LFS243:
Service Mesh Fundamentals

- V.10.04.2022 -

Lab 1.1 - Install Kubernetes and Create a Cluster

Overview

In this lab, you'll be installing Kubernetes and creating three Kubernetes clusters, one for each of the service mesh technologies you'll be using in this course: Linkerd, Istio, and Consul.

To follow along with this course, there are a few ways that you can implement a lab environment:

- Creating a Kind cluster locally, like on your Mac
- Creating a Kind cluster on a Linux desktop with a GUI, like Ubuntu
- Creating a Kubernetes cloud cluster, like GKE or EKS.

All of the labs will work through this course regardless of what option you choose. However, there will be differences. For example, you'll see in some of the labs that Kind is stopped and started for each cluster. If you're using GKE, you wouldn't do that. Instead, you'd just have three separate GKE clusters that you run workloads on.

To make this simple, free, and as consistent as possible, most of the labs will use an easily replicable and easy-to-follow installation method using Kubernetes in Docker (kind). All subsequent lab exercises assume you are using a **`kind`** cluster. For the purposes of keeping the explanations simple, we'll use the phrase "VM" to talk about the Kind cluster. "VM" could mean your local computer or it could mean an Ubuntu desktop. To keep things interchangeable, we'll use "VM".

Please keep in mind that if you decide to use another type of Kubernetes cluster, for example, in the cloud, it may accrue cost depending on how many credits you have on your cloud account. Ensure to use a cloud calculator for confirmation.

Kind consists of:

- The necessary packages (Go, container image builder, cluster creation, etc.)
- The CLI (**`kind`**) to use Kind
- Container images (Docker) to run Systemd, Kubernetes

Throughout this course, you will only be using one cluster at a time, and the other clusters must be stopped so they don't interfere with the one that's being used. Each lab will tell you when you need to start or stop a cluster, and you'll be given the commands to use each time.

1. You will use a virtual machine (VM) to host your Kubernetes clusters. Using your preferred cloud provider (please note the cloud provider will accrue cost) or hypervisor, provision an Ubuntu VM (version 20.04 LTS) with the following characteristics:

    a. The VM must have at least 2 CPUs, 8 GB of memory, and 30 GB of boot disk space.

    b. The VM must use an IP address that is reachable from your computer.

    c. The VM must have network access for TCP ports 80 (HTTP), 443 (HTTPS), and 8080 (multiple graphical user interfaces). You may need to add firewall rules that permit this activity; always be cautious when altering firewall rules so you do not inadvertently allow unwanted network traffic.

2. Log into your Ubuntu VM through a command-line interface, like SSH.

3. The latest version of Docker Engine must be installed in the Ubuntu VM. See https://docs.docker.com/engine/install/ubuntu/#install-using-the-repository if you need instructions on installing Docker Engine in Ubuntu. Follow the instructions under the **Install using the repository** heading. Skip the instruction about installing a specific version, because you'll be installing the latest version in the item before that, and stop when you reach the "Install from a package" heading.

4. To test that your Docker installation is working, issue this command and confirm that your output looks similar to the text below:

```
yourname@ubuntu-vm:~$ sudo docker run hello-world

Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
0e03bdcc26d7: Pull complete
Digest:
sha256:49a1c8800c94df04e9658809b006fd8a686cab8028d33cfba2cc049724
254202
Status: Downloaded newer image for hello-world:latest
Hello from Docker!

This message shows that your installation appears to be working
correctly.
```

```
To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the
Docker Hub.
   (amd64)
3. The Docker daemon created a new container from that image
which runs the
   executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client,
which sent it
   to your terminal.

To try something more ambitious, you can run an Ubuntu container
with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/

For more examples and ideas, visit:
https://docs.docker.com/get-started/
```

5. To move forward with Kind, you must belong to the **`docker`** user group. Join it by issuing the command below.

```
yourname@ubuntu-vm:~$ sudo usermod -aG docker $USER

[no output if the command succeeds]
```

6. You will need to restart your session after issuing the previous command so that the change in group membership takes effect. You can restart your session simply by logging out and back in.

```
yourname@ubuntu-vm:~$ exit

[existing session will close; start another command-line session
to log back into the Ubuntu VM]
```

7. **`kubectl`** must be installed in the VM. See the [Install kubectl binary with curl on Linux documentation page] for instructions on installing **`kubectl`**. Follow the instructions under the “Install kubectl binary with curl on Linux” heading to download the latest release. When you’re done, verify that **`kubectl`** is installed by issuing this command and confirming that your output looks similar to the text below:

```
yourname@ubuntu-vm:~$ kubectl version --short

Client Version: version.Info{Major:"1", Minor:"19",
GitVersion:"v1.19.3", GitCommit:"1e11e4a2108024935ecfcb2912226c
edeafd99df", GitTreeState:"clean",
BuildDate:"2020-10-14T12:50:19Z", GoVersion:"go1.15.2",
Compiler:"gc", Platform:
"linux/amd64"}
```

8. **helm** must be installed in the VM. For instructions on installing it, see the [“From Script” page](). Follow the instructions under the “From Script” heading to fetch and execute the installer script. When you’re done, you should verify that **helm** is installed. Enter the following command and make sure your output looks similar to the output below:

```
yourname@ubuntu-vm:~$ helm version

version.BuildInfo{Version:"v3.4.0",
GitCommit:"7090a89efc8a18f3d8178bf47d2462450349a004",
GitTreeState:"clean", GoVersion:"go1.14.10"}
```

9. The next step is to enter the following three commands in the Ubuntu VM to install **kind** and set the permissions correctly. See the [kind documentation]() for full installation information. Please note that the version in this lab may be a different version that you see on the Kind documentation, as the version may have been updated. The recommendation is to check that prior.

```
yourname@ubuntu-vm:~$ curl -Lo ./kind
https://kind.sigs.k8s.io/dl/v0.14.0/kind-linux-amd64


% Total     % Received % Xferd Average Speed Time Time Time
Current
Dload Upload Total Spent Left Speed
100    97 100    97  0  0   184 0 --:--:-- --:--:-- --:--:-- 184
100   629 100   629  0  0   879 0 --:--:-- --:--:-- --:--:-- 879
100 9900k 100 9900k  0  0 10.9M 0 --:--:-- --:--:-- --:--:--
10.9M

yourname@ubuntu-vm:~$ chmod +x ./kind

[no output if the command succeeds]
```

```
yourname@ubuntu-vm:~$ sudo mv ./kind /usr/local/bin/kind

[no output if the command succeeds]
```

10. To confirm that Kind was installed, run the following and you should see output similar to the below (it might be a different version number depending on what point in time you're doing this lab):

```
yourname@ubuntu-vm:~$ helm version
kind v0.8.1 go1.14.2 linux/amd64
```

11. After **kind** is installed, you will provision the first cluster, which will be for the Consul service mesh, by running the command below in the Ubuntu VM. The command will create the Kubernetes cluster and allow it to bind to port 80 (HTTP) and 443 (HTTPS). This is necessary for making sure your Ingress controller can route traffic. Note that it may take a few minutes for the provisioning to complete.

```
yourname@ubuntu-vm:~$ cat <<EOF | kind create cluster --name
consul --config=-
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
  kubeadmConfigPatches:
  - |
    kind: InitConfiguration
    nodeRegistration:
      kubeletExtraArgs:
        node-labels: "ingress-ready=true"
  extraPortMappings:
  - containerPort: 80
    hostPort: 80
    protocol: TCP
  - containerPort: 443
    hostPort: 443
    protocol: TCP
EOF

Creating cluster "consul" ...
✓ Ensuring node image (kindest/node:v1.18.2) 🖼
✓ Preparing nodes 📦
✓ Writing configuration 📜
✓ Starting control-plane 🕹️
✓ Installing CNI 🔌
```

```
✓ Installing StorageClass 💾
Set kubectl context to "kind-consul"
You can now use your cluster with:

kubectl cluster-info --context kind-consul

Thanks for using kind! 😊
```

12. Once **kind** finishes creating your new cluster for Consul, point **kubectl** at the cluster and confirm that the Kubernetes master and KubeDNS are running:

```
yourname@ubuntu-vm:~$ kubectl cluster-info --context kind-consul

Kubernetes master is running at https://127.0.0.1:43963
KubeDNS is running at
https://127.0.0.1:43963/api/v1/namespaces/kube-system/services/ku
be-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl
cluster-info dump'.
```

13. Stop the cluster for Consul by issuing this command (give this command a few seconds to run as the output may not happen right away):

```
yourname@ubuntu-vm:~$ docker stop $(docker container ls -a -f
name=consul-control-plane -q)

2e64e6e909e1
```

14. Now you will provision the second cluster, for the Istio service mesh, by running the command below in the Ubuntu VM. This is identical to the command you issued for Consul except it gives this cluster a different name—”**istio**” instead of “**consul**”. Note that it may take a few minutes for the provisioning to complete.

```
yourname@ubuntu-vm:~$ cat <<EOF | kind create cluster --name
istio --config=-
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
  kubeadmConfigPatches:
  - |
    kind: InitConfiguration
    nodeRegistration:
```

```
      kubeletExtraArgs:
        node-labels: "ingress-ready=true"
  extraPortMappings:
  - containerPort: 80
    hostPort: 80
    protocol: TCP
  - containerPort: 443
    hostPort: 443
    protocol: TCP
EOF

Creating cluster "istio" ...
✓ Ensuring node image (kindest/node:v1.18.2) 🖼
✓ Preparing nodes 📦
✓ Writing configuration 📜
✓ Starting control-plane 🕹️
✓ Installing CNI 🔌
✓ Installing StorageClass 💾
Set kubectl context to "kind-istio"
You can now use your cluster with:

kubectl cluster-info --context kind-istio

Thanks for using kind! 😊
```

15. Once **kind** finishes creating your new cluster for Istio, point **kubectl** at the cluster and confirm that the Kubernetes master and KubeDNS are running:

```
yourname@ubuntu-vm:~$ kubectl cluster-info --context kind-istio

Kubernetes master is running at https://127.0.0.1:33559
KubeDNS is running at
https://127.0.0.1:33559/api/v1/namespaces/kube-system/services/ku
be-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl
cluster-info dump'.
```

16. Stop the cluster for Istio by issuing this command:

```
yourname@ubuntu-vm:~$ docker stop $(docker ps -a -f
name=istio-control-plane -q)

2d1d09fadf21
```

17. Now you will provision the third cluster, for the Linkerd service mesh, by running the command below in the Ubuntu VM. This cluster will be named "**`linkerd`**". Note that it may take a few minutes for the provisioning to complete.

```
yourname@ubuntu-vm:~$ cat <<EOF | kind create cluster --name
linkerd --config=-
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
  kubeadmConfigPatches:
  - |
    kind: InitConfiguration
    nodeRegistration:
      kubeletExtraArgs:
        node-labels: "ingress-ready=true"
  extraPortMappings:
  - containerPort: 80
    hostPort: 80
    protocol: TCP
  - containerPort: 443
    hostPort: 443
    protocol: TCP
EOF

Creating cluster "linkerd" ...
✓ Ensuring node image (kindest/node:v1.18.2) 🖼
✓ Preparing nodes 📦
✓ Writing configuration 📜
✓ Starting control-plane 🕹️
✓ Installing CNI 🔌
✓ Installing StorageClass 💾
Set kubectl context to "kind-linkerd"
You can now use your cluster with:

kubectl cluster-info --context kind-linkerd

Thanks for using kind! 😊
```

18. Once **`kind`** finishes creating your new cluster for Linkerd, point **`kubectl`** at the cluster and confirm that the Kubernetes master and KubeDNS are running:

```
yourname@ubuntu-vm:~$ kubectl cluster-info --context kind-linkerd
```

```
Kubernetes master is running at https://127.0.0.1:33499
KubeDNS is running at
https://127.0.0.1:33499/api/v1/namespaces/kube-system/services/ku
be-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl
cluster-info dump'.
```

19. Stop the Linkerd cluster by issuing this command:

```
yourname@ubuntu-vm:~$ docker stop $(docker ps -a -f
name=linkerd-control-plane -q)

8af4c08524df
```

20. Congrats! You have successfully completed the setup of your clusters.

    Please Note: Depending on how long you're planning on taking to do the next labs, the Kind clusters may have to be deleted and re-created. Engineers have noticed that if you wait around 1 week or more, the clusters don't work the way they did in the beginning, as random errors occur. If you see this, you'll most likely have to run `kind delete cluster --name name_of_cluster` and start it again.

    Also, if you plan on shutting down a cluster, the state isn't preserved.

Lab 2.1 - Calculate Average Downtime for Apps

Overview

In this lab, you'll be using an uptime calculator. Also called an availability or service level agreement (SLA) calculator, an uptime calculator actually calculates downtime or unavailability. You enter the percentage of time you need something to be available on average, and it displays how much downtime that translates to.

Calculating downtime is complex, with many factors to consider. In the next chapter we'll take a closer look at those factors. For this lab, we're simplifying things as much as possible. The learning objective of this lab is to illustrate that the number of microservices an app is composed of can directly affect the total downtime for the app. As every application and system is different, the goal of this lab is not to reflect realistic values of how much downtime a typical app might experience.

The uptime calculator we're using in this lab is on the https://uptime.is/ website. This is one of many such calculators publicly available, and we are using it as an example.

For the first part of the exercise, assume you are calculating the expected downtime for a single-service app based on its SLA.

1. Open a web browser and use it to visit the https://uptime.is/ website. This will display an uptime calculator.
2. In the Change SLA level box, enter the value 99 and hit the ENTER or RETURN key. The numbers below show how much time something would be unavailable, on average, every day, week, month, and year if it was only available 99% of the time. An SLA level of 99% corresponds to almost 15 minutes of downtime a day.
3. Change the value in the box to 99.9 and hit the ENTER or RETURN key. The downtime averages are updated, indicating that 99.9% uptime means almost a minute and a half of downtime a day on average.
4. Enter 99.99 in the box and hit ENTER or RETURN. The new downtime average is only 8 seconds a day.

5. Leave the https://uptime.is/ website open—you'll need it again in the next part of this exercise.

For the second part of this exercise, assume you have three microservices, and a cloud-native app can't function without all three of those microservices being available at the same time. Let's see how the availability of each microservice would affect the cloud-native app's availability.

6. Open your computer's calculator or a spreadsheet program.
7. Let's assume each microservice has an SLA of 99%, which is roughly 15 minutes of downtime a day. To calculate the availability in this situation, multiply 99% by 99% by 99%. You would enter this in a calculator like .99 x .99 x .99 or in a spreadsheet like = .99 * .99 *.99
8. The answer will be 0.97 when rounded. That's an SLA of 97%. Go back to the https://uptime.is/ website, enter 99.7% in the box, and hit ENTER or RETURN. It shows that an SLA of 99.7% for each microservice translates to almost 45 minutes of downtime a day for the cloud-native app on average.

Explaining the Calculation
The reason we multiply 99% by 99% by 99% is because we're calculating the probability of all three microservices being available at any given time. Let's choose an arbitrary time—say, 7:00 PM. There's a 99% chance the first microservice will be available at that time. There's also a 99% chance that the second microservice will be available at that time. To find the probability of both being available at the same time, we have to multiply together the odds of each being available.

For the next part of this exercise, assume you have 10 microservices, and a cloud-native app can't function without all 10 of those microservices being available at the same time.

9. Open your computer's calculator or a spreadsheet program.
10. In this scenario, there are 10 microservices that each have an SLA of 99%. To calculate the availability in this situation, multiply ten values of 99% together. You would enter this in a calculator like .99 x .99 x .99 x .99 x .99 x .99 x .99 x .99 x .99 x .99 or in a spreadsheet like = .99 * .99 * .99 * .99 * .99 * .99 * .99 * .99 * .99 * .99 You can also use an exponent function to take .99 to the 10^{th} power.
11. The answer will be 0.904 when rounded. That's an SLA of 90.4%. Go back to the https://uptime.is/ website, enter 90.4% in the box, and hit ENTER or RETURN. So even though each of the microservices has an average downtime of less than 15 minutes a day, together they cause over 2 hours, 18 minutes a day of downtime on average for the app dependent on all of them.

Using Exponential Functions
Scientific calculators have exponential functions that look like x^y. You enter the value for x, then hit the x^y key, then enter the value for y. y is how many times you want to multiply x by itself (x). So if you enter .99 for x and 10 for y, the calculator will multiply .99 ten times.

You can do the same thing in spreadsheet programs. For example, in Microsoft Excel you can enter this formula: =0.99^10

12. Compare the results of the calculations for 99% availability of each app component:

- Single-service app: 15 minutes of downtime a day
- Cloud-native app with three microservices: 45 minutes of downtime a day
- Cloud-native app with 10 microservices: 2 hours, 18 minutes of downtime a day

Don't worry—there are solutions out there to reduce downtime for cloud-native apps. The next chapter of this course will talk about strategies for improving the resilience of cloud-native apps.

Lab 3.1 - Choosing the Best Resilience Strategy

Overview

In this lab, you'll be given a series of simplified app scenarios and asked to choose the best resilience strategy or combination of resilience strategies for each scenario. The resilience strategies you should choose from are:

- Load balancing
  - Round robin
  - Least request
  - Session affinity/sticky sessions
- Timeouts (with or without automatic retries)
- Deadlines
- Circuit breakers

Some scenarios will clearly have one "right" answer, while others might have a few answers that are reasonable. The goal of this lab is not to have you do a long, detailed analysis of complex scenarios or to determine how to implement resilience for each scenario. Instead, you will be applying what you've learned about resilience strategies to handle common situations.

Scenarios

Scenario 1: You're helping to prepare for an existing cloud-native app to be more resilient than it currently is. In the first phase of the project, there will be three instances of each microservice instead of one. The concern is that the backend microservice is written such that it only works correctly if one instance receives every request within a single user session, and this can't be changed in the app code until a later phase of the project. Which one of the resilience strategies would best address this concern immediately?

