JTC10 Istio Mesh Networking

Lab0 - Lab information

Istio a joint collaboration between IBM, Google and Lyft provides an easy way to create a service mesh that will manage many of these complex tasks automatically, without the need to modify the microservices themselves.

In this Lab you will learn the basic operations for setting up and operating an Istio Mesh Network.

> **IMPORTANT: For this lab you need a PC/Mac with at least 16GB of RAM and you have to configure your VM with at least 12GB of RAM**
>
> **If you do not have a PC that matches these specifications you can follow the steps described here to setup a Kubernetes instance on IBM Cloud.**

Lab sources

All the source code for the lab is available here:

https://github.com/niklaushirt/training

Lab overview

- In this Lab you will learn the basics of Istio:

1. Get to know Istio
2. Install Istio
3. Deploy the Bookinfo Demo App
4. Monitoring with Kiali
5. Traffic flow management
6. Telemetry
7. Authentication

8. Traffic Mirroring
9. Fault Injection
10. Clean Up

Lab0 - Lab semantics

Nomenclatures

Shell Commands

The commands that you are going to execute to progress the Labs will look like this:

THIS IS AN EXAMPLE - DO NOT EXECUTE THIS!

```Bash
kubectl create -f redis-slave-service.yaml

 > Output Line 1
 > Output Line 2
 > Output Line 3
 ...
```

> **IMPORTANT NOTE:** The example output of a command is prefixed by ">" in order to make it more distinguishable.
>
> So in the above example you would only enter/copy-paste `kubectl create -f redis-slave-service.yaml` and the output from the command is "Output Line 1" to "Output Line 3"

Code Examples

Code examples are presented like this:

```YAML
apiVersion: lab.ibm.com/v1beta1
kind: MyResource
metadata:
  name: example
spec:
  size: 3
  image: busybox
```

This is only for illustration and is not being actively used in the Labs.

Lab 0 - Prepare the Lab environment

Before starting the Labs, let's make sure that we have the latest source code from the GitHub repository:

https://github.com/niklaushirt/training

1. Open a Terminal window by clicking on the Termnial icon in the left sidebar - we will use this extensively later as well

2. Execute the following commands to initialize your Training Environment

   ```bash
   ./welcome.sh
   ```

   This will

   - pull the latest example code from my GitHub repository
   - start minikube if not already running
   - installs the registry
   - installs the Network Plugin (Cilium)
   - starts the Personal Training Environment

   > During this you will have to provide a name (your name) that will be used to show your progress in the Instructor Dashboard in order to better assist you.

Lab 1 - Get to know ISTIO

Microservices and containers changed application design and deployment patterns, but along with them brought challenges like service discovery, routing, failure handling, and visibility to microservices. "Service mesh" architecture was born to handle these features. Applications are getting decoupled internally as microservices, and the responsibility of maintaining coupling between these microservices is passed to the service mesh.

Istio, a joint collaboration between IBM, Google and Lyft provides an easy way to create a service mesh that will manage many of these complex tasks automatically, without the need to modify the microservices themselves. Istio does this by:

1. Deploying a **control plane** that manages the overall network infrastructure and enforces the policy and traffic rules defined by the devops team

2. Deploying a **data plane** which includes “sidecars”, secondary containers that sit along side of each instance of a microservice and act as a proxy to intercept all incoming and outgoing network traffic. Sidecars are implemented using Envoy, an open source edge proxy

Once Istio is installed some of the key feature which it makes available include

- Traffic management using **Istio Pilot**: In addition to providing content and policy based load balancing and routing, Pilot also maintains a canonical representation of services in the mesh.

- Access control using **Istio Auth**: Istio Auth secures the service-to-service communication and also provides a key management system to manage keys and certificates.

