
NAME      READY   STATUS       AGE
default   0/0     Configured   4m46s
```

- The **`oc get pods -n istio-system`** command should show all pods in running state.

```
NAME                                     READY   STATUS    RESTARTS   AGE
grafana-5756fc9795-kwbt9                 2/2     Running   0          6m3s
istio-citadel-6d5f6954b-7vt8n            1/1     Running   0          7m57s
istio-egressgateway-686897485c-hbstt     1/1     Running   0          6m23s
istio-galley-7d785cd74-jmv65             1/1     Running   0          7m18s
istio-ingressgateway-69d89fbdfc-z87nh    1/1     Running   0          6m23s
istio-pilot-774dbf49b8-4dvbj             2/2     Running   0          6m36s
istio-policy-548f54dc4f-5whzv            2/2     Running   0          7m
istio-sidecar-injector-55f45f76b5-rm8bx  1/1     Running   0          6m17s
istio-telemetry-85c546dc76-thr7m         2/2     Running   0          7m
jaeger-799ff9bcc7-lrl4r                  2/2     Running   0          7m18s
kiali-5d7f76d45d-pdct2                   1/1     Running   0          5m30s
prometheus-79fb5bf69d-rz7hd              2/2     Running   0          7m38s
```

Envoy Proxy Sidecar Injection Failures

Run the **`oc get pods`** command in your project, and verify that you can see two containers (2/2) in Running state. If you see only one container for service mesh managed applications, it means proxy sidecar injection failed.

If you do not see the Envoy proxy sidecar injected into applications deployed on service mesh, then verify the following:

- You have added your project to the service mesh member roll resource.

  Run the **`oc get smmr default -n istio-system -o yaml`** command and verify that your project appears in the member list.

- You have added the **`sidecar.istio.io/inject`** annotation in the applications deployment resource and set its value to **`"true"`**.

- Run the **`oc delete pod`** ***`pod_name`*** to delete the pod. OpenShift will spawn a new pod and re-inject the sidecar proxy.

Project Deletion Failure in Lab Scripts

You are asked to run the lab finish script after you complete each lab. This script may sometimes fail to delete the OpenShift project due to timeouts, and you may see a **`FAIL`** message being displayed.

You can safely ignore this message. Verify if the project still exists, and then manually delete the project using the **`oc delete project`** command.