Scenario 1 Answer: Implementing session affinity/sticky sessions will force all requests for a microservice during one session to be handled by the same microservice instance.

Scenario 2: You've been asked to find the simplest resilience strategy for an app. Which one of the resilience strategies would generally be the simplest?

Scenario 2 Answer: Round robin is usually the simplest. Requests are made in the same order over and over. There's nothing else to keep track of, like how long a request has been waiting for a reply from a microservice instance, how many requests each instance currently has, or which instance a session used before.

Scenario 3: You want to ensure that if a microservice instance stops responding to requests, those requests are reissued to another instance of the microservice without a human needing to intervene. Which resilience strategy would you recommend?

Scenario 3 Answer: There are a few options here. The simplest is to use timeouts and automatic retries. A second option is to use circuit breakers in addition to timeouts and automatic retries. That way, if a particular microservice instance has repeated problems, it won't receive any more requests until the problem is corrected.

Scenario 4: You're helping to select a resilience strategy for a financial app. After a transaction in a payment request is processed, there could be a communication failure involving the reply, and automatically reissuing the request could cause the payment to be duplicated. Which resilience strategy would you recommend to avoid this problem?

Scenario 4 Answer: Using timeouts without automatic retries will help ensure that requests aren't accidentally sent and processed more than once.

Scenario 5: Once your app receives a request from a user, it needs to process that request and return a response to the user within 1.5 seconds. The request starts with microservice A, which uses microservice B, which in turn uses microservice C. The replies then traverse back from C to B and B to A. In this scenario, which resilience strategy would you recommend to ensure each request is handled within 1.5 seconds?

Scenario 5 Answer: Deadlines could be used to set a time limit for the entire chain of communications from A to B to C to B to A.

Scenario 6: You're concerned about any microservice instance being overwhelmed, so you want to make sure that new requests go to the instances with the fewest current requests. Which resilience strategy would you use?

Scenario 6 Answer: The least request algorithm for load balancing is the one that distributes new requests to the instances with the least requests at the time.

Lab 3.2 - Deploy a Demo Application

Overview

In this lab, you'll be installing a demo application, Emoji Vote (emojivoto), in each of the three clusters you created in the first lab. According to Linkerd, "The emojivoto application is a standalone Kubernetes application that uses a mix of gRPC and HTTP calls to allow users to vote on their favorite emojis." This demo app will be used by all the service meshes in the rest of the labs in this course.

Thanks to Buoyant and the Linkerd community for making the Emoji Vote app publicly available. The information in this lab is based on the ["Install the demo app" instructions from Linkerd].

1. Log into your Ubuntu VM with the three Kubernetes clusters you set up in the first chapter.

2. To start the cluster that will contain the Consul service mesh, issue this command:

```
yourname@ubuntu-vm:~$ docker start $(docker ps -a -f
name=consul-control-plane -q)

2e64e6e909e1
```

3. To switch the **kind** context to the Consul cluster, use this command:

```
yourname@ubuntu-vm:~$ kubectl config use-context kind-consul

Switched to context "kind-consul".
```

   Please wait around 1-2 minutes (closer to 1 minute) for the Kind Kubernetes cluster to get up and running

4. To install Emoji Vote for the Consul cluster, run the following command:

```
yourname@ubuntu-vm:~$ curl -sL
https://run.linkerd.io/emojivoto.yml \
  | kubectl apply -f -

namespace/emojivoto created
serviceaccount/emoji created
serviceaccount/voting created
serviceaccount/web created
service/emoji-svc created
service/voting-svc created
service/web-svc created
deployment.apps/emoji created
deployment.apps/vote-bot created
deployment.apps/voting created
deployment.apps/web created
```

5. Stop the Consul cluster by using this command:

```
yourname@ubuntu-vm:~$ docker stop $(docker container ls -a -f
name=consul-control-plane -q)

2e64e6e909e1
```

6. To start the cluster that will contain the Istio service mesh, issue this command:

```
yourname@ubuntu-vm:~$ docker start $(docker ps -a -f
name=istio-control-plane -q)

2d1d09fadf21
```

7. To switch the **kind** context to the Istio cluster, use this command:

```
yourname@ubuntu-vm:~$ kubectl config use-context kind-istio

Switched to context "kind-istio".
```

8. To install Emoji Vote for the Istio cluster, run the following command:

```
yourname@ubuntu-vm:~$ curl -sL
https://run.linkerd.io/emojivoto.yml \
  | kubectl apply -f -

namespace/emojivoto created
serviceaccount/emoji created
```

```
serviceaccount/voting created
serviceaccount/web created
service/emoji-svc created
service/voting-svc created
service/web-svc created
deployment.apps/emoji created
deployment.apps/vote-bot created
deployment.apps/voting created
deployment.apps/web created
```

9. To stop the Istio cluster, run this command:

```
yourname@ubuntu-vm:~$ docker stop $(docker ps -a -f
name=istio-control-plane -q)

2d1d09fadf21
```

10. To start the cluster that will contain the Linkerd service mesh, issue this command:

```
yourname@ubuntu-vm:~$ docker start $(docker ps -a -f
name=linkerd-control-plane -q)

8af4c08524df
```

11. To switch the **kind** context to the Linkerd cluster, use this command:

```
yourname@ubuntu-vm:~$ kubectl config use-context kind-linkerd

Switched to context "kind-linkerd".
```

12. To install Emoji Vote for the Linkerd cluster, run the following command:

```
yourname@ubuntu-vm:~$ curl -sL
https://run.linkerd.io/emojivoto.yml \
  | kubectl apply -f -

namespace/emojivoto created
serviceaccount/emoji created
serviceaccount/voting created
serviceaccount/web created
service/emoji-svc created
service/voting-svc created
service/web-svc created
deployment.apps/emoji created
```

```
deployment.apps/vote-bot created
deployment.apps/voting created
deployment.apps/web created
```

13. To stop the Linkerd cluster, run this command:

```
yourname@ubuntu-vm:~$ docker stop $(docker ps -a -f
name=linkerd-control-plane -q)
```

```
8af4c08524df
```

Lab 4.1 - Deploy an Ingress Controller

Overview

In this lab, you'll be installing an ingress controller for each of the three clusters. It is the typical way of getting traffic from outside of your cluster to apps running within the cluster.

The Nginx Ingress Controller is a popular ingress controller. It has native integrations with all three service meshes we'll be using in subsequent labs—Consul, Istio, and Linkerd—so we will use it as our ingress controller throughout this course.

Consul Cluster

1. Connect to the VM that hosts your Kubernetes clusters.

2. To start the cluster that will contain the Consul service mesh, issue this command:

```
yourname@ubuntu-vm:~$ docker start $(docker ps -a -f
name=consul-control-plane -q)

2e64e6e909e1
```

3. To switch the **`kind`** context to the Consul cluster, use this command:

```
yourname@ubuntu-vm:~$ kubectl config use-context kind-consul

Switched to context "kind-consul".
```

4. To install the Nginx Ingress Controller in the VM, use the following Kubernetes manifests which will create the namespace, install the Operator, install the CRDs, permissions, and install the Ingress.

```
yourname@ubuntu-vm:~$
```

```
kubectl apply -f
https://raw.githubusercontent.com/kubernetes/ingress-nginx/contro
ller-v1.1.1/deploy/static/provider/cloud/deploy.yaml

namespace/ingress-nginx
serviceaccount/ingress-nginx
configmap/ingress-nginx-controller
clusterrole.rbac.authorization.k8s.io/ingress-nginx
clusterrolebinding.rbac.authorization.k8s.io/ingress-nginx
role.rbac.authorization.k8s.io/ingress-nginx unchanged
rolebinding.rbac.authorization.k8s.io/ingress-nginx
service/ingress-nginx-controller-admission
service/ingress-nginx-controller
deployment.apps/ingress-nginx-controller
ingressclass.networking.k8s.io/nginx
validatingwebhookconfiguration.admissionregistration.k8s.io/ingress-ngin
x-admission
serviceaccount/ingress-nginx-admission
clusterrole.rbac.authorization.k8s.io/ingress-nginx-admission
clusterrolebinding.rbac.authorization.k8s.io/ingress-nginx-admission
role.rbac.authorization.k8s.io/ingress-nginx-admission
rolebinding.rbac.authorization.k8s.io/ingress-nginx-admission
job.batch/ingress-nginx-admission-create
job.batch/ingress-nginx-admission-patch
```

5. Make sure that all Kubernetes resources in the **ingress-nginx** namespace are running successfully. The resources will include Pods, Services, Deployments, and ReplicaSets.

```
yourname@ubuntu-vm:~$ kubectl get all -n ingress-nginx

pod/nginxingress-nginx-ingress-6c64f544c6-pcw89   1/1     Running
0          17s
service/nginxingress-nginx-ingress   LoadBalancer   10.96.99.199
<pending>     80:32496/TCP,443:32252/TCP   18s
pod/emissary-ingress-9c45f6447-lfbcx         1/1     Running   0
113s
deployment.apps/nginxingress-nginx-ingress   1/1     1
1           18s
replicaset.apps/nginxingress-nginx-ingress-6c64f544c6   1
1          1        18s
```

At this point, you may be wondering why the Ingress Controller was installed in the **ingress-nginx**. When installing Ingress Controllers, there will be several different use cases. Whether it's for a specific app, a cluster-wide ingress controller for all apps, or even an Ingress Controller that's on a separate worker node listening to all requests that come in. We've decided to go with the cluster-wide Ingress Controller option for isolation purposes. We can still have the Ingress Controller listen in on only specific namespaces, which you'll see next.

You can learn more about the different options per the Nginx Ingress documentation below:

- **Cluster-wide Ingress Controller (default)**. The Ingress Controller handles configuration resources created in any namespace of the cluster. As NGINX is a high-performance load balancer capable of serving many applications at the same time, this option is used by default in our installation manifests and Helm chart.
- **Single-namespace Ingress Controller**. You can configure the Ingress Controller to handle configuration resources only from a particular namespace, which is controlled through the **`-watch-namespace`** command-line argument. This can be useful if you want to use different NGINX Ingress Controllers for different applications, both in terms of isolation and/or operation.
- **Ingress Controller for Specific Ingress Class**. This option works in conjunction with either of the options above. You can further customize which configuration resources are handled by the Ingress Controller by configuring the class of the Ingress Controller and using that class in your configuration resources. See the section [Configuring Ingress Class](#).

6. Next, create a Kubernetes manifest that has the Ingress API which contains the name of the Emoji app service, port, and path to reach the application. You'll notice that there are three annotations - enabling service upstream, Istio, and HashiCorp Consul. These are all for the Service Mesh's that we'll be working with throughout this course. Don't worry about them too much right now as we'll be diving into it more in Chapter 5.

```
yourname@ubuntu-vm:~$ kubectl apply -f - <<EOF
    —
    apiVersion: networking.k8s.io/v1
    kind: Ingress
    metadata:
      namespace: emojivoto
      name: ingress-emojivoto
      annotations:
        ingress.kubernetes.io/rewrite-target: /
        nginx.ingress.kubernetes.io/service-upstream: "true"
        kubernetes.io/ingress.class: istio
        consul.hashicorp.com/connect-inject: “true”
    spec:
      # ingressClassName: nginx
      rules:
```

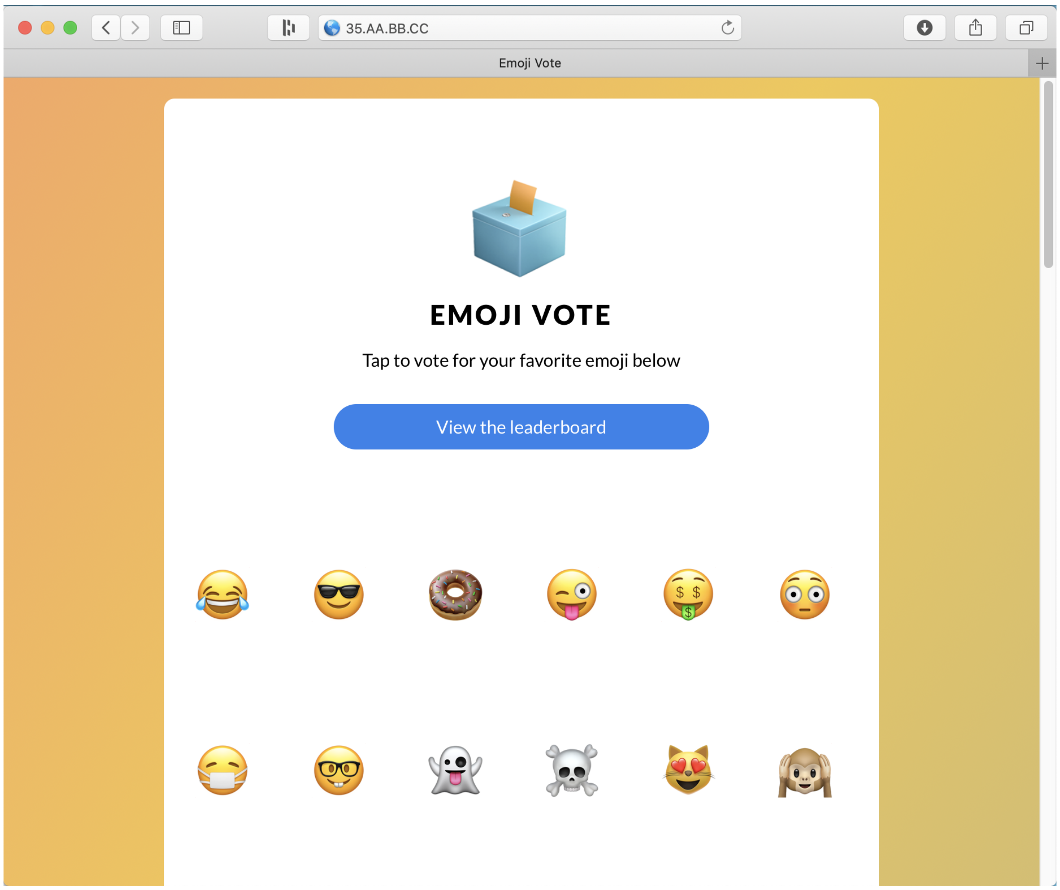
```
    - http:
        paths:
        - pathType: Prefix
          path: "/"
          backend:
            service:
              name: web-svc
              port:
                number: 80

EOF
```

7. Notice that the **`ingressClassName`** is commented out in the above code. If you keep the **`ingressClass`** on with the Istio annotation, you'll receive an error. However, if you apply the Ingress configuration without the **`ingressClassName`** and Istio turned on, and then once it's created, apply the configuration again, it'll work just fine. Run the following code to re-apply the Ingress configuration with the **`ingressClassName`**:

```
yourname@ubuntu-vm:~$ kubectl apply -f - <<EOF
—
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  namespace: emojivoto
  name: ingress-emojivoto
  annotations:
    ingress.kubernetes.io/rewrite-target: /
    nginx.ingress.kubernetes.io/service-upstream: "true"
    kubernetes.io/ingress.class: istio
    consul.hashicorp.com/connect-inject: “true”
spec:
  ingressClassName: nginx
  rules:
    - http:
        paths:
        - pathType: Prefix
          path: "/"
          backend:
            service:
              name: web-svc
              port:
                number: 80
     EOF
```

8. The following command is to forward traffic from the Ingress Controller to your localhost because the Nginx Ingress Controller does not have a load balancer associated with it. The Nginx Ingress Controller listens on port 80, and you're reaching it via your localhost on port 8080.To test that the EmojiApp works, run the following command, open up a web browser, and go to `http://127.0.0.1:8080/`.

```
kubectl port-forward service/ingress-nginx-controller -n
ingress-nginx 8080:80
```



9. Try out the Emoji Vote app. You might notice that some parts of the app are broken—for example, if you click on the donut emoji, you'll get a 404 page. Don't worry, these errors are intentional (and we'll correct them in subsequent labs.)