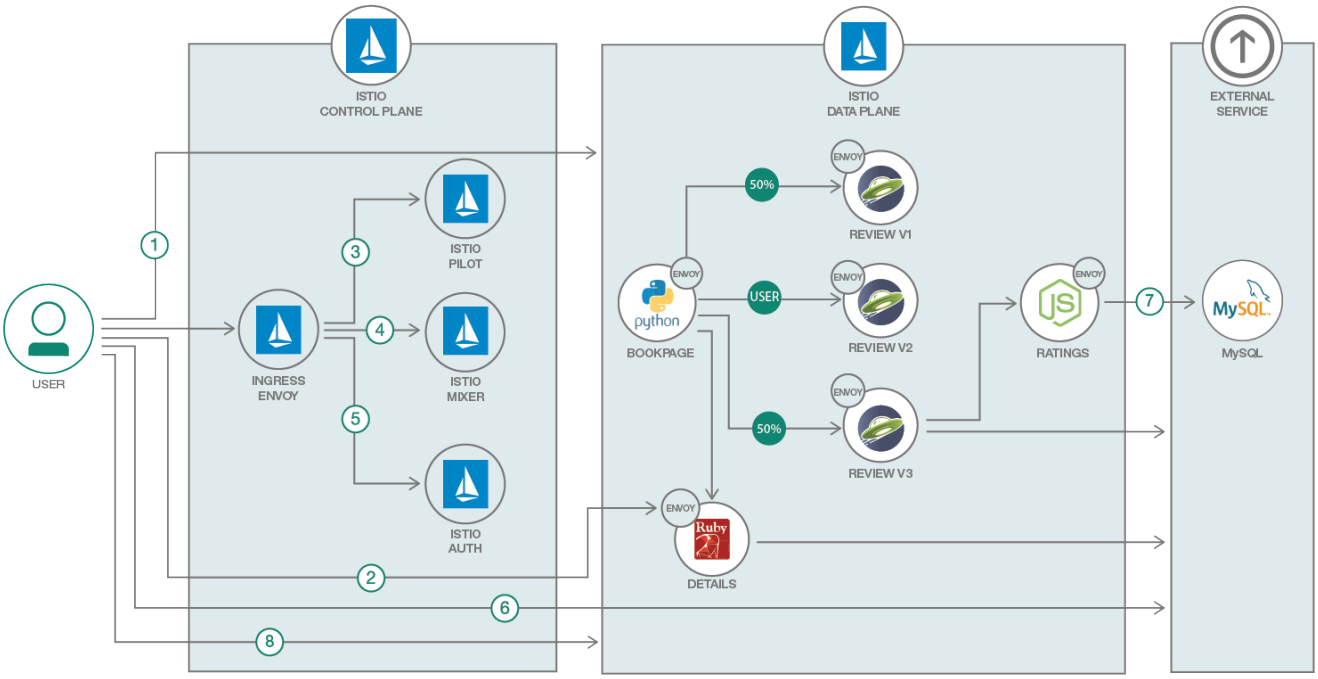
- Monitoring, reporting and quota management using **Istio Mixer**: Istio Mixer provides in depth monitoring and logs data collection for microservices, as well as collection of request traces. Precondition checking like whether the service consumer is on whitelist, quota management like rate limits etc. are also configured using Mixer.

In the first part of this journey we show how we can deploy the sample BookInfo application and inject sidecars to get the Istio features mentioned above, and walk through the key ones. The BookInfo is a simple application that is composed of four microservices, written in different languages for each of its microservices namely Python, Java, Ruby, and Node.js. The application does not use a database, and stores everything in local filesystem.

Also since Istio tightly controls traffic routing to provide above mentioned benefits, it introduces some drawbacks. Outgoing traffic to external services outside the Istio data plane can only be enabled by specialized configuration, based on the protocol used to connect to the external service.

In the second part of the journey we focus on how Istio can be configured to allow applications to connect to

external services. For that we modify the sample BookInfo application to use an external database and then use it as a base to show Istio configuration for enabling egress traffic.



Included Components

- Istio
- Kiali
- Grafana
- Jaeger
- Prometheus

Lab 2 - Installing Istio

In this module, you download and install Istio.

1. Execute the following command to get the latest ISTIO source:

```bash
cd
wget https://github.com/istio/istio/releases/download/1.6.4/istio-1.6.4-linux-amd64.tar.gz
tar xfvz istio-1.6.4-linux-amd64.tar.gz
```

2. Add the `istioctl` client to your executables.

```bash
export PATH=./istio-1.6.4/bin:$PATH
```

3. Install Istio into the cluster:

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

> 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-third-party-service-account-tokens for details.
> ✔ Istio core installed

> - Processing resources for Istiod. Waiting for Deployment/istio-system/istiod
> ...
```

4. Label the default namespace for automatic sidecar injection

```bash
kubectl label namespace default istio-injection=enabled


> namespace/default labeled
```

5. Check the labels

```
Bash
kubectl get ns --show-labels


> NAME                   STATUS   AGE   LABELS
> default                Active   66d   istio-injection=enabled
> istio-system           Active   62s   istio-injection=disabled,istio-operato
r-managed=Reconcile,operator.istio.io/    > component=Base,operator.istio.io/m
anaged=Reconcile,operator.istio.io/version=1.4.0
> kube-node-lease        Active   66d   <none>
> kube-public            Active   66d   <none>
> kube-system            Active   66d   <none>
> kubernetes-dashboard   Active   66d   addonmanager.kubernetes.io/mode=Reconc
ile,kubernetes.io/minikube-addons=dashboard
```

6. Ensure the corresponding pods are all in **`Running`** state before you continue.