10. When you are done trying out the demo app, stop the Consul cluster by using this command:

```
yourname@ubuntu-vm:~$ docker stop $(docker container ls -a -f
name=consul-control-plane -q)
```

```
2e64e6e909e1
```

Istio Cluster

11. To start the cluster that will contain the Istio service mesh, issue this command:

```
yourname@ubuntu-vm:~$ docker start $(docker ps -a -f
name=istio-control-plane -q)

2d1d09fadf21
```

12. To switch the **kind** context to the Istio cluster, use this command:

```
yourname@ubuntu-vm:~$ kubectl config use-context kind-istio

Switched to context "kind-istio".
```

13. To install the Nginx Ingress Controller in the VM, use the following Kubernetes Manifests which will create the namespace, install the Operator, install the CRDs, permissions, and install the Ingress:

```
yourname@ubuntu-vm:~$

kubectl apply -f
https://raw.githubusercontent.com/kubernetes/ingress-nginx/contro
ller-v1.1.1/deploy/static/provider/cloud/deploy.yaml

namespace/ingress-nginx created
serviceaccount/ingress-nginx created
configmap/ingress-nginx-controller created
clusterrole.rbac.authorization.k8s.io/ingress-nginx created
clusterrolebinding.rbac.authorization.k8s.io/ingress-nginx created
role.rbac.authorization.k8s.io/ingress-nginx created
rolebinding.rbac.authorization.k8s.io/ingress-nginx created
service/ingress-nginx-controller-admission created
service/ingress-nginx-controller created
deployment.apps/ingress-nginx-controller created
ingressclass.networking.k8s.io/nginx created
validatingwebhookconfiguration.admissionregistration.k8s.io/ingress-ngin
x-admission created
serviceaccount/ingress-nginx-admission created
clusterrole.rbac.authorization.k8s.io/ingress-nginx-admission created
clusterrolebinding.rbac.authorization.k8s.io/ingress-nginx-admission
created
role.rbac.authorization.k8s.io/ingress-nginx-admission created
rolebinding.rbac.authorization.k8s.io/ingress-nginx-admission created
```

```
job.batch/ingress-nginx-admission-create created
job.batch/ingress-nginx-admission-patch created
```

14. Make sure that all Kubernetes resources in the **`ingress-nginx`** namespace are running successfully. The resources will include Pods, Services, Deployments, and ReplicaSets:

```
yourname@ubuntu-vm:~$ kubectl get all -n ingress-nginx
NAME                                                      READY   STATUS
RESTARTS   AGE
pod/ingress-nginx-admission-create-vcspx       0/1     Completed   0
113s
pod/ingress-nginx-admission-patch-nvggr        0/1     Completed   1
113s
pod/ingress-nginx-controller-b66cc4b74-njp2t   1/1     Running     0
113s

NAME                                         TYPE           CLUSTER-IP
EXTERNAL-IP   PORT(S)                      AGE
service/ingress-nginx-controller             LoadBalancer
10.96.182.254   <pending>     80:30635/TCP,443:31499/TCP   113s
service/ingress-nginx-controller-admission   ClusterIP      10.96.9.72
<none>        443/TCP                      113s

NAME                                       READY   UP-TO-DATE
AVAILABLE   AGE
deployment.apps/ingress-nginx-controller   1/1     1            1
113s

NAME                                                DESIRED   CURRENT
READY   AGE
replicaset.apps/ingress-nginx-controller-b66cc4b74   1         1
1       113s

NAME                                       COMPLETIONS   DURATION
AGE
job.batch/ingress-nginx-admission-create   1/1           14s
113s
job.batch/ingress-nginx-admission-patch    1/1           15s
113s
```

15. Next, create a Kubernetes Manifest that has the Ingress API which contains the name of the Emoji app service, port, and path to reach the application. You'll notice that there are

three annotations - enabling service upstream, Istio, and HashiCorp Consul. These are all for the Service Mesh's that we'll be working with throughout this course. Don't worry about them too much right now as we'll be diving into it more in Chapter 5.

```
yourname@ubuntu-vm:~$ kubectl apply -f - <<EOF
—
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  namespace: emojivoto
  name: ingress-emojivoto
  annotations:
    ingress.kubernetes.io/rewrite-target: /
    nginx.ingress.kubernetes.io/service-upstream: "true"
    kubernetes.io/ingress.class: istio
    consul.hashicorp.com/connect-inject: “true”
spec:
  # ingressClassName: nginx
  rules:
    - http:
        paths:
        - pathType: Prefix
          path: "/"
          backend:
            service:
              name: web-svc
              port:
                number: 80

EOF
```

16. Notice that the ingressClassName is commented out in the above code. If you keep the ingressClass on with the Istio annotation, you'll receive an error. However, if you apply the Ingress configuration without the ingressClassName and Istio turned on, and then once it's created, apply the configuration again, it'll work just fine. Run the following code to re-apply the Ingress configuration with the ingressClassName

```
yourname@ubuntu-vm:~$ kubectl apply -f - <<EOF
—
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  namespace: emojivoto
  name: ingress-emojivoto
```

```
  annotations:
    ingress.kubernetes.io/rewrite-target: /
    nginx.ingress.kubernetes.io/service-upstream: "true"
    kubernetes.io/ingress.class: istio
    consul.hashicorp.com/connect-inject: “true”
spec:
  ingressClassName: nginx
  rules:
    - http:
        paths:
        - pathType: Prefix
          path: "/"
          backend:
            service:
              name: web-svc
              port:
                number: 80
     EOF
```

17. The following command is to forward traffic from the Ingress Controller to your localhost because the Nginx Ingress Controller does not have a load balancer associated with it. The Nginx Ingress Controller listens on port 80, and you’re reaching it via your localhost on port 8080.To test that the EmojiApp works, run the following command, open up a web browser, and go to http://127.0.0.1:8080/

```
kubectl port-forward service/ingress-nginx-controller -n
ingress-nginx 8080:80
```

18. When you are done with the Emoji Vote application, stop the Istio cluster by running this command:

```
yourname@ubuntu-vm:~$ docker stop $(docker ps -a -f
name=istio-control-plane -q)

2d1d09fadf21
```

Linkerd Cluster

19. To start the cluster that will contain the Linkerd service mesh, issue this command:

```
yourname@ubuntu-vm:~$ docker start $(docker ps -a -f
name=linkerd-control-plane -q)

8af4c08524df
```

20. To switch the **kind** context to the Linkerd cluster, use this command:

```
yourname@ubuntu-vm:~$ kubectl config use-context kind-linkerd

Switched to context "kind-linkerd".
```

21. To install the Nginx Ingress Controller in the VM, use the following Kubernetes Manifests which will create the namespace, install the Operator, install the CRDs, permissions, and install the Ingress.

```
yourname@ubuntu-vm:~$

kubectl apply -f
https://raw.githubusercontent.com/kubernetes/ingress-nginx/contro
ller-v1.1.1/deploy/static/provider/cloud/deploy.yaml
```

```
namespace/ingress-nginx created
serviceaccount/ingress-nginx created
configmap/ingress-nginx-controller created
clusterrole.rbac.authorization.k8s.io/ingress-nginx created
clusterrolebinding.rbac.authorization.k8s.io/ingress-nginx created
role.rbac.authorization.k8s.io/ingress-nginx created
rolebinding.rbac.authorization.k8s.io/ingress-nginx created
service/ingress-nginx-controller-admission created
service/ingress-nginx-controller created
deployment.apps/ingress-nginx-controller created
ingressclass.networking.k8s.io/nginx created
validatingwebhookconfiguration.admissionregistration.k8s.io/ingress-nginx-admi
ssion created
serviceaccount/ingress-nginx-admission created
clusterrole.rbac.authorization.k8s.io/ingress-nginx-admission created
clusterrolebinding.rbac.authorization.k8s.io/ingress-nginx-admission created
role.rbac.authorization.k8s.io/ingress-nginx-admission created
rolebinding.rbac.authorization.k8s.io/ingress-nginx-admission created
job.batch/ingress-nginx-admission-create created
job.batch/ingress-nginx-admission-patch created
```

22. Make sure that all Kubernetes resources in the ingress-nginx namespace are running successfully. The resources will include Pods, Services, Deployments, and ReplicaSets

```
kubectl get all -n ingress-nginx

NAME                                                      READY   STATUS
RESTARTS   AGE
```

```
pod/ingress-nginx-admission-create-r7gvr       0/1     Completed   0
63s
pod/ingress-nginx-admission-patch-6prps        0/1     Completed   0
63s
pod/ingress-nginx-controller-b66cc4b74-pp4kh   1/1     Running     0
64s

NAME                                          TYPE          CLUSTER-IP
EXTERNAL-IP   PORT(S)                      AGE
service/ingress-nginx-controller              LoadBalancer
10.96.142.246   <pending>     80:32495/TCP,443:30997/TCP   64s
service/ingress-nginx-controller-admission   ClusterIP
10.96.228.215   <none>        443/TCP                      64s

NAME                                       READY   UP-TO-DATE
AVAILABLE   AGE
deployment.apps/ingress-nginx-controller   1/1     1            1
64s

NAME                                                 DESIRED   CURRENT
READY   AGE
replicaset.apps/ingress-nginx-controller-b66cc4b74   1         1
1       64s

NAME                                       COMPLETIONS   DURATION   AGE
job.batch/ingress-nginx-admission-create   1/1           7s         63s
job.batch/ingress-nginx-admission-patch    1/1           7s         63s
```

23. Next, create a Kubernetes Manifest that has the Ingress API which contains the name of the Emoji app service, port, and path to reach the application. You'll notice that there are three annotations - enabling service upstream, Istio, and HashiCorp Consul. These are all for the Service Mesh's that we'll be working with throughout this course. Don't worry about them too much right now as we'll be diving into it more in Chapter 5.

```
yourname@ubuntu-vm:~$ kubectl apply -f - <<EOF
      –
      apiVersion: networking.k8s.io/v1
      kind: Ingress
      metadata:
        namespace: emojivoto
        name: ingress-emojivoto
        annotations:
          ingress.kubernetes.io/rewrite-target: /
          nginx.ingress.kubernetes.io/service-upstream: "true"
          kubernetes.io/ingress.class: istio
          consul.hashicorp.com/connect-inject: “true”
      spec:
```

```
    # ingressClassName: nginx
    rules:
      - http:
          paths:
          - pathType: Prefix
            path: "/"
            backend:
              service:
                name: web-svc
                port:
                  number: 80

  EOF
```

24. Notice that the ingressClassName is commented out in the above code. If you keep the **`ingressClass`** on with the Istio annotation, you'll receive an error. However, if you apply the Ingress configuration without the **`ingressClassName`** and Istio turned on, and then once it's created, apply the configuration again, it'll work just fine. Run the following code to re-apply the Ingress configuration with the **`ingressClassName`**.

```
yourname@ubuntu-vm:~$ kubectl apply -f - <<EOF
–
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  namespace: emojivoto
  name: ingress-emojivoto
  annotations:
    ingress.kubernetes.io/rewrite-target: /
    nginx.ingress.kubernetes.io/service-upstream: "true"
    kubernetes.io/ingress.class: istio
    consul.hashicorp.com/connect-inject: “true”
spec:
  ingressClassName: nginx
  rules:
    - http:
        paths:
        - pathType: Prefix
          path: "/"
          backend:
            service:
              name: web-svc
              port:
                number: 80
```

```
EOF
```

25. The following command is to forward traffic from the Ingress Controller to your localhost because the Nginx Ingress Controller does not have a load balancer associated with it. The Nginx Ingress Controller listens on port 80, and you're reaching it via your localhost on port 8080.To test that the EmojiApp works, run the following command, open up a web browser, and go to http://127.0.0.1:8080/

```
kubectl port-forward service/ingress-nginx-controller -n
ingress-nginx 8080:80
```

26. To stop the Linkerd cluster, run this command:

```
yourname@ubuntu-vm:~$ docker stop $(docker ps -a -f
name=linkerd-control-plane -q)

8af4c08524df
```

Lab 4.2 - Secure Ingress Traffic

Overview

In this lab, you'll be securing ingress traffic from the client to Ambassador for each of the three clusters by using TLS termination. Configuring TLS termination can be complicated because of the behavior of different proxies and load balancers that pass traffic to the Ingress controller, the need to negotiate TLS versions with untrusted clients, and other reasons.

Consul Cluster

1. Connect to the VM that hosts your Kubernetes clusters.

2. To start the cluster that will contain the Consul service mesh, issue this command:

```
yourname@ubuntu-vm:~$ docker start $(docker ps -a -f
name=consul-control-plane -q)

2e64e6e909e1
```

3. To switch the **`kind`** context to the Consul cluster, use this command:

```
yourname@ubuntu-vm:~$ kubectl config use-context kind-consul

Switched to context "kind-consul".
```

4. Forward traffic for the Emoji application locally to ensure that the app is up and running and can receive requests

```
yourname@ubuntu-vm:~$ kubectl port-forward
service/ingress-nginx-controller -n ingress-nginx 8080:80
```

5. Verify the routing requests by issuing the command below. A response code of 200 indicates that Nginx Ingress is properly routing traffic to the demo app. If you receive a different code, wait a few minutes and issue the command again.

```
yourname@ubuntu-vm:~$ curl -s -o /dev/null -w "Upstream Response
Code: %{http_code}\n" \
  http://localhost:8080

Upstream Response Code: 200
```

6. One of the most popular ways to secure Ingress Nginx traffic is by using **`cert-manager`**. Cert-manager is an add-on that allows you to automate the management and issuing of TLS certificates from various resources and different certificate providers. It does things like automatically ensure that the certificates you're using are up to date and renewed. In this case and for the purposes of these labs, we'll use LetsEncrypt, which is a popular and open-source way to create and utilize certificates. First we will install cert-manager, which will create everything that is needed for cert-manager to run on Kubernetes including RBAC permissions and CRDs.

```
yourname@ubuntu-vm:~$ kubectl apply -f
https://github.com/cert-manager/cert-manager/releases/download/v1.9.1/
cert-manager.yaml

namespace/cert-manager created
customresourcedefinition.apiextensions.k8s.io/certificaterequests.cert-m
anager.io created
customresourcedefinition.apiextensions.k8s.io/certificates.cert-manager.
io created
customresourcedefinition.apiextensions.k8s.io/challenges.acme.cert-manag
er.io created
customresourcedefinition.apiextensions.k8s.io/clusterissuers.cert-manage
r.io created
customresourcedefinition.apiextensions.k8s.io/issuers.cert-manager.io
created
customresourcedefinition.apiextensions.k8s.io/orders.acme.cert-manager.i
o created
serviceaccount/cert-manager-cainjector created
serviceaccount/cert-manager created
serviceaccount/cert-manager-webhook created
configmap/cert-manager-webhook created
clusterrole.rbac.authorization.k8s.io/cert-manager-cainjector created
clusterrole.rbac.authorization.k8s.io/cert-manager-controller-issuers
created
```

7. Next, you will need to create an Issuer. The Issuer is what generates the certificate and allows you to connect the cert for secure communication to your Nginx Ingress Controller. Essentially, it “issues” a certificate to your Ingress Controller so you can access your app over HTTPS. Notice how it’s using an example “acme” server. The reason why is because this is a lab environment in a Kind cluster. If you were in a production environment, it would be production-level entries for server, email, etc.

```
yourname@ubuntu-vm:~$ kubectl apply -f - <<EOF
---
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
 name: letsencrypt-staging
 namespace: cert-manager
spec:
 acme:
   server: https://acme-staging-v02.api.letsencrypt.org/directory
   email: example@test.com
   privateKeySecretRef:
     name: letsencrypt-staging
   solvers:
   - http01:
       ingress:
         class:  nginx
EOF
```

8. Next, you will need to update your Ingress Controller configuration to ensure that the issuer is connected to the Ingress Controller and a Kubernetes secret is associated with the cert. The changes between the Ingress configuration below and the Ingress configuration from the previous lab are highlighted in orange so you can see the differences. You’ll notice that the host is a sample “echo1.example.com”. The reason why is because you’re deploying this on a local KinD cluster. If this was production, it would be the path/URL to the production environment/website.

```
yourname@ubuntu-vm:~$ kubectl apply -f - <<EOF
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  namespace: emojivoto
  name: ingress-emojivoto
  annotations:
    ingress.kubernetes.io/rewrite-target: /
    nginx.ingress.kubernetes.io/service-upstream: "true"
    kubernetes.io/ingress.class: istio
    consul.hashicorp.com/connect-inject: “true”
    cert-manager.io/issuer: "letsencrypt-staging"
```

```
spec:
  tls:
  - hosts:
    - echo1.example.com
    secretName: tlstest
  ingressClassName: nginx
  rules:
    - http:
        paths:
        - pathType: Prefix
          path: "/"
          backend:
            service:
              name: web-svc
              port:
                number: 80
EOF
```

9. Now that the Ingress Controller is configured, you can forward traffic over port 443.

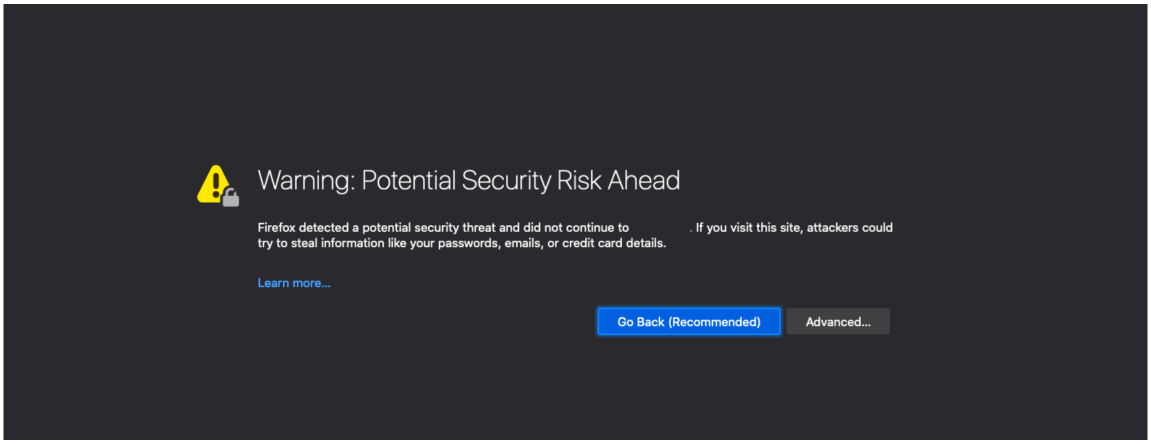
```
yourname@ubuntu-vm:~$ kubectl port-forward
service/ingress-nginx-controller -n ingress-nginx 8080:443
```

10. Nginx Ingress is now configured to use the self-signed TLS certificate via cert-manager to establish and terminate TLS connections for any incoming host. This allows you to make requests to the VM's IP address or **`localhost`** over HTTPS. The Nginx Ingress Controller is now listening on port 443 for TLS connections instead of port 80. It is also automatically redirecting cleartext requests to HTTPS. Enter https://127.0.0.1:8080/ in a web browser and you should now see the emoji app pop up.

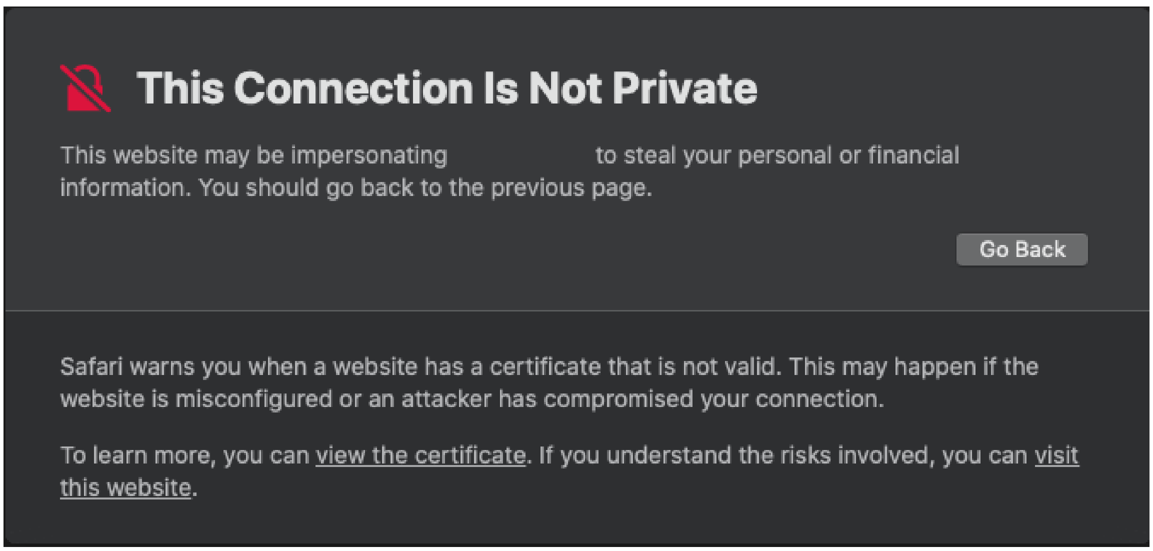
11. Since you are using a self-signed certificate, you will probably get a warning page about an untrusted certificate when trying to access the application. This is expected and can be bypassed. Below are examples of the warning you may receive from a few browsers.

    If you are unable or unwilling to bypass the warning pages, you may skip this step and the next step. The warning page indicates that you are being redirected from HTTP to HTTPS and that your self-signed certificate is being encountered, and the steps you would skip are illustrating that in a second way.

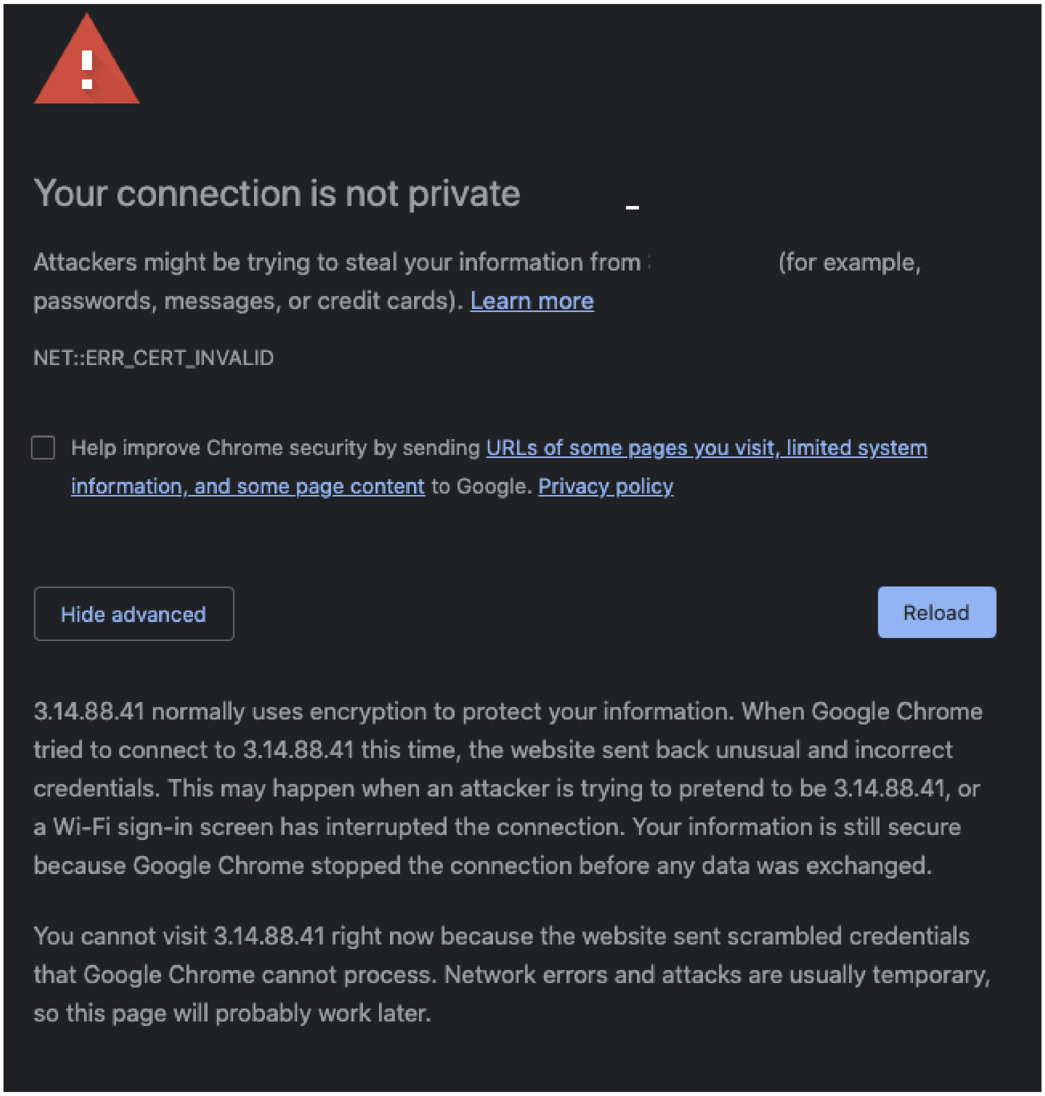
    Note: If you are using Google Chrome for your browser, you will need to type **`thisisunsafe`** on the page and hit Enter or Return to bypass the security warning.



Firefox

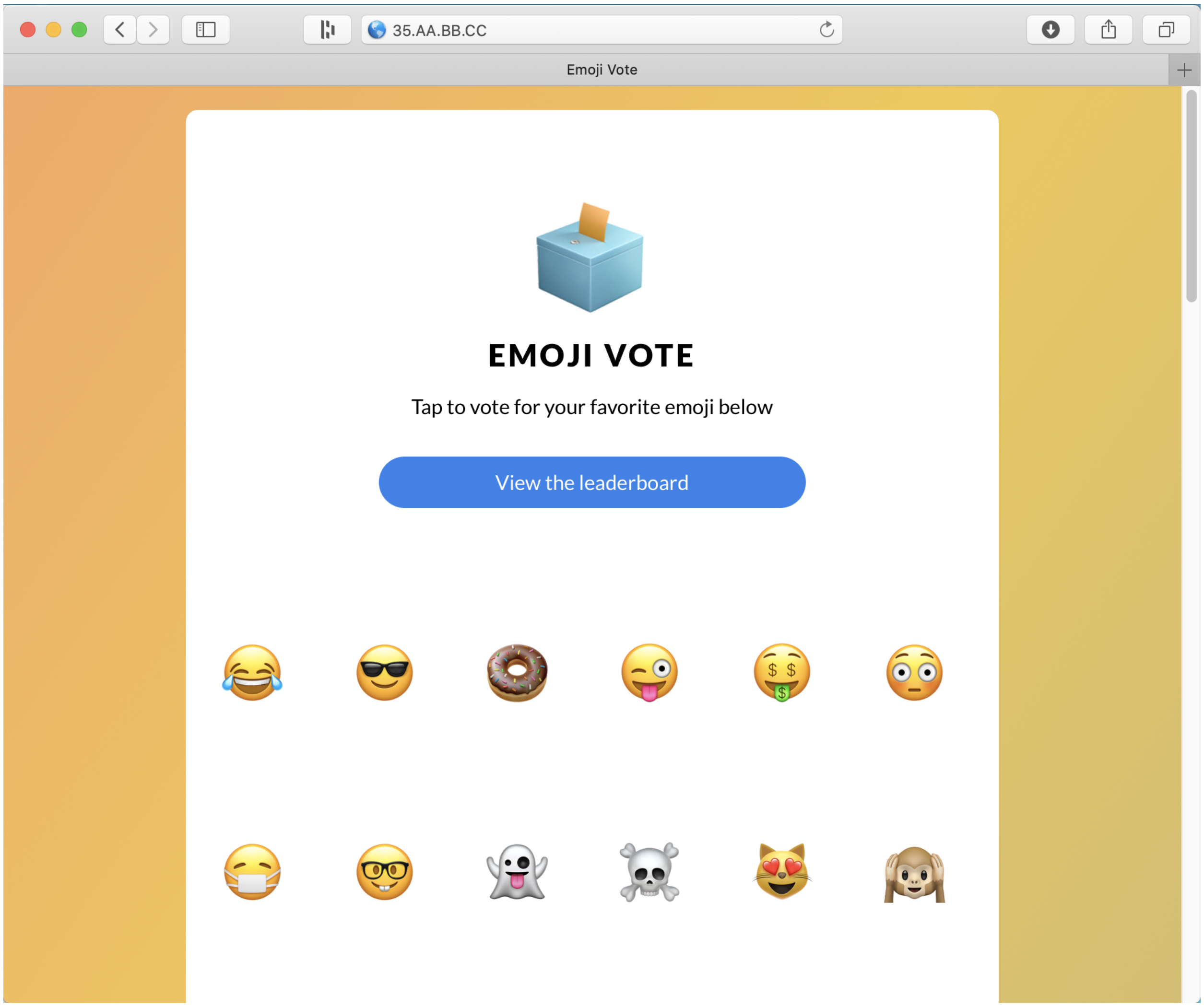


Safari



Chrome

12. After bypassing the security warning, you are now accessing the application over an encrypted connection! Try out the app. It should function just like it did in the previous lab, including giving you a 404 error when you click on the donut emoji.



13. When you are done with the app, stop the Consul cluster by using this command:

```
yourname@ubuntu-vm:~$ docker stop $(docker container ls -a -f
name=consul-control-plane -q)

2e64e6e909e1
```

Istio Cluster

14. To start the cluster that will contain the Istio service mesh, issue this command:

```
yourname@ubuntu-vm:~$ docker start $(docker ps -a -f
name=istio-control-plane -q)
```

```
2d1d09fadf21
```

15. To switch the **kind** context to the Istio cluster, use this command:

```
yourname@ubuntu-vm:~$ kubectl config use-context kind-istio

Switched to context "kind-istio".
```

16. Forward traffic for the Emoji application locally to ensure that the app is up and running and can receive requests

```
yourname@ubuntu-vm:~$ kubectl port-forward
service/emissary-ingress -n emissary 8080:80
```

17. Install Cert manager

```
yourname@ubuntu-vm:~$ kubectl apply -f
https://github.com/cert-manager/cert-manager/releases/download/v1.9.1/
cert-manager.yaml
```

```
namespace/cert-manager created
customresourcedefinition.apiextensions.k8s.io/certificaterequests.cert-m
anager.io created
customresourcedefinition.apiextensions.k8s.io/certificates.cert-manager.
io created
customresourcedefinition.apiextensions.k8s.io/challenges.acme.cert-manag
er.io created
customresourcedefinition.apiextensions.k8s.io/clusterissuers.cert-manage
r.io created
customresourcedefinition.apiextensions.k8s.io/issuers.cert-manager.io
created
customresourcedefinition.apiextensions.k8s.io/orders.acme.cert-manager.i
o created
serviceaccount/cert-manager-cainjector created
serviceaccount/cert-manager created
serviceaccount/cert-manager-webhook created
configmap/cert-manager-webhook created
clusterrole.rbac.authorization.k8s.io/cert-manager-cainjector created
clusterrole.rbac.authorization.k8s.io/cert-manager-controller-issuers
created
```

18. Create the Issuer

```
yourname@ubuntu-vm:~$ kubectl apply -f - <<EOF
---
```

```
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
 name: letsencrypt-staging
 namespace: cert-manager
spec:
 acme:
   server: https://acme-staging-v02.api.letsencrypt.org/directory
   email: example@test.com
   privateKeySecretRef:
     name: letsencrypt-staging
   solvers:
   - http01:
       ingress:
         class:  nginx
EOF
```

19. Update the ingress controller to use the cert issued by cert-manager

```
yourname@ubuntu-vm:~$ kubectl apply -f - <<EOF
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  namespace: emojivoto
  name: ingress-emojivoto
  annotations:
    ingress.kubernetes.io/rewrite-target: /
    nginx.ingress.kubernetes.io/service-upstream: "true"
    kubernetes.io/ingress.class: istio
    consul.hashicorp.com/connect-inject: “true”
    cert-manager.io/issuer: "letsencrypt-staging"
spec:
  tls:
  - hosts:
    - echo1.example.com
    secretName: tlstest
  ingressClassName: nginx
  rules:
    - http:
        paths:
        - pathType: Prefix
          path: "/"
          backend:
            service:
              name: web-svc
              port:
                number: 80
EOF
```

20. Now that the Ingress Controller is configured, you can forward traffic over port 443.

```
yourname@ubuntu-vm:~$ kubectl port-forward
service/ingress-nginx-controller -n ingress-nginx 8080:443
```

21. Nginx Ingress is now configured to use the self-signed TLS certificate via cert-manager to establish and terminate TLS connections for any incoming host. This allows you to make requests to the VM's IP address or **`localhost`** over HTTPS. The Nginx Ingress Controller Ambassador is now listening on port 443 for TLS connections instead of port 80. It is also automatically redirecting cleartext requests to HTTPS. Enter https://127.0.0.1:8080/ in a web browser and you should now see the emoji app pop up.

22. Since you are using a self-signed certificate, you will probably get a warning page about an untrusted certificate when trying to access the application. This is expected and can be bypassed.

    If you are unable or unwilling to bypass the warning pages, you may skip this step and the next step. The warning page indicates that you are being redirected from HTTP to HTTPS and that your self-signed certificate is being encountered, and the steps you would skip are illustrating that in a second way.

    Note: If you are using Google Chrome for your browser, you will need to type **`thisisunsafe`** on the page and hit Enter or Return to bypass the security warning.

23. After bypassing the security warning, you are now accessing the application over an encrypted connection! Try out the app. It should function just like it did in the previous lab, including giving you a 404 error when you click on the donut emoji.

24. When you are done with the app, stop the Istio cluster by running this command:

```
yourname@ubuntu-vm:~$ docker stop $(docker ps -a -f
name=istio-control-plane -q)

2d1d09fadf21
```

Linkerd Cluster

25. To start the cluster that will contain the Linkerd service mesh, issue this command:

```
yourname@ubuntu-vm:~$ docker start $(docker ps -a -f
name=linkerd-control-plane -q)

8af4c08524df
```

26. To switch the **`kind`** context to the Linkerd cluster, use this command:

```
yourname@ubuntu-vm:~$ kubectl config use-context kind-linkerd

Switched to context "kind-linkerd".
```

27. Forward traffic for the Emoji application locally to ensure that the app is up and running and can receive requests

```
yourname@ubuntu-vm:~$ kubectl port-forward
service/emissary-ingress -n emissary 8080:80
```

28. Install Cert manager

```
yourname@ubuntu-vm:~$ kubectl apply -f
https://github.com/cert-manager/cert-manager/releases/download/v1.9.1/
cert-manager.yaml
```

```
namespace/cert-manager created
customresourcedefinition.apiextensions.k8s.io/certificaterequests.cert-m
anager.io created
customresourcedefinition.apiextensions.k8s.io/certificates.cert-manager.
io created
customresourcedefinition.apiextensions.k8s.io/challenges.acme.cert-manag
er.io created
customresourcedefinition.apiextensions.k8s.io/clusterissuers.cert-manage
r.io created
customresourcedefinition.apiextensions.k8s.io/issuers.cert-manager.io
created
customresourcedefinition.apiextensions.k8s.io/orders.acme.cert-manager.i
o created
serviceaccount/cert-manager-cainjector created
serviceaccount/cert-manager created
serviceaccount/cert-manager-webhook created
configmap/cert-manager-webhook created
clusterrole.rbac.authorization.k8s.io/cert-manager-cainjector created
clusterrole.rbac.authorization.k8s.io/cert-manager-controller-issuers
created
```

29. Create the Issuer

```
yourname@ubuntu-vm:~$ kubectl apply -f - <<EOF
---
apiVersion: cert-manager.io/v1
```

```
kind: ClusterIssuer
metadata:
 name: letsencrypt-staging
 namespace: cert-manager
spec:
 acme:
   server: https://acme-staging-v02.api.letsencrypt.org/directory
   email: example@test.com
   privateKeySecretRef:
     name: letsencrypt-staging
   solvers:
   - http01:
       ingress:
         class:  nginx
EOF
```

30. Update the ingress controller to use the cert issued by cert-manager

```
yourname@ubuntu-vm:~$ kubectl apply -f - <<EOF
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  namespace: emojivoto
  name: ingress-emojivoto
  annotations:
    ingress.kubernetes.io/rewrite-target: /
    nginx.ingress.kubernetes.io/service-upstream: "true"
    kubernetes.io/ingress.class: istio
    consul.hashicorp.com/connect-inject: “true”
    cert-manager.io/issuer: "letsencrypt-staging"
spec:
  tls:
  - hosts:
    - echo1.example.com
    secretName: tlstest
  ingressClassName: nginx
  rules:
    - http:
        paths:
        - pathType: Prefix
          path: "/"
          backend:
            service:
              name: web-svc
              port:
                number: 80
EOF
```

31. Now that the Ingress Controller is configured, you can forward traffic over port 443.

```
yourname@ubuntu-vm:~$ kubectl port-forward
service/ingress-nginx-controller -n ingress-nginx 8080:443
```

32. Nginx Ingress is now configured to use the self-signed TLS certificate via cert-manager to establish and terminate TLS connections for any incoming host. This allows you to make requests to the VM's IP address or **`localhost`** over HTTPS. The Nginx Ingress Controller Ambassador is now listening on port 443 for TLS connections instead of port 80. It is also automatically redirecting cleartext requests to HTTPS. Enter https://127.0.0.1:8080/ in a web browser and you should now see the emoji app pop up

33. Since you are using a self-signed certificate, you will probably get a warning page about an untrusted certificate when trying to access the application. This is expected and can be bypassed.