```
Bash
kubectl get pods -n istio-system

> NAME                                    READY   STATUS    RESTARTS   AGE
> grafana-5dc4b4676c-dsdb5                1/1     Running   0          21m
> istio-egressgateway-5c7967cf9d-gnrpj    1/1     Running   0          21m
> istio-ingressgateway-676fbf789d-vldbf   1/1     Running   0          21m
> istio-tracing-8584b4d7f9-cj9p6          1/1     Running   0          21m
> istiod-55cd8455dd-4gq2j                 1/1     Running   0          23m
> kiali-6f457f5964-pvjw5                  1/1     Running   0          21m
> prometheus-7f54ff69cd-lf4xz             2/2     Running   0          21m
```

7. Patch the NodePort of the Istio Ingress to 30762.

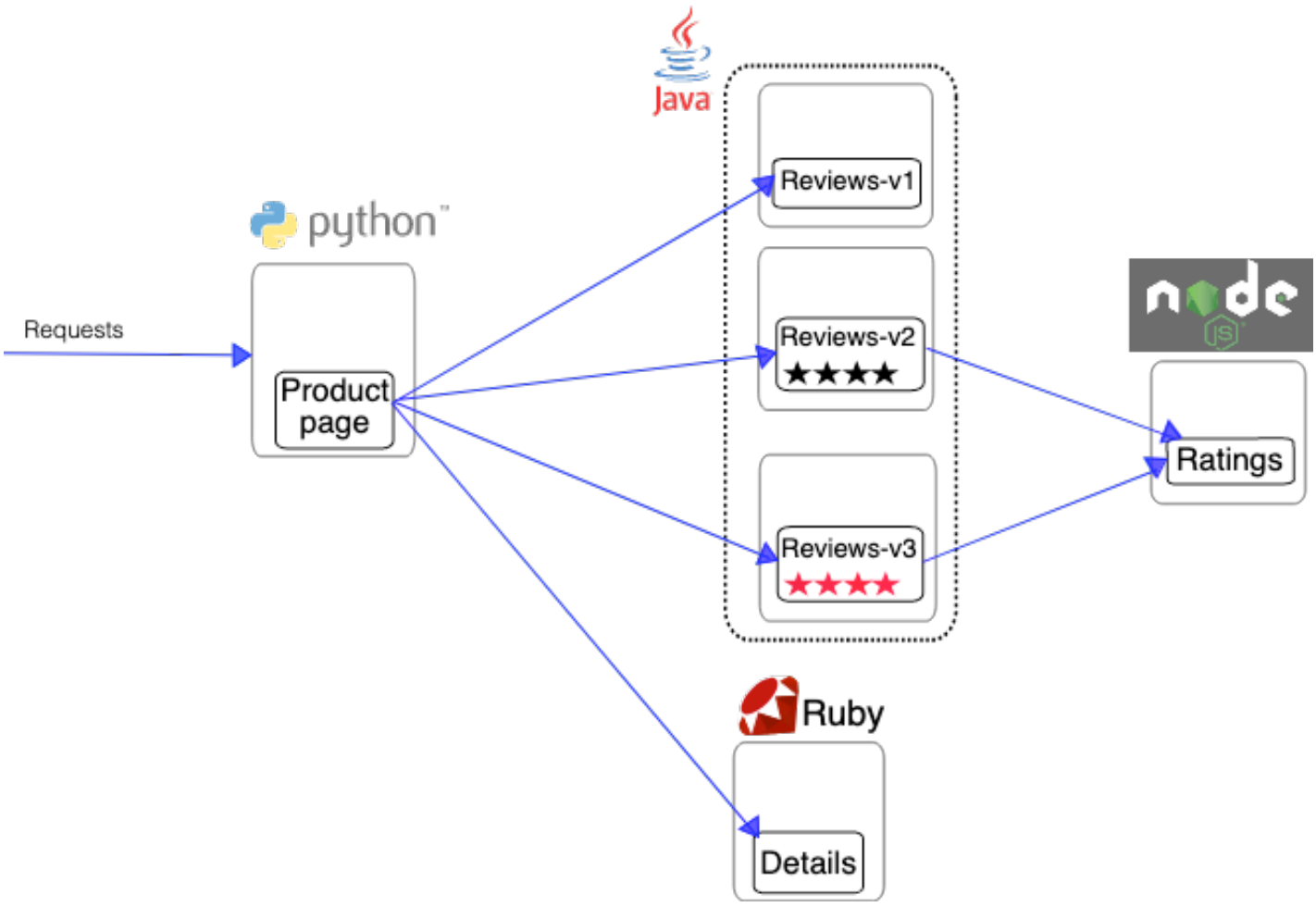
```
Bash
kubectl patch service -n istio-system istio-ingressgateway -p '{"spec": {"port
s": [{"nodePort": 30762,"port":80,"name": "http2"}],"type": "NodePort"}}'

 > service/istio-ingressgateway patched
```

Before you continue, make sure all the pods are deployed and are either in the **`Running`** or **`Completed`** state. If they're in `Pending` or `CrashLoopBackOff` state, wait a few minutes to let the deployment to settle.

Congratulations! You successfully installed Istio into your cluster.

Lab 3 - Deploy sample BookInfo application with Istio sidecar injected

In this part, we will be using the sample BookInfo Application that comes as default with Istio code base. As mentioned above, the application that is composed of four microservices, written in different languages for each of its microservices namely Python, Java, Ruby, and Node.js. The default application doesn't use a database and all the microservices store their data in the local file system.



Envoys are deployed as sidecars on each microservice. Injecting Envoy into your microservice means that the Envoy sidecar would manage the ingoing and outgoing calls for the service. To inject an Envoy sidecar to an existing microservice configuration, do:

```bash
kubectl apply -f ~/training/istio/samples/bookinfo/platform/kube/bookinfo.yaml

> service/details created
> serviceaccount/bookinfo-details created
> deployment.apps/details-v1 created
> service/ratings created
> ...
```

After a few minutes, you should now have your Kubernetes Pods running and have an Envoy sidecar in each of them alongside the microservice. The microservices are **productpage, details, ratings, and reviews**. Note that you'll have three versions of the reviews microservice.