    If you are unable or unwilling to bypass the warning pages, you may skip this step and the next step. The warning page indicates that you are being redirected from HTTP to HTTPS and that your self-signed certificate is being encountered, and the steps you would skip are illustrating that in a second way.

    Note: If you are using Google Chrome for your browser, you will need to type **`thisisunsafe`** on the page and hit Enter or Return to bypass the security warning.

34. After bypassing the security warning, you are now accessing the application over an encrypted connection! Try out the app. It should function just like it did in the previous lab, including giving you a 404 error when you click on the donut emoji.

35. When you are done with the app, stop the Linkerd cluster by running this command:

```
yourname@ubuntu-vm:~$ docker stop $(docker ps -a -f
name=linkerd-control-plane -q)

8af4c08524df
```

THE LINUX FOUNDATION | Training & Certification

Lab 5.1 - Install a Linkerd Service Mesh

Overview

In this lab, you'll be installing a service mesh using Linkerd.

1. Connect to the VM that hosts your Kubernetes clusters.

2. To start the cluster that will contain the Linkerd service mesh, issue this command:

```
yourname@ubuntu-vm:~$ docker start $(docker ps -a -f
name=linkerd-control-plane -q)

8af4c08524df
```

3. To switch the **kind** context to the Linkerd cluster, use this command:

```
yourname@ubuntu-vm:~$ kubectl config use-context kind-linkerd

Switched to context "kind-linkerd".
```

4. Download the **linkerd** command line interface (CLI):

```
yourname@ubuntu-vm:~$ curl -sL https://run.linkerd.io/install |
sh

Downloading linkerd2-cli-stable-2.8.1-linux...
 % Total    % Received % Xferd  Average Speed   Time    Time
Time  Current
                                Dload  Upload   Total   Spent
Left  Speed
```

```
100   644  100   644    0     0   4380      0 --:--:-- --:--:--
--:--:--  4380
100 37.0M  100 37.0M    0     0  17.5M      0  0:00:02  0:00:02
--:--:-- 22.8M
Download complete!

Validating checksum...
Checksum valid.

Linkerd stable-2.8.1 was successfully installed 🎉

Add the linkerd CLI to your path with:
 export PATH=$PATH:/home/yourname/.linkerd2/bin

Now run:
 linkerd check --pre                         # validate that Linkerd
can be installed
 linkerd install | kubectl apply -f -        # install the control
plane into the 'linkerd' namespace
 linkerd check                               # validate everything
worked!
 linkerd dashboard                           # launch the dashboard

Looking for more? Visit https://linkerd.io/2/next-steps
```

5. Add the linkerd CLI to your current **PATH**:

```
yourname@ubuntu-vm:~$ export PATH=$PATH:$HOME/.linkerd2/bin

[no output if the command succeeds]
```

6. You should also add the linkerd CLI to the **$PATH** variable for your shell with the command below (assuming you are using **bash** for your shell). This will permanently add it to your **PATH**. Without doing this once, if you log out and log back in, your **PATH** will be replaced and your **linkerd** commands will not work correctly until you manually update your **PATH** as you did in the previous step.

```
echo "export PATH=$PATH:$HOME/.linkerd2/bin" >> ~/.bashrc

[no output if the command succeeds]
```

7. Verify the download and that your cluster is ready to install Linkerd with the command below. If any of the status checks fail, investigate and address the issue, then run this command again to confirm the status checks pass before continuing to the next step.

```
yourname@ubuntu-vm:~$ linkerd check --pre

kubernetes-api
--------------
√ can initialize the client
√ can query the Kubernetes API

kubernetes-version
------------------
√ is running the minimum Kubernetes API version
√ is running the minimum kubectl version

pre-kubernetes-setup
--------------------
√ control plane namespace does not already exist
√ can create non-namespaced resources
√ can create ServiceAccounts
√ can create Services
√ can create Deployments
√ can create CronJobs
√ can create ConfigMaps
√ can create Secrets
√ can read Secrets
√ can read extension-apiserver-authentication configmap
√ no clock skew detected

pre-kubernetes-capability
-------------------------
√ has NET_ADMIN capability
√ has NET_RAW capability

linkerd-version
---------------
√ can determine the latest version
√ cli is up-to-date

Status check results are √
```

8. Install Linkerd. Make sure that the single quotes you use are plain quotes and not curly quotes.

```
yourname@ubuntu-vm:~$ linkerd install \
  | sed 's|-enforced-host=.*|-enforced-host=|' \
  | kubectl apply -f -
```

```
namespace/linkerd created
clusterrole.rbac.authorization.k8s.io/linkerd-linkerd-identity
created
clusterrolebinding.rbac.authorization.k8s.io/linkerd-linkerd-iden
tity created
[additional output omitted]
serviceaccount/linkerd-grafana created
configmap/linkerd-grafana-config created
service/linkerd-grafana created
deployment.apps/linkerd-grafana created
```

9. Wait for Linkerd to start before continuing. This command will check to see if everything is ready, and if not will keep you updates on the status.

```
yourname@ubuntu-vm:~$ linkerd check

kubernetes-api
--------------
√ can initialize the client
√ can query the Kubernetes API

kubernetes-version
------------------
√ is running the minimum Kubernetes API version
√ is running the minimum kubectl version

linkerd-existence
-----------------
√ 'linkerd-config' config map exists
√ heartbeat ServiceAccount exist
√ control plane replica sets are ready
√ no unschedulable pods
√ controller pod is running
√ can initialize the client
√ can query the control plane API

linkerd-config
--------------
√ control plane Namespace exists
√ control plane ClusterRoles exist
√ control plane ClusterRoleBindings exist
√ control plane ServiceAccounts exist
√ control plane CustomResourceDefinitions exist
√ control plane MutatingWebhookConfigurations exist
```

```
√ control plane ValidatingWebhookConfigurations exist
√ control plane PodSecurityPolicies exist

linkerd-identity
----------------
√ certificate config is valid
√ trust anchors are using supported crypto algorithm
√ trust anchors are within their validity period
√ trust anchors are valid for at least 60 days
√ issuer cert is using supported crypto algorithm
√ issuer cert is within its validity period
√ issuer cert is valid for at least 60 days
√ issuer cert is issued by the trust anchor

linkerd-api
-----------
√ control plane pods are ready
√ control plane self-check
√ [kubernetes] control plane can talk to Kubernetes
√ [prometheus] control plane can talk to Prometheus
√ tap api service is running

linkerd-version
---------------
√ can determine the latest version
√ cli is up-to-date

control-plane-version
---------------------
√ control plane is up-to-date
√ control plane and cli versions match

linkerd-addons
--------------
√ 'linkerd-config-addons' config map exists

linkerd-grafana
---------------
√ grafana add-on service account exists
√ grafana add-on config map exists
√ grafana pod is running

Status check results are √
```

10. Install the Linkerd dashboard so you can get a visual of everything that's happening in your environment:

```
yourname@ubuntu-vm:~$ linkerd viz install | kubectl apply -f -
```

11. Check the Linkerd viz environment to confirm it deployed as expected.

```
yourname@ubuntu-vm:~$ linkerd viz check

√ linkerd-viz Namespace exists
√ linkerd-viz ClusterRoles exist
√ linkerd-viz ClusterRoleBindings exist
√ tap API server has valid cert
√ tap API server cert is valid for at least 60 days
√ tap API service is running
√ linkerd-viz pods are injected
√ viz extension pods are running
√ viz extension proxies are healthy
√ viz extension proxies are up-to-date
√ viz extension proxies and cli versions match
√ prometheus is installed and configured correctly
√ can initialize the client
√ viz extension self-check

Status check results are √
```

12. With Linkerd, the proxy is lightweight and transparently intercepts traffic, so unlike the Envoy-based service mesh implementations, you will actually inject Linkerd into Nginx Ingress via sidecar proxy to the mesh by updating the Ingress manifest to include the Istio injector annotation. Below is the manifest to run and you'll see the update to the annotation in orange:

```
yourname@ubuntu-vm:~$

kubectl apply -f - <<EOF
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  namespace: emojivoto
  name: ingress-emojivoto
  annotations:
    ingress.kubernetes.io/rewrite-target: /
```

```
    nginx.ingress.kubernetes.io/service-upstream: "true"
    linkerd.io/inject: enabled
    kubernetes.io/ingress.class: istio
    consul.hashicorp.com/connect-inject: “true”
spec:
  ingressClassName: nginx
  rules:
    - http:
        paths:
        - pathType: Prefix
          path: "/"
          backend:
            service:
              name: web-svc
              port:
                number: 80
EOF
```

13. After installing Linkerd and updating the Nginx Ingress controller, you need to add the demo application you installed to the mesh by using **`linkerd`** and **`kubectl`**:

```
yourname@ubuntu-vm:~$ curl -sL
https://run.linkerd.io/emojivoto.yml \
  | linkerd inject - \
  | kubectl apply -f -

namespace "emojivoto" injected
serviceaccount "emoji" skipped
serviceaccount "voting" skipped
serviceaccount "web" skipped
service "emoji-svc" skipped
service "voting-svc" skipped
service "web-svc" skipped
deployment "emoji" injected
deployment "vote-bot" injected
deployment "voting" injected
deployment "web" injected

namespace/emojivoto configured
serviceaccount/emoji unchanged
serviceaccount/voting unchanged
serviceaccount/web unchanged
service/emoji-svc unchanged
service/voting-svc unchanged
```

```
service/web-svc unchanged
deployment.apps/emoji configured
deployment.apps/vote-bot configured
deployment.apps/voting configured
deployment.apps/web configured
```

14. The demo application is now meshed in the Linkerd mesh and ready to receive traffic from Nginx Ingress. Expose the Linkerd Dashboard by entering this command:

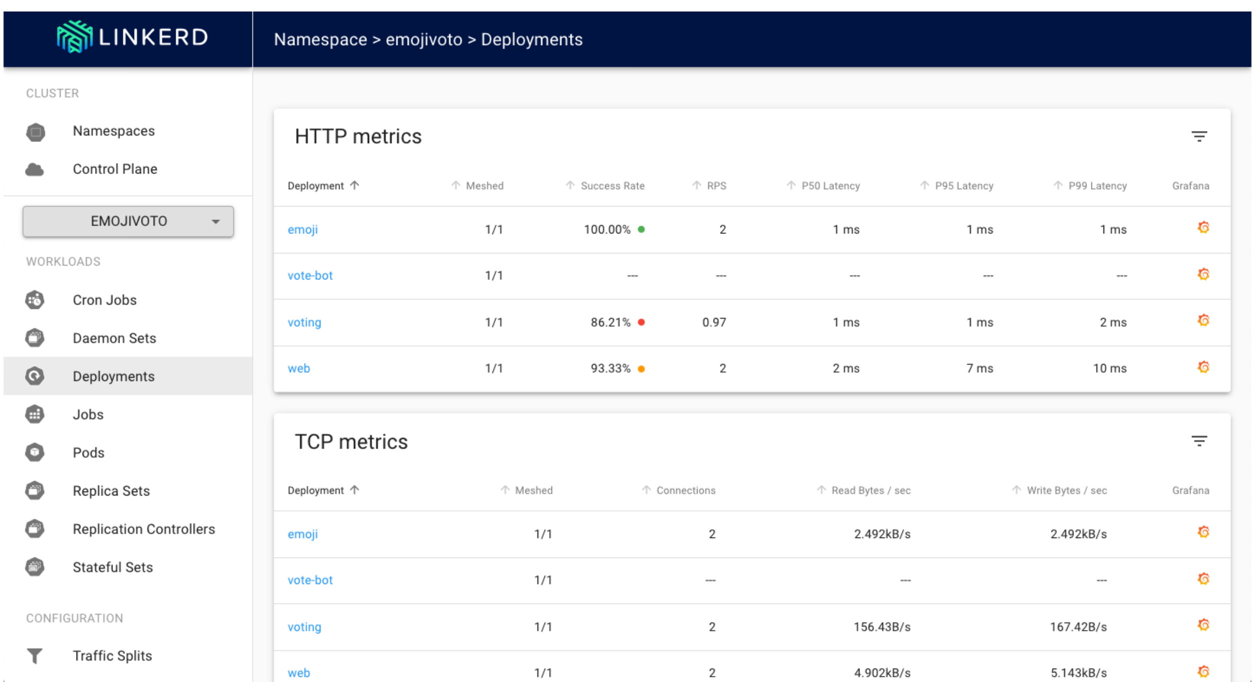
    ```
    yourname@ubuntu-vm:~$ linkerd viz dashboard --port 8080 --address
    0.0.0.0 --show url
    ```

    ```
    http://0.0.0.0:8080
    Grafana dashboard available at:
    http://0.0.0.0:8080/grafana
    ```

15. Open and explore the Linkerd dashboard in your web browser by going to the IP address of your VM at port `8080`, which would look something like this:

    ```
    http://127.0.0.1:8080
    ```

    Here you can get a live look at how services in the mesh are performing. For example, if you take a look at the deployments running in the `emojivoto` namespace, you can see that there appears to be an issue with the `voting` service. You will learn how to address this in a future lab.



16. When you are done exploring the dashboard, hit Control-C in your terminal to stop the dashboard from running.

17. You've reached the end of this lab. If you're not starting the next lab right away, you can stop the Linkerd cluster by running this command:

```
yourname@ubuntu-vm:~$ docker stop $(docker ps -a -f
name=linkerd-control-plane -q)

8af4c08524df
```

Lab 5.2. Install an Istio Service Mesh

Overview

In this lab, you'll be installing a service mesh using Istio.

In the previous labs, you created an Ingress Controller with annotations. One of those annotations was for Istio. Because the configuration is already set, you don't have to do anything other than installing Istio itself because the annotation already exists inside of the Ingress Controller. Below is the Nginx Ingress Controller manifest with the Istio annotation highlighted in orange to remind you of what it looks like. **Please note** that you do **not** have to re-run the configuration as you already did in the previous labs.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  namespace: emojivoto
  name: ingress-emojivoto
  annotations:
    ingress.kubernetes.io/rewrite-target: /
    nginx.ingress.kubernetes.io/service-upstream: "true"
    linkerd.io/inject: enabled
    kubernetes.io/ingress.class: istio
    consul.hashicorp.com/connect-inject: “true”
spec:
  ingressClassName: nginx
  rules:
    - http:
        paths:
        - pathType: Prefix
          path: "/"
          backend:
            service:
              name: web-svc
              port:
```

```
          number: 80
```

1. Connect to the VM that hosts your Kubernetes clusters.

2. To start the cluster that will contain the Istio service mesh, issue this command:

```
yourname@ubuntu-vm:~$ docker start $(docker ps -a -f
name=istio-control-plane -q)

2d1d09fadf21
```

3. To switch the **`kind`** context to the Istio cluster, use this command:

```
yourname@ubuntu-vm:~$ kubectl config use-context kind-istio

Switched to context "kind-istio".
```

4. Download the Istio version 1.6.8 release:

```
yourname@ubuntu-vm:~$ curl -L https://istio.io/downloadIstio | sh

curl -L https://istio.io/downloadIstio | sh
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   101  100   101    0     0   1373      0 --:--:-- --:--:-- --:--:--  1578
100  4926  100  4926    0     0  23399      0 --:--:-- --:--:-- --:--:-- 23399

Downloading istio-1.14.2 from
https://github.com/istio/istio/releases/download/1.14.2/istio-1.14.2-osx-arm64
.tar.gz ...

Istio 1.14.2 Download Complete!

Istio has been successfully downloaded into the istio-1.14.2 folder on your
system.

Next Steps:
See https://istio.io/latest/docs/setup/install/ to add Istio to your
Kubernetes cluster.

To configure the istioctl client tool for your workstation,
add the /Users/michael/istio-1.14.2/bin directory to your environment path
variable with:
       export PATH="$PATH:/Users/michael/istio-1.14.2/bin"
```

```
Begin the Istio pre-installation check by running:
       istioctl x precheck
```

5. Add **istioctl** to your current **PATH** by using the **export PATH** command specified below:

```
yourname@ubuntu-vm:~$ export PATH=$PWD/bin:$PATH

[no output if the command succeeds]
```

6. You should also add the linkerd CLI to the **$PATH** variable for your shell with the command below (assuming you are using **bash** for your shell). This will permanently add it to your **PATH**. Without doing this once, if you log out and log back in, your **PATH** will be replaced and your **linkerd** commands will not work correctly until you manually update your **PATH** as you did in the previous step.

```
yourname@ubuntu-vm:~$ echo "export
PATH=$PATH:$HOME/istio-1.14.2/bin" >> ~/.bashrc

[no output if the command succeeds]
```

7. Install Istio in the cluster with the following command. Note that you may be prompted about proceeding with the install; make sure to press ‘y’ and hit ‘enter’ to continue. Simply hitting ‘enter’ without first pressing ‘y’ will cause the default (capitalized) option, ‘N’, to be chosen and the install will be terminated.

```
yourname@ubuntu-vm:~$ istioctl install \
  --set values.gateways.istio-ingressgateway.enabled=false

This will install the default Istio profile into the cluster.
Proceed? (y/N) y
Detected that your cluster does not support third party JWT
authentication. Falling back to less secure first party
JWT. See
https://istio.io/docs/ops/best-practices/security/#configure-thir
d-party-service-account-tokens for details.
✔ Istio core installed
✔ Istiod installed
✔ Addons installed
✔ Installation complete
```

8. Let’s explore what you just installed! There is a lot of configuration that goes into making sure Istio runs the way it is supposed to. Taking a look at the containers that are

installed, Istio is a relatively simple deployment with a single pod running the control plane processes, **prometheus** for collecting metrics from the control plane and data plane, and a **grafana** instance for viewing those metrics. The following command will generate the equivalent to the **yaml** installed above. Running through this file will give you a better sense of [Istio's architecture](#).

```
yourname@ubuntu-vm:~$ istioctl manifest generate \
  --set values.gateways.istio-ingressgateway.enabled=false
  --set values.global.jwtPolicy=first-party-jwt >
istio-manifest.yaml

[no output if the command succeeds]
```

9. Next, you can install Grafana to work with Istio by capturing output from Istio so you can see it in a dashboard.

```
yourname@ubuntu-vm:~$ kubectl apply -f
https://raw.githubusercontent.com/istio/istio/release-1.14/samples/addons/grafana.yaml
```

10. Nginx Ingress is now configured to route to services in your Istio mesh. You need to add the demo application you installed to the mesh. **istioctl kube-inject** takes the Kubernetes configuration manifest and outputs it with the service proxy configurations added to any object that deploys applications in Kubernetes (Deployment, Pod, Job, etc.). Pairing this with **kubectl** allows you to manually inject service proxies to the application

```
yourname@ubuntu-vm:~$ curl -sL
https://run.linkerd.io/emojivoto.yml \
  | istioctl kube-inject -f - \
  | kubectl apply -f -

namespace/emojivoto unchanged
serviceaccount/emoji unchanged
serviceaccount/voting unchanged
serviceaccount/web unchanged
service/emoji-svc unchanged
service/voting-svc unchanged
service/web-svc unchanged
deployment.apps/emoji configured
deployment.apps/vote-bot configured
deployment.apps/voting configured
```

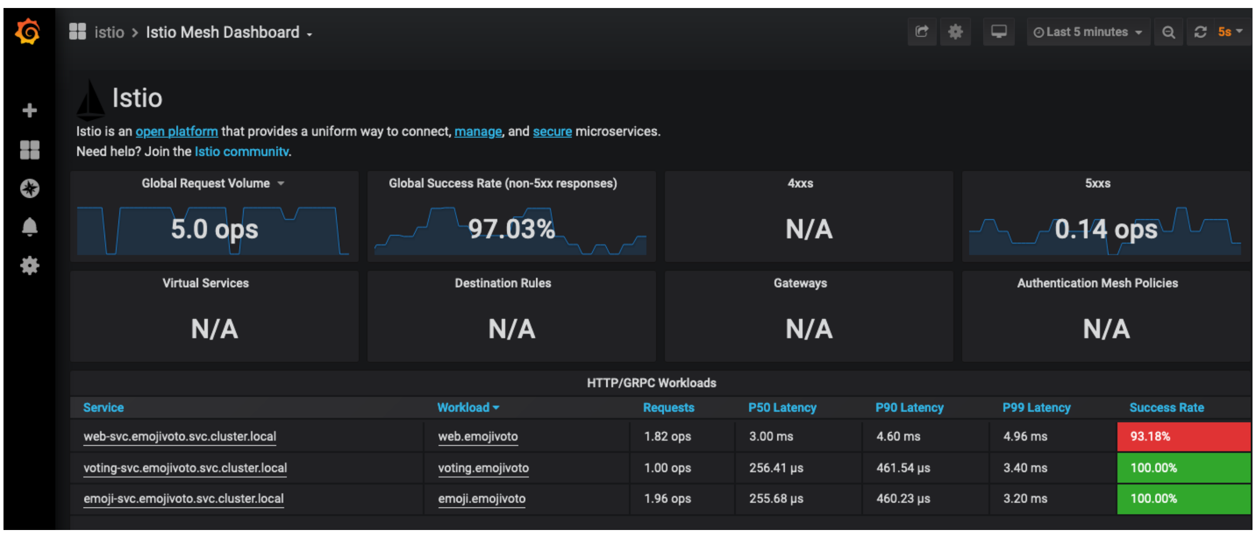
```
deployment.apps/web configured
```

11. The demo application is now meshed in the Istio mesh and ready to receive traffic from Nginx Ingress. You can access this dashboard in two steps.

    a. Expose the Istio dashboard by port-forwarding to the VM:

    ```
    yourname@ubuntu-vm:~$ kubectl port-forward -n istio-system
    svc/grafana 8080:3000

    Forwarding from 127.0.0.1:8080 -> 3000
    Forwarding from [::1]:8080 -> 3000
    ```

    b. Access the default Istio dashboard by opening **`http://localhost:8080`** in a web browser, where you replace **`{{VM_IP_ADDRESS}}`** with your VM's IP address. Through this dashboard, you can get a live look at how services in the mesh are performing. For example, you can see that there appears to be an issue with the **`web-svc.emojivoto.svc.cluster.local`** service. You will learn how to address this in a future lab.



12. You've reached the end of this lab. If you're not starting the next lab right away, you can stop the Istio cluster by running this command:

```
yourname@ubuntu-vm:~$ docker stop $(docker ps -a -f
name=istio-control-plane -q)

2d1d09fadf21
```

Lab 5.3 - Configure Mutual TLS for Istio

Overview

In this lab, you'll be configuring mutual TLS (mTLS) with Istio. Istio has two configuration modes for mTLS:

- **`PERMISSIVE`** allows both encrypted and unencrypted (plaintext) traffic. While all requests between service proxies are encrypted by default, this mode allows services outside of the mesh to connect to services within the mesh without using encryption or authentication.

- **`STRICT`** requires that all connections be encrypted and that all services connecting to a service in the mesh authenticate themselves with certificates. This mode makes the mesh a closed environment that only services with the right certificates can connect to.

The default for Istio is **`PERMISSIVE`** mode because that makes migrations easy. Since you have already meshed your application, you can enable **`STRICT`** mode to prevent services outside of the mesh from connecting.

1. If you don't still have an active session from the previous Istio lab, perform these steps to reestablish it:

    a. Connect to the VM that hosts your Kubernetes clusters.

    b. To start the cluster that contains the Istio service mesh, issue this command:

    ```
    yourname@ubuntu-vm:~$ docker start $(docker ps -a -f
    name=istio-control-plane -q)

    2d1d09fadf21
    ```

    c. To switch the **`kind`** context to the Istio cluster, use this command:

```
yourname@ubuntu-vm:~$ kubectl config use-context kind-istio

Switched to context "kind-istio".
```

2. To demonstrate that **STRICT** mode requires all services to authenticate, you first need to have a service running outside of the mesh. Since all services in your demo application and your ingress controller are already establishing secure connections, you need to deploy another service. To do that, you can run a container that has the **curl** binary in the default namespace so the sidecar is not auto injected. Use the following command to do that:

```
yourname@ubuntu-vm:~$ kubectl run -n default curl-pod \
  --image curlimages/curl --command -- sleep infinity

pod/curl-pod created
```

3. Create a forwarder to the **web-svc** emoji service

```
yourname@ubuntu-vm:~$ kubectl port-forward service/web-svc -n
emojivoto 8080:80
```

4. Now you can send requests from a service outside the mesh. Make sure it works by sending a request to the **web-svc**. You should receive a **200** response code, which indicates you can successfully connect to the **web-svc** service with **PERMISSIVE** mode.

```
yourname@ubuntu-vm:~$ kubectl exec -n default curl-pod \
  -- curl -s -o /dev/null -w "Upstream Response Code:
%{http_code}\n" \
  http://web-svc.emojivoto/

Upstream Response Code: 200
```

5. Enable **STRICT** mode with a **PeerAuthentication** configuration.

```
yourname@ubuntu-vm:~$ kubectl apply -n istio-system -f - <<EOF
apiVersion: "security.istio.io/v1beta1"
kind: "PeerAuthentication"
metadata:
  name: "default"
spec:
  mtls:
    mode: STRICT
EOF
```

```
peerauthentication.security.istio.io/default created
```

6. To clean up from this lab, remove the **`PeerAuthentication`** configuration:

```
yourname@ubuntu-vm:~$ kubectl delete -n istio-system
peerauthentication default

peerauthentication.security.istio.io "default" deleted
```

7. Finish the lab cleanup by removing the **`curl-pod`** service:

```
yourname@ubuntu-vm:~$ kubectl delete pod -n default curl-pod

pod "curl-pod" deleted
```

8. You've reached the end of this lab. If you're not starting the next lab right away, you can stop the Istio cluster by running this command:

```
yourname@ubuntu-vm:~$ docker stop $(docker ps -a -f
name=istio-control-plane -q)

2d1d09fadf21
```

THE LINUX FOUNDATION | Training & Certification

Lab 5.4 - Install a Consul Service Mesh

Overview

In this lab, you'll be installing a service mesh using Consul.

Consul provides a lot of the same functionality as other service meshes, but instead of being designed to run entirely on Kubernetes, Consul was built to connect services throughout your datacenter regardless of where they are running.

In the previous labs, you created an Ingress Controller with annotations. One of those annotations was for Consul. Because the configuration is already set, you don't have to do anything other than installing Consul itself because the annotation already exists inside of the Ingress Controller. Below is the Nginx Ingress Controller manifest with the Consul annotation highlighted in orange to remind you of what it looks like. **Please note** that you do **not** have to re-run the configuration as you already did in the previous labs.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  namespace: emojivoto
  name: ingress-emojivoto
  annotations:
    ingress.kubernetes.io/rewrite-target: /
    nginx.ingress.kubernetes.io/service-upstream: "true"
    linkerd.io/inject: enabled
    kubernetes.io/ingress.class: istio
    consul.hashicorp.com/connect-inject: “true”
spec:
  ingressClassName: nginx
  rules:
    - http:
        paths:
        - pathType: Prefix
          path: "/"
```

```
      backend:
        service:
          name: web-svc
          port:
            number: 80
```

1. Connect to the VM that hosts your Kubernetes clusters.

2. To start the cluster that will contain the Consul service mesh, issue this command:

```
yourname@ubuntu-vm:~$ docker start $(docker ps -a -f
name=consul-control-plane -q)

2e64e6e909e1
```

3. To switch the **kind** context to the Consul cluster, use this command:

```
yourname@ubuntu-vm:~$ kubectl config use-context kind-consul

Switched to context "kind-consul".
```

4. To install Consul with Helm, issue this command:

```
yourname@ubuntu-vm:~$ helm repo add hashicorp
https://helm.releases.hashicorp.com

"hashicorp" has been added to your repositories
```

5. Enter the following **two** commands to configure Consul to run on your single node cluster and automatically inject the sidecar to pods in the **emojivoto** namespace. This may take a while to complete.

```
yourname@ubuntu-vm:~$ kubectl create namespace consul

namespace/consul created

yourname@ubuntu-vm:~$ helm upgrade --install -n consul consul
hashicorp/consul --wait -f - <<EOF
global:
  name: consul
server:
  replicas: 1
  bootstrapExpect: 1
```

```
connectInject:
  enabled: true
EOF

NAME: consul
LAST DEPLOYED: Thu Jul  9 20:05:10 2020
NAMESPACE: consul
STATUS: deployed
REVISION: 1
NOTES:
Thank you for installing HashiCorp Consul!
Now that you have deployed Consul, you should look over the docs
on using
Consul with Kubernetes available here:
https://www.consul.io/docs/platform/k8s/index.html
Your release is named consul.
To learn more about the release if you are using Helm 2, run:
 $ helm status consul
 $ helm get consul
To learn more about the release if you are using Helm 3, run:
 $ helm status consul
 $ helm get all consul
```

6. Make sure Consul has started before proceeding with the next step by issuing the following command. You should see three names starting with **-consul**, and each one of them should have a status of **Running** and a Ready value of **1/1**. If you don't see those for each of the three names, wait a few minutes and run the command again.

```
yourname@ubuntu-vm:~$ kubectl get pods -n consul

consul-connect-injector-webhook-deployment-dcd56544f-kpq9t   1/1
Running    0          70s
consul-p2rv7                                                 1/1
Running    0          70s
consul-server-0                                              1/1
Running    0          68s
```

7. Let's explore what you just installed! Run the command below, which will output all of the configuration you installed in the cluster. This includes a Consul server backend, an agent that acts as the control plane for the sidecar proxies, and a service in charge of adding your services to the mesh. Running through this output will give you a better sense of Consul's architecture.

```
helm get all consul -n consul
```

```
NAME: consul
LAST DEPLOYED: Sat Jul 25 17:59:43 2020
NAMESPACE: consul
STATUS: deployed
REVISION: 1
USER-SUPPLIED VALUES:
connectInject:
 enabled: true
global:
 name: consul
server:
 bootstrapExpect: 1
 replicas: 1

[remaining content deleted for brevity]
```

8. After installing Consul, you need to add the application you installed to the mesh. Consul decides to add a pod to the mesh by looking for the **`consul.hashicorp.com/connect-inject: "true"`** annotation. Since the `consul` CLI does not have a command for automatically do this, you can manually accomplish this with the following command:

```
yourname@ubuntu-vm:~$ curl -sL
https://run.linkerd.io/emojivoto.yml \
  | sed 's|    metadata:|    metadata:\n      annotations:\n
consul.hashicorp.com/connect-inject: "true"|' \
  | kubectl apply -f -

namespace/emojivoto unchanged
serviceaccount/emoji unchanged
serviceaccount/voting unchanged
serviceaccount/web unchanged
service/emoji-svc unchanged
service/voting-svc unchanged
service/web-svc unchanged
deployment.apps/emoji configured
deployment.apps/vote-bot configured
deployment.apps/voting configured
deployment.apps/web configured
```

9. Verify that the Emoji Vote pods are running in the cluster and ready by issuing this command. You should see a status of "**`Running`**" and a Ready value of "**`3/3`**" for each of the four pods. These extra pods are the **`consul-connect-sidecar-proxy`** and the

consul-connect-lifecycle-sidecar, which are responsible for routing traffic and interacting with the control plane, respectively. If you don't see the expected status and Ready values, wait a few minutes and try the command again.

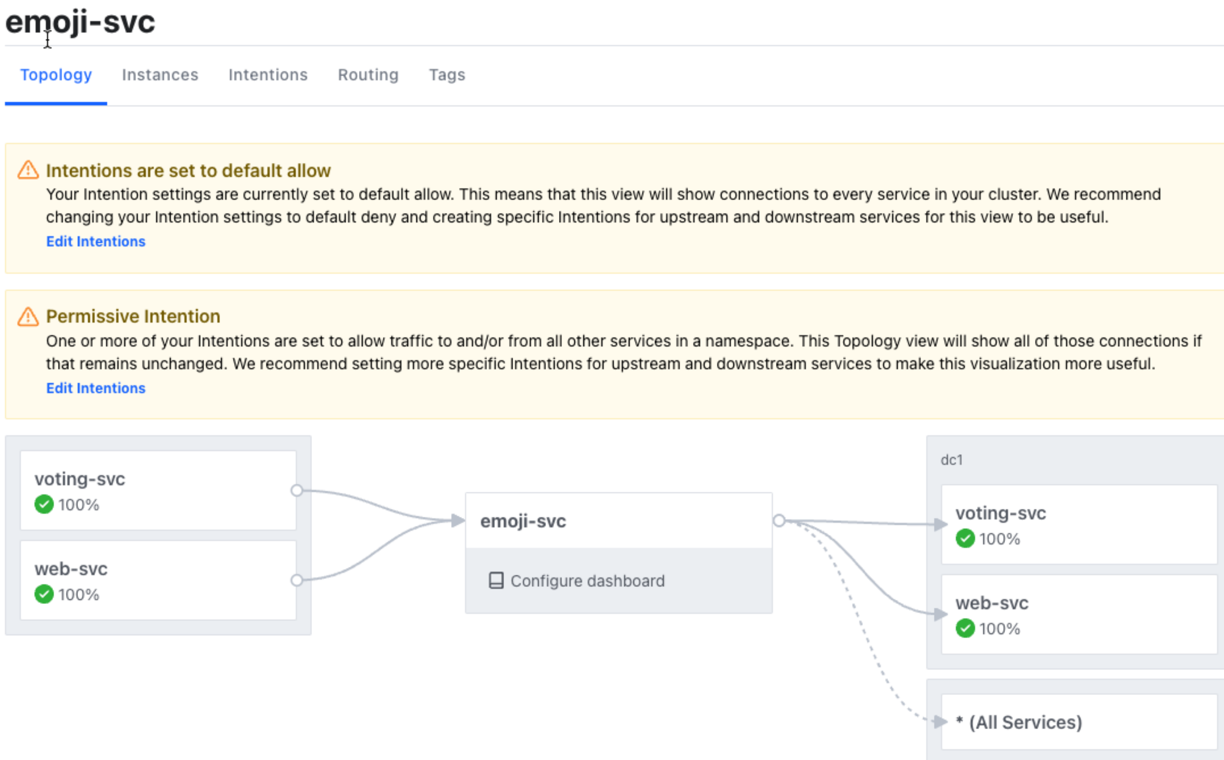
```
yourname@ubuntu-vm:~$ kubectl get pod -n emojivoto -w

NAME                          READY   STATUS    RESTARTS   AGE
emoji-587bf97dbb-6jmsx        3/3     Running   0          91s
vote-bot-5f987ddd8f-7fsxk     3/3     Running   0          91s
voting-85477587c9-8qmfk       3/3     Running   0          91s
web-bd44c459c-dzczb           3/3     Running   0          91s
```

10. Once you see a **Running** status and a Ready value of **3/3** for all four names, hit Control-C to stop the status updates from being displayed. This may take a few minutes.

11. Consul does not automatically hijack the pods' networking as other meshes do. This means that you can still connect directly to the container running on the pod without going through the service proxy. This has the benefits of making migrations to mesh easier and giving options to opt out of the mesh, but you would need to update the `Service` definitions so requests go to the service proxy instead of the application container. Combined with the previous command above to inject the service proxies, the following command will inject and connect the application to the mesh.

```
yourname@ubuntu-vm:~$ curl -sL
https://run.linkerd.io/emojivoto.yml \
  | sed 's|    metadata:|    metadata:\n      annotations:\n
consul.hashicorp.com/connect-inject: "true"|' \
  | sed 's|targetPort: 8080|targetPort: 20000|' \
  | kubectl apply -f -

namespace/emojivoto unchanged
serviceaccount/emoji unchanged
serviceaccount/voting unchanged
serviceaccount/web unchanged
service/emoji-svc configured
service/voting-svc configured
service/web-svc configured
deployment.apps/emoji unchanged
deployment.apps/vote-bot unchanged
deployment.apps/voting unchanged
deployment.apps/web unchanged
```

12. The demo application is now a part of the Consul Connect service mesh and ready to receive traffic from Nginx Ingress. Take some time to explore Consul a little more.