```bash
kubectl get pods


> NAME                                READY     STATUS    RESTARTS   AGE
> details-v1-1520924117-48z17         2/2       Running   0          6m
> productpage-v1-560495357-jk1lz      2/2       Running   0          6m
> ratings-v1-734492171-rnr5l          2/2       Running   0          6m
> reviews-v1-874083890-f0qf0          2/2       Running   0          6m
> reviews-v2-1343845940-b34q5         2/2       Running   0          6m
> reviews-v3-1813607990-8ch52         2/2       Running   0          6m
```

Expose the Application via Ingress Gateway

The components deployed on the service mesh by default are not exposed outside the cluster. External access to individual services so far has been provided by creating an external load balancer or node port on each service.

An Ingress Gateway resource can be created to allow external requests through the Istio Ingress Gateway to the backing services.

Create an Istio ingress gateway to access your services over a public IP address.

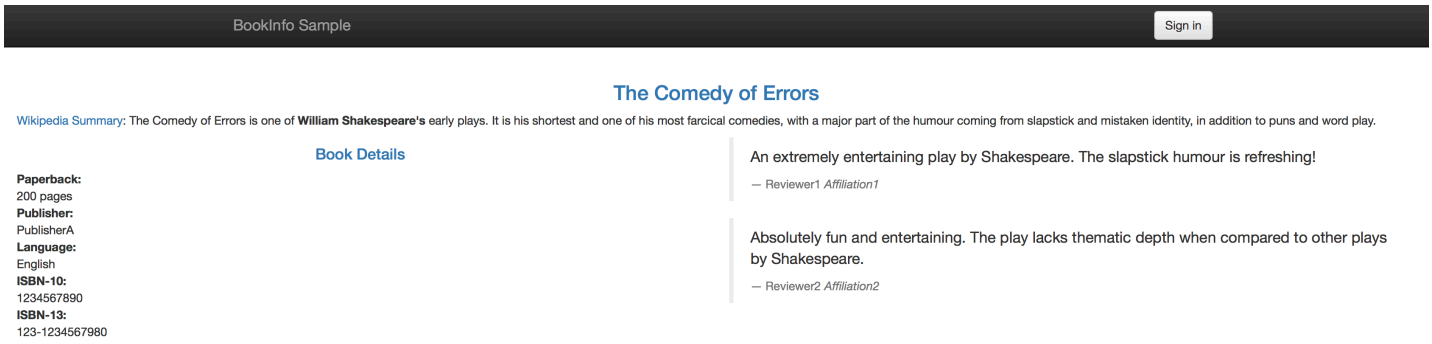
```bash
kubectl apply -f  ~/training/istio/samples/bookinfo/networking/bookinfo-gateway.yaml

> gateway.networking.istio.io/bookinfo-gateway created
> virtualservice.networking.istio.io/bookinfo created
```

Now you can access your application via:

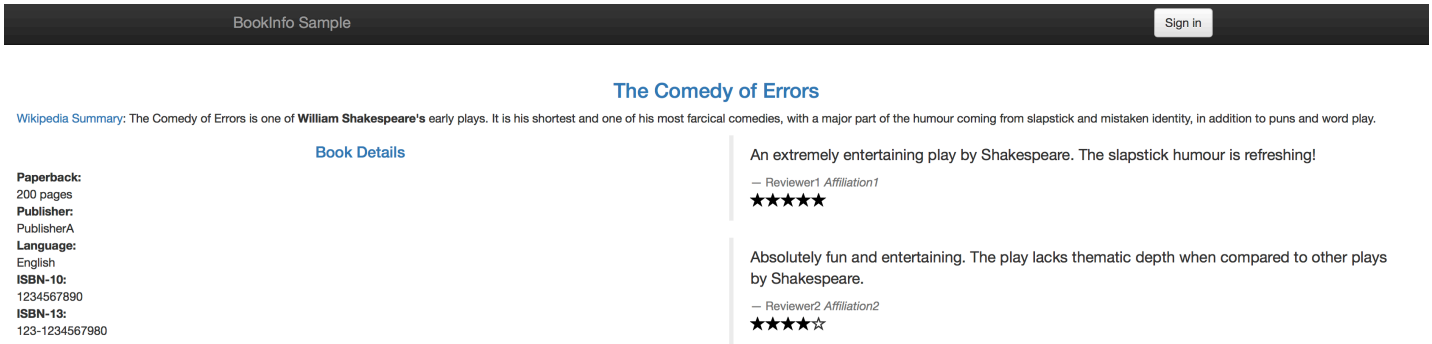
```bash
firefox --new-tab http://$(minikube ip):30762/productpage
```

If you refresh the page multiple times, you'll see that the *reviews* section of the page changes. That's because there are 3 versions of **reviews***(reviews-v1, reviews-v2, reviews-v3)* deployment for our **reviews** service. Istio's load-balancer is using a round-robin algorithm to iterate through the 3 instances of this service

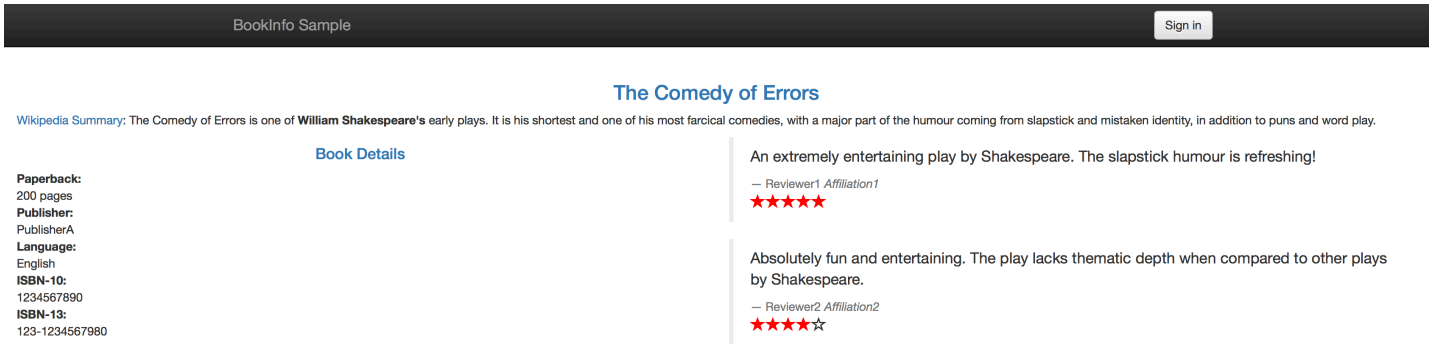
V1 - No ratings



V2 - Ratings with black stars



V3- Ratings with red stars



Sidecar injection

In Kubernetes, a sidecar is a utility container in the pod, and its purpose is to support the main container. For Istio to work, Envoy proxies must be deployed as sidecars to each pod of the deployment. There are two ways of injecting the Istio sidecar into a pod: manually using the istioctl CLI tool or automatically using the Istio Initializer. In this exercise, we will use the manual injection. Manual injection modifies the controller configuration, e.g. deployment. It does this by modifying the pod template spec such that all pods for that deployment are created with the injected sidecar.

Lab 4 - Monitoring with Kiali

Kiali is an open-source project that installs on top of Istio to visualize your service mesh. It provides deeper insight into how your microservices interact with one another, and provides features such as circuit breakers and request rates for your services

In order to create some more sustained traffic, open a new tab in the Terminal and paste the following code