13. You've reached the end of this lab. If you're not starting the next lab right away, you can stop the Consul cluster by running this command:

```
yourname@ubuntu-vm:~$ docker stop $(docker container ls -a -f
name=consul-control-plane -q)

2e64e6e909e1
```

Lab 6.1 - Configure SMI Traffic Splitting with Linkerd

Overview

In this lab, you'll be implementing traffic splitting by using the SMI specification with Linkerd to perform a canary release. The canary release will transition users to a new version of the demo app that fixes the bug with the donut emoji.

1. Connect to the Ubuntu VM that hosts your Kubernetes clusters.

2. To start the cluster that contains the Linkerd service mesh, issue this command:

```
yourname@ubuntu-vm:~$ docker start $(docker ps -a -f
name=linkerd-control-plane -q)
```

```
8af4c08524df
```

3. To switch the **kind** context to the Linkerd cluster, use this command:

```
yourname@ubuntu-vm:~$ kubectl config use-context kind-linkerd
```

```
Switched to context "kind-linkerd".
```

4. Install the Linkerd SMI

```
yourname@ubuntu-vm:~$ curl --proto '=https' --tlsv1.2 -sSfL
https://linkerd.github.io/linkerd-smi/install | sh
```

```
yourname@ubuntu-vm:~$ linkerd smi install | kubectl apply -f -
```

```
namespace/linkerd-smi created
deployment.apps/smi-adaptor created
clusterrole.rbac.authorization.k8s.io/smi-adaptor created
clusterrolebinding.rbac.authorization.k8s.io/smi-adaptor created
```

```
serviceaccount/smi-adaptor created
customresourcedefinition.apiextensions.k8s.io/trafficsplits.split.smi-sp
ec.io configured
```

5. Verify that the installation was successful.

```
yourname@ubuntu-vm:~$ linkerd smi check

linkerd-smi
-----------
√ linkerd-smi extension Namespace exists
√ SMI extension service account exists
√ SMI extension pods are injected
√ SMI extension pods are running
√ SMI extension proxies are healthy

Status check results are √
```

6. Create a new namespace and install the SMI sample app:

```
yourname@ubuntu-vm:~$ kubectl create namespace trafficsplit-sample

yourname@ubuntu-vm:~$ linkerd inject
https://raw.githubusercontent.com/linkerd/linkerd2/main/test/integration/viz/t
rafficsplit/testdata/application.yaml | kubectl -n trafficsplit-sample apply
-f -
```

7. Confirm that the installation of the app was successful.

```
yourname@ubuntu-vm:~$ kubectl get deployments -n
trafficsplit-sample

NAME          READY   UP-TO-DATE   AVAILABLE   AGE
backend       1/1     1            1           70s
failing       1/1     1            1           70s
slow-cooker   1/1     1            1           69s
```

8. Next, configure a traffic split to split traffic on the **backend-svc** to distribute load between it and the **failing-svc**.

```
yourname@ubuntu-vm:~$ cat <<EOF | kubectl apply -f -
apiVersion: split.smi-spec.io/v1alpha2
kind: TrafficSplit
metadata:
  name: backend-split
```

```
  namespace: trafficsplit-sample
spec:
  service: backend-svc
  backends:
  - service: backend-svc
    weight: 500
  - service: failing-svc
    weight: 500
EOF
```

9. Verify that the traffic splitting is working as expected by running the following command.

```
yourname@ubuntu-vm:~$ linkerd viz edges deploy -n trafficsplit-sample
```

```
      SRC            DST            SRC_NS                DST_NS
SECURED
prometheus     backend       linkerd-viz           trafficsplit-sample   √
prometheus     failing       linkerd-viz           trafficsplit-sample   √
prometheus     slow-cooker   linkerd-viz           trafficsplit-sample   √
slow-cooker    backend       trafficsplit-sample   trafficsplit-sample   √
slow-cooker    failing       trafficsplit-sample   trafficsplit-sample   √
```

10. Clean up the environment with the sample app that you just deployed for this lab with the following command.

```
yourname@ubuntu-vm:~$ kubectl delete namespace/trafficsplit-sample
```

11. Make sure to stop the Linkerd cluster by running this command:

```
yourname@ubuntu-vm:~$ docker stop $(docker ps -a -f
name=linkerd-control-plane -q)

8af4c08524df
```

Lab 7.1 - Timeouts and Retries with Linkerd

Overview

In this lab, you'll be configuring timeouts and retries for service-to-service networking with Linkerd. Timeouts and retries are important for mitigating cascading failures when a microservice is not responding as expected.

1. Connect to the VM that hosts your Kubernetes clusters.

2. To start the cluster that contains the Linkerd service mesh, issue this command:

   ```
   yourname@ubuntu-vm:~$ docker start $(docker ps -a -f
   name=linkerd-control-plane -q)

   8af4c08524df
   ```

3. To switch the **`kind`** context to the Linkerd cluster, use this command:

   ```
   yourname@ubuntu-vm:~$ kubectl config use-context kind-linkerd

   Switched to context "kind-linkerd".
   ```

4. In the previous labs you used the Emojivoto application to demonstrate basic service mesh functionality. Since Linkerd is currently unable to configure retries on gRPC requests, which is what Emojivoto uses, you will use a different example application for this lab. Run the following command to remove the Emojivoto application from your cluster. Note that it may take this command a few minutes to complete executing and return you to a command line prompt.

   ```
   yourname@ubuntu-vm:~$ kubectl delete namespace emojivoto
   ```

```
namespace "emojivoto" deleted
serviceaccount "emoji" deleted
serviceaccount "voting" deleted
serviceaccount "web" deleted
service "emoji-svc" deleted
service "voting-svc" deleted
service "web-svc" deleted
deployment.apps "emoji" deleted
deployment.apps "vote-bot" deleted
deployment.apps "voting" deleted
deployment.apps "web" deleted
```

5. Make sure Linkerd and Nginx Ingress have started and have all resources ready before continuing. The following command will list the **Pod**s in all namespaces. All should have a status of **Running** and a ready status with the same numbers on the left and right sides of the slash ("/"). If any of the lines don't show those statuses, wait a few minutes and reissue the command again.

```
yourname@ubuntu-vm:~$ kubectl get pods -n linkerd

NAMESPACE              NAME
linkerd-destination-7df897cf9b-4w49m      4/4     Running   8
(2m42s ago)   20h
linkerd-identity-5f4dbf785d-vxnrm         2/2     Running   5
(2m42s ago)   20h
linkerd-proxy-injector-5497f6fbd7-qzwvx   2/2     Running   4
(2m42s ago)   20h


yourname@ubuntu-vm:~$ kubectl get pods -n ingress-nginx

ingress-nginx-admission-create-r7gvr       0/1     Completed   0
40h
ingress-nginx-admission-patch-6prps        0/1     Completed   0
40h
ingress-nginx-controller-b66cc4b74-pp4kh   1/1     Running     4
(2m7s ago)   40h
```

6. For this lab, you will use another application provided by Buoyant (the creators of Linkerd) to demonstrate Linkerd's ability to perform retries and timeouts on individual services in the mesh. Install the booksapp application:

```
yourname@ubuntu-vm:~$ kubectl create ns booksapp && \
```

```
curl -sL https://run.linkerd.io/booksapp.yml \
| linkerd inject - \
| kubectl -n booksapp apply -f -
```

```
namespace/booksapp created

service "webapp" skipped
deployment "webapp" injected
service "authors" skipped
deployment "authors" injected
service "books" skipped
deployment "books" injected
deployment "traffic" injected

service/webapp created
deployment.apps/webapp created
service/authors created
deployment.apps/authors created
service/books created
deployment.apps/books created
deployment.apps/traffic created
```

7. Next, configure the Nginx Ingress to use the new app.

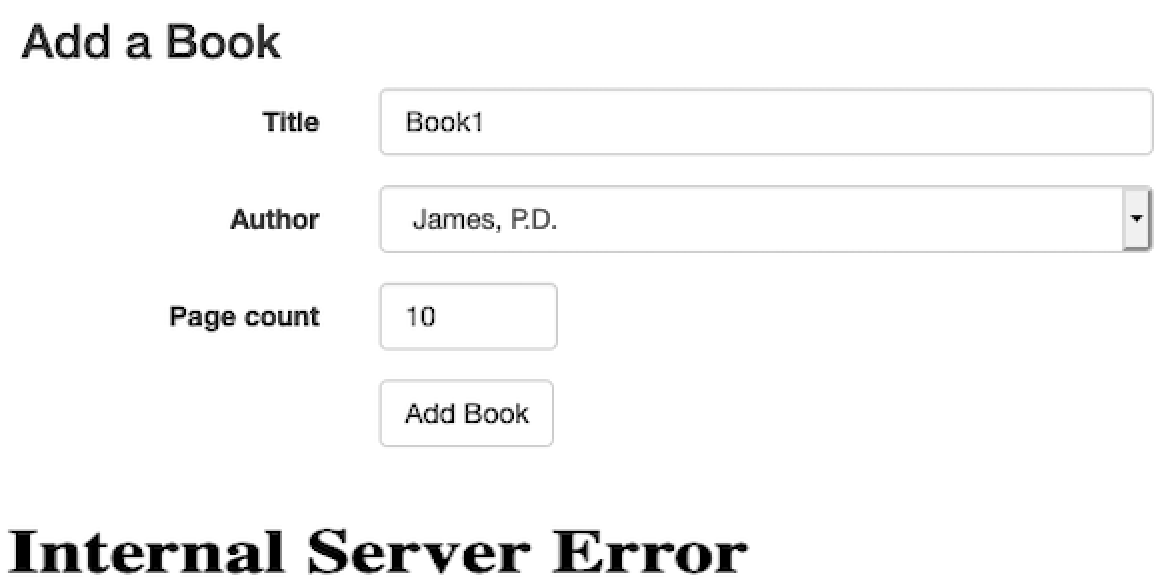
```
yourname@ubuntu-vm:~$ kubectl apply -f - <<EOF
—
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  namespace: booksapp
  name: ingress-booksapp
  annotations:
    ingress.kubernetes.io/rewrite-target: /
    nginx.ingress.kubernetes.io/service-upstream: "true"
    consul.hashicorp.com/connect-inject: “true”
spec:
  ingressClassName: nginx
  rules:
    - http:
        paths:
        - pathType: Prefix
          path: "/"
          backend:
            service:
              name: webapp
```

```
            port:
              number: 7000
EOF

ingress.networking.k8s.io/ingress-booksapp created
```

8. Create a Kubernetes `port-forward`:

```
yourname@ubuntu-vm:~$ kubectl port-forward
service/ingress-nginx-controller -n ingress-nginx 8080:80
```

9. Navigate to 127.0.0.1:8080 in a web browser to interact with the new application. This application is a database of books and authors. Clicking a book will give you information about the book like the page count and author. Clicking an author will give you a list of the books they have written. You can also add authors and books to the database. Add several books to the database, and you will find that the function fails around half the time.



10. Linkerd exposes the [`ServiceProfile`]() configuration to see per-route metrics, and to configure timeouts and automatic retries on those routes. This allows for you to pinpoint some errors and mitigate them at the network level while you work on patching the code. For HTTP services, there are two mechanisms for creating `ServiceProfiles`. You can use live traffic to [automatically create `ServiceProfiles`]() based on requests through the proxy or you can use [OpenAPI (Swagger)]() documentation for the service to generate a `ServiceProfile`. To guarantee you have a full understanding of the available routes, you will use the available OpenAPI docs for this service.

```
yourname@ubuntu-vm:~$ curl -sL
https://run.linkerd.io/booksapp/webapp.swagger \
  | linkerd -n booksapp profile --open-api - webapp \
  | kubectl -n booksapp apply -f -

serviceprofile.linkerd.io/webapp.booksapp.svc.cluster.local
created
```

11. Wait a few minutes, then take a look at the metrics for these routes from the command line. The **`linkerd routes`** command returns a snapshot of the performance of each route in the **`ServiceProfile`** above. The output appears to indicate that **`POST`** requests to the **`/books`** endpoint fail around half the time.

```
yourname@ubuntu-vm:~$ linkerd viz routes -n booksapp
service/webapp

ROUTE                          SERVICE   SUCCESS      RPS
LATENCY_P50    LATENCY_P95    LATENCY_P99
GET /                          webapp    100.00%    0.6rps          15ms
      20ms         20ms
GET /authors/{id}              webapp    100.00%    0.6rps          8ms
      10ms         10ms
GET /books/{id}                webapp    100.00%    1.1rps          8ms
      18ms         20ms
POST /authors                  webapp    100.00%    0.6rps          9ms
      19ms         20ms
POST /authors/{id}/delete      webapp    100.00%    0.6rps          15ms
      20ms         20ms
POST /authors/{id}/edit        webapp          -     -              -
      -            -
POST /books                    webapp     45.83%    3.0rps
9ms         18ms         20ms
POST /books/{id}/delete        webapp    100.00%    0.6rps          8ms
      10ms         10ms
POST /books/{id}/edit          webapp     42.35%    1.4rps          16ms
      84ms         97ms
[DEFAULT]                      webapp          -     -              -
      -            -
```

12. You now know that requests from the webapp service to the books service appear to be failing. You can get more visibility to this by creating a **`ServiceProfile`** for the upstream services as well. You can do this with OpenAPI documentation for the books servicel.

```
yourname@ubuntu-vm:~$ curl -sL
https://run.linkerd.io/booksapp/books.swagger \
  | linkerd -n booksapp profile --open-api - books \
  | kubectl -n booksapp apply -f -

serviceprofile.linkerd.io/books.booksapp.svc.cluster.local
created
```

13. Wait a few minutes, then run the **`linkerd routes`** command below to review the requests from the webapp service to the books service. You can see that **`POST`** *and* **`PUT`** actions seem to be the ones causing errors.

```
yourname@ubuntu-vm:~$ linkerd viz routes -n booksapp
deploy/webapp --to service/books

ROUTE                          SERVICE    SUCCESS     RPS
LATENCY_P50   LATENCY_P95   LATENCY_P99
DELETE /books/{id}.json        books    100.00%    1.3rps          5ms
     9ms          10ms
GET /books.json                books    100.00%    1.9rps          2ms
     3ms          4ms
GET /books/{id}.json           books    100.00%    2.5rps          2ms
     3ms          17ms
POST /books.json               books 44.25%    2.9rps              5ms
     16ms         19ms
PUT /books/{id}.json           books 54.05%    1.2rps              6ms
     84ms         97ms
[DEFAULT]                      books       -     -          -
-          -
```

14. It is a little strange that two independent routes, both the **`POST`** *and* **`PUT`**, are erroring at the same rate. You can dig a little deeper by seeing if both of these routes rely on another service. Checking the output of **`linkerd tap`** with the command below will give you a log of the requests flowing through the books service. Hit **`Control-C`** to stop the log from scrolling so you can read it. From that log, you should find that the books service is making a lot of **`HEAD`** requests to the authors service. Based on the number of 503s that are being returned, those requests appear to be failing quite frequently.

```
yourname@ubuntu-vm:~$ linkerd viz tap -n booksapp deploy/books

req id=9:49 proxy=out src=10.244.0.53:37820 dst=10.244.0.50:7001
tls=true :method=HEAD :authority=authors:7001
:path=/authors/3252.json
```

```
rsp id=9:49 proxy=out src=10.244.0.53:37820 dst=10.244.0.50:7001
tls=true :status=503 latency=2197µs
end id=9:49 proxy=out src=10.244.0.53:37820 dst=10.244.0.50:7001
tls=true duration=16µs response-length=0B
```

15. Let's investigate this further. To start, create a **`ServiceProfile`** for the authors service from the swagger doc.

```
yourname@ubuntu-vm:~$ curl -sL
https://run.linkerd.io/booksapp/authors.swagger \
  | linkerd -n booksapp profile --open-api - authors \
  | kubectl -n booksapp apply -f -

serviceprofile.linkerd.io/authors.booksapp.svc.cluster.local
created
```

16. After creating the **`ServiceProfile`**, wait a few minutes and then check **`linkerd routes`** for requests from the books service to the authors service. You will see that the **`HEAD`** requests are failing.

```
yourname@ubuntu-vm:~$ linkerd viz routes -n booksapp deploy/books
--to service/authors

ROUTE                           SERVICE   SUCCESS     RPS
LATENCY_P50   LATENCY_P95   LATENCY_P99
DELETE /authors/{id}.json       authors         -      -           -
      -           -
GET /authors.json               authors         -      -           -
      -           -
GET /authors/{id}.json          authors         -      -           -
      -           -
HEAD /authors/{id}.json         authors    54.87%   3.8rps
1ms          2ms          3ms
POST /authors.json              authors         -      -           -
      -           -
[DEFAULT]                       authors         -      -           -
      -           -
```

17. From the output of the commands in the previous steps, you can see that linkerd is reporting similar behavior to what we have been seeing while interacting with the application. The actions that add books to the database fail around half the time. Based on your debugging work, it seems that this is actually due to an issue with the **`HEAD`** requests to the authors service. This issue needs to be remedied quickly. Since coding

can be a time consuming activity, Linkerd gives you the ability to easily put some mitigations in place while you work on debugging what went wrong.

a. The **ServiceProfile** is not just capable of *reporting* per-route metrics; it can also *configure* per-route rules for linkerd. Based on the data above, adding books to the database appears to be failing around half the time. You can configure automatic retries on those actions so that Linkerd will try to re-send the request instead of reporting an error to your users. Manually edit the **ServiceProfile** by running **kubectl edit** to open it in a text editor.

```
yourname@ubuntu-vm:~$ kubectl edit -n booksapp
serviceprofile authors.booksapp.svc.cluster.local

[Opens a text editor on success]
```

b. Examine the **ServiceProfile** and find the **HEAD /authors/{id}.json** routes. Adding **isRetryable: true** to that configuration will tell Linkerd that it should retry requests to that route. Edit the **ServiceProfile** so it matches the **isRetryable: true** configuration below. Save and exit the editor to apply the edit.

```
...
  - condition:
      method: HEAD
      pathRegex: /authors/[^/]*\.json
    isRetryable: true
    name: HEAD /authors/{id}.json

serviceprofile.linkerd.io/authors.booksapp.svc.cluster.local
edited
```

18. You have now configured retries for **HEAD** requests to **/authors/{id}.json** in the **authors** service. After giving it some time, check **linkerd routes** to confirm that the success rate has now climbed back up to 100%.

```
yourname@ubuntu-vm:~$ linkerd viz routes -n booksapp
service/webapp

ROUTE                          SERVICE   SUCCESS     RPS
LATENCY_P50    LATENCY_P95    LATENCY_P99
GET /                          webapp    100.00%    0.9rps          16ms
      25ms          29ms
GET /authors/{id}              webapp    100.00%    0.9rps          8ms
      15ms          19ms
```

```
GET /books/{id}             webapp   100.00%   1.7rps        8ms
     10ms         10ms
POST /authors               webapp   100.00%   0.9rps        13ms
     19ms         20ms
POST /authors/{id}/delete   webapp   100.00%   0.9rps        18ms
     29ms         30ms
POST /authors/{id}/edit     webapp         -     -           -
     -            -
POST /books                 webapp   100.00%   1.7rps        15ms
     20ms         28ms
POST /books/{id}/delete     webapp   100.00%   0.9rps        8ms
     10ms         10ms
POST /books/{id}/edit       webapp   100.00%   0.9rps        18ms
     72ms         94ms
[DEFAULT]                   webapp         -     -           -
     -            -
```

19. Congratulations! You have successfully mitigated an issue for your users without having to make any code changes. Your team now has time to debug and find the right changes to make, and does not have to rush to push out a bug fix!