```bash
for i in `seq 1 200000`; do curl http://$(minikube ip):30762/productpage; done
```

Note

> If you get this:
>
> for i in `seq 1 200000` ; do curl http://:30762/productpage\; done for>
>
> Just delete the backslash \ after productpage.

You can open Kiali with

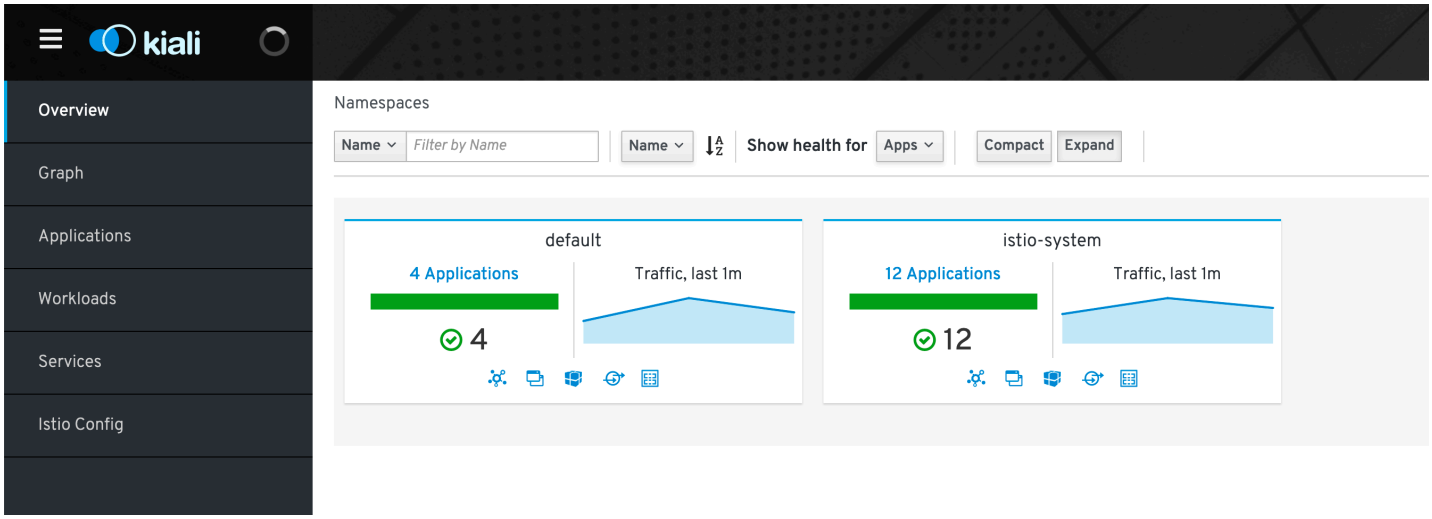
```bash
istioctl dashboard kiali > /dev/null &
```

> Just hit enter a few times to get back the prompt if neeed.

Login is:

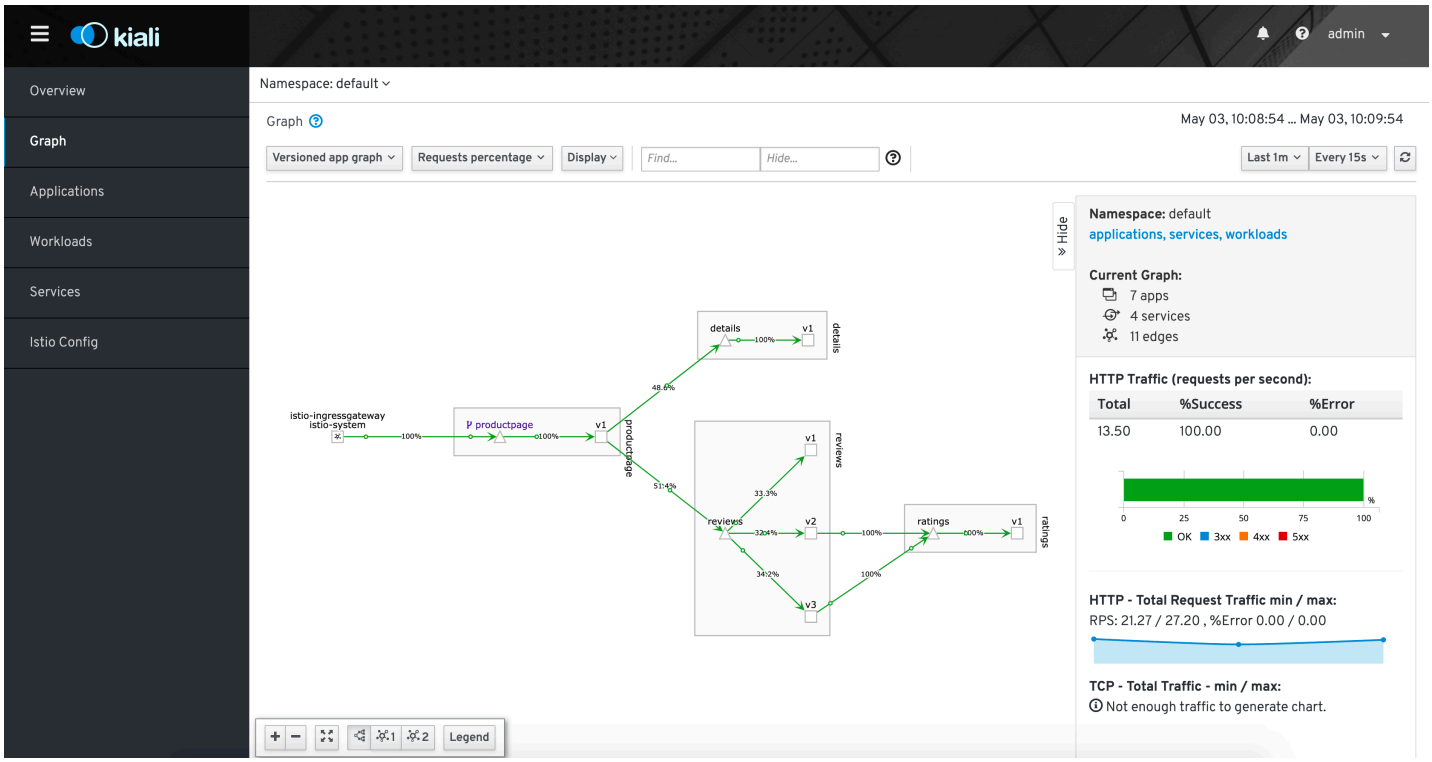
```
User: admin
Password: admin
```

And you should now see the Dashboard



1. Select `Graph` in the left hand menu.
2. Then select `default` from the Namespaces drop-down menu
3. Make sure that you select `Versoned App Graph` from the second drop-down menu
4. Select `Requests Percentage` and `Traffic Animation` from the `Display drop-down menu
5. And make sure that you check all types from the `Display` drop-down menu
6. If you see no objects, there might be no traffic flowing through your mesh network yet (you chan change the seeting to display inactive objects too)
7. You might want to change the refresh rate to 10s to more easily observe the changes in traffic

You can then observe traffic flowing through your mesh network.



Get more info on the Kiali website.

Visualizing Your Mesh

Lab 5 - Traffic flow management using Istio Pilot

Using rules to manage traffic

The core component used for traffic management in Istio is Pilot, which manages and configures all the Envoy proxy instances deployed in a particular Istio service mesh. It lets you specify what rules you want to use to route traffic between Envoy proxies, which run as sidecars to each service in the mesh. Each service consists of any number of instances running on pods, containers, VMs etc. Each service can have any number of versions (a.k.a. subsets). There can be distinct subsets of service instances running different variants of the app binary. These variants are not necessarily different API versions. They could be iterative changes to the same service, deployed in different environments (prod, staging, dev, etc.). Pilot translates high-level rules into low-level configurations and distributes this config to Envoy instances. Pilot uses three types of configuration resources to manage traffic within its service mesh: Virtual Services, Destination Rules, and Service Entries.

Virtual Services

A VirtualService defines a set of traffic routing rules to apply when a host is addressed. Each routing rule defines matching criteria for traffic of a specific protocol. If the traffic is matched, then it is sent to a named destination service (or subset or version of it) defined in the service registry.

Destination Rules

A DestinationRule defines policies that apply to traffic intended for a service after routing has occurred. These rules specify configuration for load balancing, connection pool size from the sidecar, and outlier detection settings to detect and evict unhealthy hosts from the load balancing pool. Any destination `host` and `subset` referenced in a `VirtualService` rule must be defined in a corresponding `DestinationRule`.

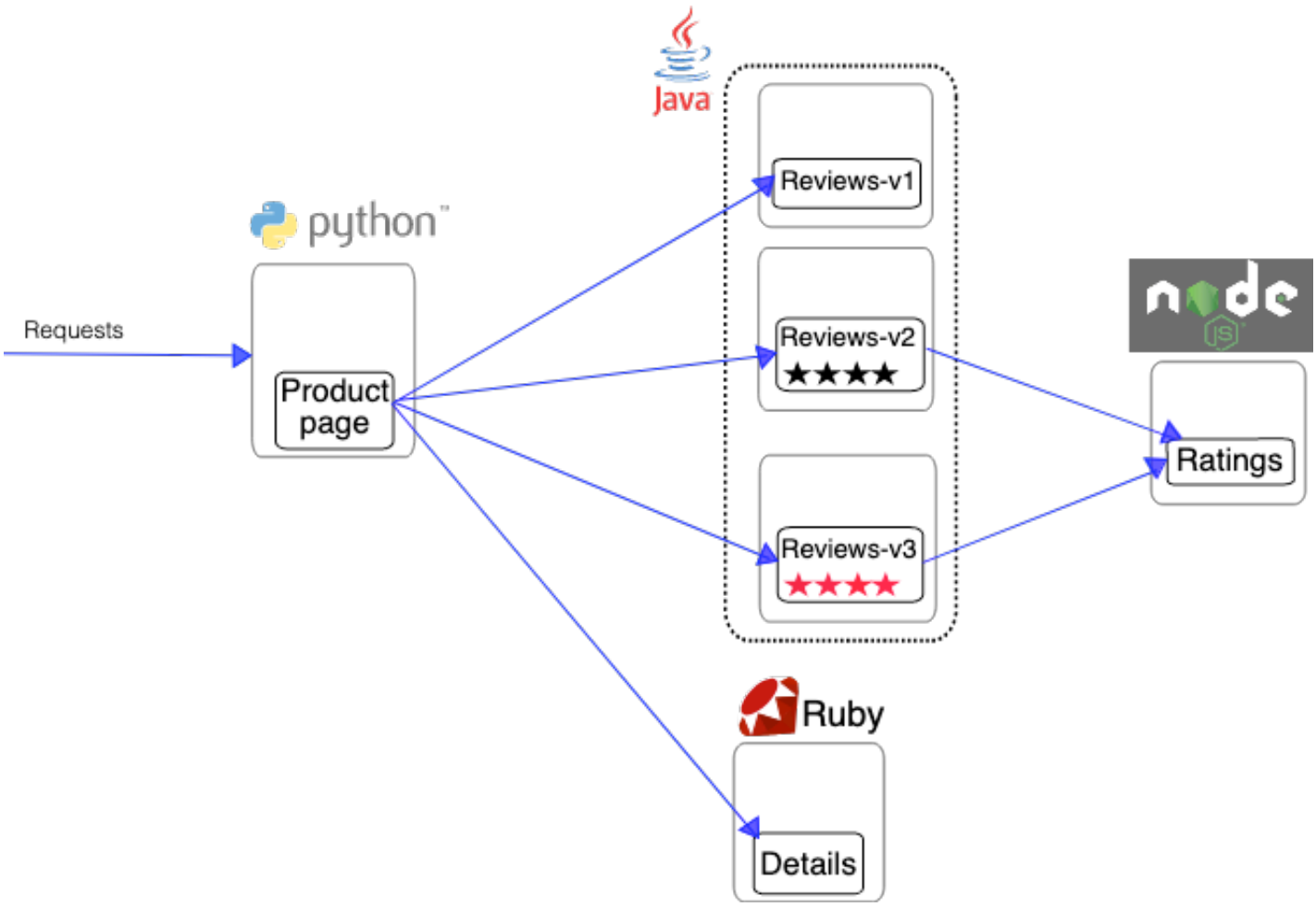
Service Entries

A ServiceEntry configuration enables services within the mesh to access a service not necessarily managed by Istio. The rule describes the endpoints, ports and protocols of a white-listed set of mesh-external domains and IP blocks that services in the mesh are allowed to access.

The Bookinfo Application

In this section, Istio will be configured to dynamically modify the network traffic between some of the components of our application. In this case we have 2 versions of the “reviews” component (v1 and v2) but

we don’t want to replace review-v1 with review-v2 immediately. In most cases, when components are upgraded it’s useful to deploy the new version but only have a small subset of network traffic routed to it so that it can be tested before the old version is removed. This is often referred to as “canary testing”.



There are multiple ways in which we can control this routing. It can be based on which user or type of device that is accessing it, or a certain percentage of the traffic can be configured to flow to one version.

This step shows you how to configure where you want your service requests to go based on weights and HTTP Headers. You would need to be in the root directory of the Istio release you have downloaded on the Prerequisites section.

Set default Destination Rules

Before moving on, we have to define the destination rules. The destination rules tell Istio what versions (subsets in Istio terminology) are available for routing. This step is required before fine-grained traffic shaping is possible.