20. While this is exciting for your users, digging in a little deeper will show you the effect it is having on your system. If you pass the **`-o wide`** flag to **`linkerd routes`**, it will show you that the **`EFFECTIVE_SUCCESS`** and **`ACTUAL_SUCCESS`** rates are quite different. In the sample output below, **`EFFECTIVE_SUCCESS`** is 100.00% but **`ACTUAL_SUCCESS`** is 48.14%. This is because the Linkerd sidecar proxy in the books service is now sending more requests to the authors service when a request fails. You can see this in the **`EFFECTIVE_RPS`** and **`ACTUAL_RPS`** values (2.6rps and 5.4 rps, respectively, in the sample output).

```
yourname@ubuntu-vm:~$ linkerd viz routes -n booksapp deploy/books
--to service/authors -o wide
```

```
ROUTE                       SERVICE   EFFECTIVE_SUCCESS
EFFECTIVE_RPS   ACTUAL_SUCCESS   ACTUAL_RPS   LATENCY_P50
LATENCY_P95   LATENCY_P99
DELETE /authors/{id}.json   authors                    -
     -               -          -          -          -
-
GET /authors.json           authors                    -
     -               -          -          -          -
-
```

```
GET /authors/{id}.json      authors             -
      -                -           -           -           -
-
HEAD /authors/{id}.json     authors           100.00%
2.6rps           48.14%     5.4rps            3ms         10ms
17ms
POST /authors.json          authors             -
      -                -           -           -           -
-
[DEFAULT]                   authors             -
      -                -           -           -           -
-
```

21. The other effect this has had is it slightly increases the latency of adding books. Running **`linkerd routes`** will show that there is a relatively high P95 latency.

```
yourname@ubuntu-vm:~$ linkerd viz routes -n booksapp
deploy/webapp --to svc/books

ROUTE                       SERVICE   SUCCESS     RPS
LATENCY_P50   LATENCY_P95   LATENCY_P99
DELETE /books/{id}.json     books    100.00%   0.9rps          8ms
      10ms         10ms
GET /books.json             books    100.00%   1.8rps          3ms
      5ms          5ms
GET /books/{id}.json        books    100.00%   2.6rps          2ms
      3ms          3ms
POST /books.json            books    100.00%   1.6rps          14ms
      19ms         20ms
PUT /books/{id}.json        books    100.00%   0.8rps          16ms
      82ms         97ms
[DEFAULT]                   books       -     -           -
-          -
```

22. This issue, like retries, can be mitigated by Linkerd until your team can fully resolve the issue.

    a. Run **`kubectl edit`** to open the **`ServiceProfile`** in a text editor.

    ```
    yourname@ubuntu-vm:~$ kubectl edit -n booksapp
    serviceprofile books.booksapp.svc.cluster.local

    [Opens a text editor on success]
    ```

b. Find the **PUT** route you need to configure retries on, and add the **timeout: 25ms** configuration below. This will cause attempts to time out more quickly. Save and exit the editor to apply the edit.

```
...
  - condition:
      method: PUT
      pathRegex: /books/[^/]*\.json
    name: PUT /books/{id}.json
    timeout: 25ms

serviceprofile.linkerd.io/books.booksapp.svc.cluster.local
edited
```

23. Wait a few minutes and run **linkerd routes** again to see the latest data. The latency numbers include time spent in the webapp itself, so they will remain above the 25 ms threshold you set. You can see that the timeouts are working because the **EFFECTIVE_SUCCESS** rate drops as the linkerd proxy times out on some requests. In the example output below, that rate is 92.16%. There are always tradeoffs when mitigating issues with microservices, and you must decide if setting around a 90% success rate with moderate latency is a reasonable way to leave the system while your team works on a fix for the service.

```
yourname@ubuntu-vm:~$ linkerd -n booksapp viz routes
deploy/webapp --to svc/books -o wide

ROUTE                      SERVICE   EFFECTIVE_SUCCESS
EFFECTIVE_RPS   ACTUAL_SUCCESS   ACTUAL_RPS   LATENCY_P50
LATENCY_P95   LATENCY_P99
DELETE /books/{id}.json    books            100.00%
0.9rps          100.00%    0.9rps           8ms        10ms
10ms
GET /books.json            books            100.00%
1.9rps          100.00%    1.9rps           3ms        4ms
5ms
GET /books/{id}.json       books            100.00%
2.7rps          100.00%    2.7rps           2ms        3ms
3ms
POST /books.json           books            100.00%
1.6rps          100.00%    1.6rps           14ms       19ms
20ms
PUT /books/{id}.json       books            92.16%
0.8rps          100.00%    0.8rps           11ms       77ms
96ms
```

```
[DEFAULT]                    books               -
-                -           -         -         -         -
```

24. You’ve reached the end of this lab. You will continue to use the Linkerd cluster in your next lab, so leave it running if you plan to move on to the next lab now. Otherwise, stop the Linkerd cluster by running this command:

```
yourname@ubuntu-vm:~$ docker stop $(docker ps -a -f
name=linkerd-control-plane -q)

8af4c08524df
```

Lab 7.2 - Debugging the Linkerd Mesh

Overview

In this lab, you'll simulate a failure in the mesh so you can try some tools to debug issues with Linkerd itself. The number of components that make up a service mesh can make debugging it hard. Having an understanding of the tools you can use to identify the root causes of issues in your application will help you in times of trouble.

For the purposes of this lab, you can use the debug sidecar.

1. If you don't still have an active session from the previous Linkerd lab, perform these steps to reestablish it:

    a. Connect to the VM that hosts your Kubernetes clusters.

    b. To start the cluster that contains the Linkerd service mesh, issue this command:

    ```
    yourname@ubuntu-vm:~$ docker start $(docker ps -a -f
    name=linkerd-control-plane -q)
    ```

    ```
    8af4c08524df
    ```

    c. To switch the **`kind`** context to the Linkerd cluster, use this command:

    ```
    yourname@ubuntu-vm:~$ kubectl config use-context
    kind-linkerd
    ```

    ```
    Switched to context "kind-linkerd".
    ```

2. You will continue to use the booksapp application from the previous lab in this lab. Issue the command below to make sure all of the **`Pod`**s in your cluster have a status of **`Running`** and a ready status with the same numbers on the left and right sides of the

slash (“/”) before continuing. If any of the lines don’t show those statuses, wait a few minutes and reissue the command again.

```
yourname@ubuntu-vm:~$ kubectl get pods --all-namespaces

NAMESPACE                 NAME
     READY   STATUS   RESTARTS   AGE
          1/1   Running   0            7h
kube-system               coredns-66bff467f8-9m6v8
     1/1   Running   0            7h
linkerd                   linkerd-controller-6df458948d-n7tvb
     2/2   Running   0            7h
booksapp                  authors-7fb885df4c-4m2s8
     2/2   Running   0            3m23s
booksapp                  books-75b8b95bcf-76zwz
     2/2   Running   0            3m23s
booksapp                  traffic-7f4758fcb4-ffpwr
     2/2   Running   0            3m23s
booksapp                  webapp-86596d7887-f5ff7
     2/2   Running   0            3m23s
linkerd                   linkerd-destination-94bcf958f-mtvnq
     2/2   Running   0            7h
[additional output omitted]
```

3. Before you simulate an issue with the mesh, issue the command below and look at the valid output of **`linkerd check --proxy`**. The **`--proxy`** flag tells linkerd to only check the status of the data plane proxies that are actually handling requests in your application.

```
yourname@ubuntu-vm:~$ linkerd check --proxy

kubernetes-api
--------------
√ can initialize the client
√ can query the Kubernetes API

kubernetes-version
------------------
√ is running the minimum Kubernetes API version
√ is running the minimum kubectl version

linkerd-existence
-----------------
√ 'linkerd-config' config map exists
```

```
√ heartbeat ServiceAccount exist
√ control plane replica sets are ready
√ no unschedulable pods
√ controller pod is running
√ can initialize the client
√ can query the control plane API

linkerd-config
--------------
√ control plane Namespace exists
√ control plane ClusterRoles exist
√ control plane ClusterRoleBindings exist
√ control plane ServiceAccounts exist
√ control plane CustomResourceDefinitions exist
√ control plane MutatingWebhookConfigurations exist
√ control plane ValidatingWebhookConfigurations exist
√ control plane PodSecurityPolicies exist

linkerd-identity
----------------
√ certificate config is valid
√ trust anchors are using supported crypto algorithm
√ trust anchors are within their validity period
√ trust anchors are valid for at least 60 days
√ issuer cert is using supported crypto algorithm
√ issuer cert is within its validity period
√ issuer cert is valid for at least 60 days
√ issuer cert is issued by the trust anchor

linkerd-identity-data-plane
---------------------------
√ data plane proxies certificate match CA

linkerd-api
-----------
√ control plane pods are ready
√ control plane self-check
√ [kubernetes] control plane can talk to Kubernetes
√ [prometheus] control plane can talk to Prometheus
√ tap api service is running

linkerd-version
---------------
√ can determine the latest version
√ cli is up-to-date
```

```
linkerd-data-plane
------------------
√ data plane namespace exists
√ data plane proxies are ready
√ data plane proxy metrics are present in Prometheus
√ data plane is up-to-date
√ data plane and cli versions match

linkerd-addons
--------------
√ 'linkerd-config-addons' config map exists

linkerd-grafana
---------------
√ grafana add-on service account exists
√ grafana add-on config map exists
√ grafana pod is running

Status check results are √
```

4. Now you will simulate an error in the control plane and then see how to debug it. The Linkerd control plane runs in the **`linkerd`** namespace and has many components that are used to run Linkerd. Take a look at the [control plane architecture]() on the Linkerd website to get a better understanding of what each component is responsible for.

5. Re-install the Emoji demo app:

```
yourname@ubuntu-vm:~$ curl --proto '=https' --tlsv1.2 -sSfL
https://run.linkerd.io/emojivoto.yml \ | kubectl apply -f -
```

6. Delete the Books app ingress:

```
yourname@ubuntu-vm:~$ kubectl delete ingress ingress-booksapp -n
booksapp
```

7. Set up the Nginx Ingress to listen to the Emoji app again as in the previous lab, you pointed it to the Books app.

```
yourname@ubuntu-vm:~$ kubectl apply -f - <<EOF
—
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
```

```
      namespace: emojivoto
      name: ingress-emojivoto
      annotations:
        ingress.kubernetes.io/rewrite-target: /
        nginx.ingress.kubernetes.io/service-upstream: "true"
        consul.hashicorp.com/connect-inject: "true"
    spec:
      ingressClassName: nginx
      rules:
        - http:
            paths:
            - pathType: Prefix
              path: "/"
              backend:
                service:
                  name: web-svc
                  port:
                    number: 80
     EOF
```

8. Next, deploy the debug sidecar container.

```
yourname@ubuntu-vm:~$ kubectl -n emojivoto get deploy/voting -o yaml \
  | linkerd inject --enable-debug-sidecar - \
  | kubectl apply -f -

deployment "voting" injected
deployment.apps/voting configured
```

9. Confirm that the debug container is running.

```
yourname@ubuntu-vm:~$ kubectl get pods -n emojivoto -l app=voting-svc \
  -o jsonpath='{.items[*].spec.containers[*].name}'

linkerd-proxy voting-svc linkerd-debug%
```

10. You can now watch the live tshark output from the logs.

```
yourname@ubuntu-vm:~$ kubectl -n emojivoto logs deploy/voting linkerd-debug -f
```

```
          2208 145.565533637  10.244.0.34 → 10.244.0.9   TLSv1.3 141
Application Data
      2209 145.565583804   10.244.0.9 → 10.244.0.34  TCP 68 35942 →
4191 [ACK] Seq=2410 Ack=49750 Win=64128 Len=0 TSval=1352785030
TSecr=1709083926
      2210 145.565612554  10.244.0.34 → 10.244.0.9   TLSv1.3 3580
Application Data
      2211 145.565633804   10.244.0.9 → 10.244.0.34  TCP 68 35942 →
4191 [ACK] Seq=2410 Ack=53262 Win=63232 Len=0 TSval=1352785030
TSecr=1709083926
      2212 145.960358304  10.244.0.30 → 10.244.0.34  GRPC 131
HEADERS[235]: POST /emojivoto.v1.VotingService/VoteTaco, DATA[235]
(GRPC) (PROTOBUF)
      2213 145.962068263  10.244.0.34 → 10.244.0.34  GRPC 158
HEADERS[235]: POST /emojivoto.v1.VotingService/VoteTaco, DATA[235]
(GRPC) (PROTOBUF)
      2214 145.962636138  10.244.0.34 → 10.244.0.34  HTTP2 98
WINDOW_UPDATE[0], PING[0]
      2215 145.962643221  10.244.0.34 → 10.244.0.34  TCP 68 50742 →
8080 [ACK] Seq=13375 Ack=7407 Win=65536 Len=0 TSval=3942159986
TSecr=3942159986
      2216 145.962707013  10.244.0.34 → 10.244.0.34  HTTP2 85 PING[0]
      2217 145.963630388  10.244.0.34 → 10.244.0.34  GRPC 104
HEADERS[235]: 200 OK, DATA[235], HEADERS[235] (GRPC) (PROTOBUF)
      2218 145.963636721  10.244.0.34 → 10.244.0.34  TCP 68 50742 →
8080 [ACK] Seq=13392 Ack=7443 Win=65536 Len=0 TSval=3942159987
TSecr=3942159987
      2219 145.963942096  10.244.0.34 → 10.244.0.30  GRPC 128
HEADERS[235]: 200 OK, DATA[235], HEADERS[235] (GRPC) (PROTOBUF)
      2220 145.963979013  10.244.0.30 → 10.244.0.34  TCP 68 35066 →
8080 [ACK] Seq=13441 Ack=8441 Win=64256 Len=0 TSval=1570900962
TSecr=3623533114
      2221 145.964110179  10.244.0.30 → 10.244.0.34  HTTP2 98
WINDOW_UPDATE[0], PING[0]
      2222 145.964734971  10.244.0.34 → 10.244.0.30  HTTP2 85 PING[0]
      2223 145.964758721  10.244.0.30 → 10.244.0.34  TCP 68 35066 →
8080 [ACK] Seq=13471 Ack=8458 Win=64256 Len=0 TSval=1570900963
TSecr=3623533115
```

11. Another option is to Exec directly into the container itself so you can run a tshark command while inside of the container. You can then view and see any errors that may be occurring inside of your application.

```
yourname@ubuntu-vm:~$ kubectl -n emojivoto exec -it \
  $(kubectl -n emojivoto get pod -l app=voting-svc \
    -o jsonpath='{.items[0].metadata.name}') \
  -c linkerd-debug -- tshark -i any -f "tcp" -V -Y "http.request"
```

```
            Interface name: any
        Encapsulation type: Linux cooked-mode capture (25)
        Arrival Time: Jul 28, 2022 12:39:53.187339506 UTC
        [Time shift for this packet: 0.000000000 seconds]
        Epoch Time: 1659011993.187339506 seconds
        [Time delta from previous captured frame: 0.000132792
seconds]
        [Time delta from previous displayed frame: 0.000000000
seconds]
        [Time since reference or first frame: 9.286449546 seconds]
        Frame Number: 154
        Frame Length: 176 bytes (1408 bits)
        Capture Length: 176 bytes (1408 bits)
        [Frame is marked: False]
        [Frame is ignored: False]
        [Protocols in frame: sll:ethertype:ip:tcp:http]
    Linux cooked capture
        Packet type: Unicast to us (0)
        Link-layer address type: 1
        Link-layer address length: 6
        Source: 46:08:50:cf:cf:17 (46:08:50:cf:cf:17)
        Unused: 0000
        Protocol: IPv4 (0x0800)
```

12. You’ve reached the end of this lab. Stop the Linkerd cluster by running this command:

```
yourname@ubuntu-vm:~$ docker stop $(docker ps -a -f
name=linkerd-control-plane -q)

8af4c08524df
```