```bash
kubectl apply -f ~/training/istio/samples/bookinfo/networking/destination-rule-reviews.yaml

> destinationrule.networking.istio.io/reviews created
```

For more details, see the Istio documentation.

A/B testing with Istio

A/B testing is a method of performing identical tests against two separate service versions in order to determine which performs better.

Set Default Routes to `reviews-v1` for all microservices

This would set all incoming routes on the services (indicated in the line `destination: <service>`) to the deployment with a tag `version: v1`. To set the default routes, run:

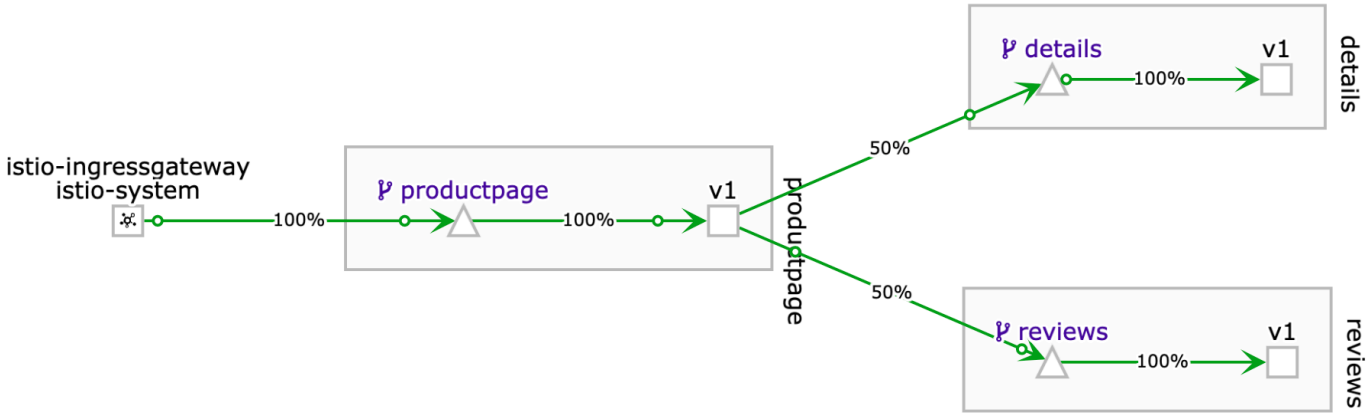
```bash
  kubectl create -f ~/training/istio/samples/bookinfo/networking/virtual-service-all-v1.yaml

  > virtualservice.networking.istio.io/reviews created
```

The definition yaml file that we have just applied looks like this

```yaml
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
  - reviews
  http:
  - route:
    - destination:
        host: reviews
        subset: v1
```

Observe in the Kiali Dashboard. After a short while you should see that all traffic is going to V1.

This may take some time to settle!



After the deployment of v2:

Route 100% of the traffic to the `version: v2` of the **reviews microservices**

This will direct/switch all incoming traffic to version v2 of the reviews microservice. Run:

Bash

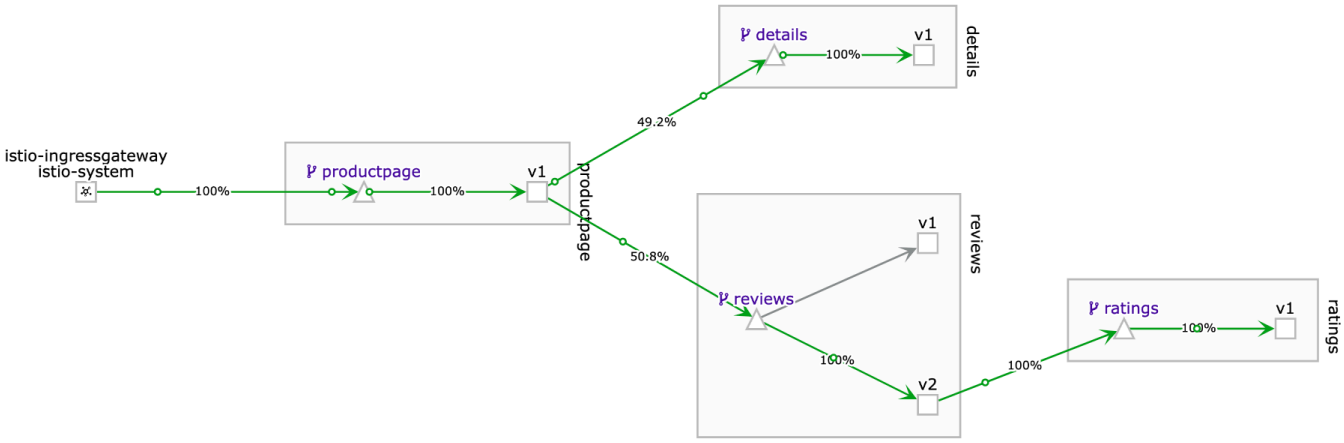
```
  kubectl apply -f ~/training/istio/samples/bookinfo/networking/virtual-service-a
ll-v2.yaml

  > virtualservice.networking.istio.io/reviews configured
```

The new definition yaml file that we have just applied looks like this

```
```yaml
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
  - reviews
  http:
  - route:
    - destination:
        host: reviews
            subset: v2
```
```

Observe in the Kiali Dashboard. After a short wile you should see that all traffic is going to V2.



Canary deployment

In `Canary Deployments`, newer versions of services are incrementally rolled out to users to minimize the risk and impact of any bugs introduced by the newer version. To begin incrementally routing traffic to the newer version of the guestbook service, modify the original `VirtualService` rule.

Route 80% of traffic on **reviews microservice** to `reviews-v1` and 20% to `reviews-v2`.

This is indicated by the `weight: 80 and 20` in the yaml file.

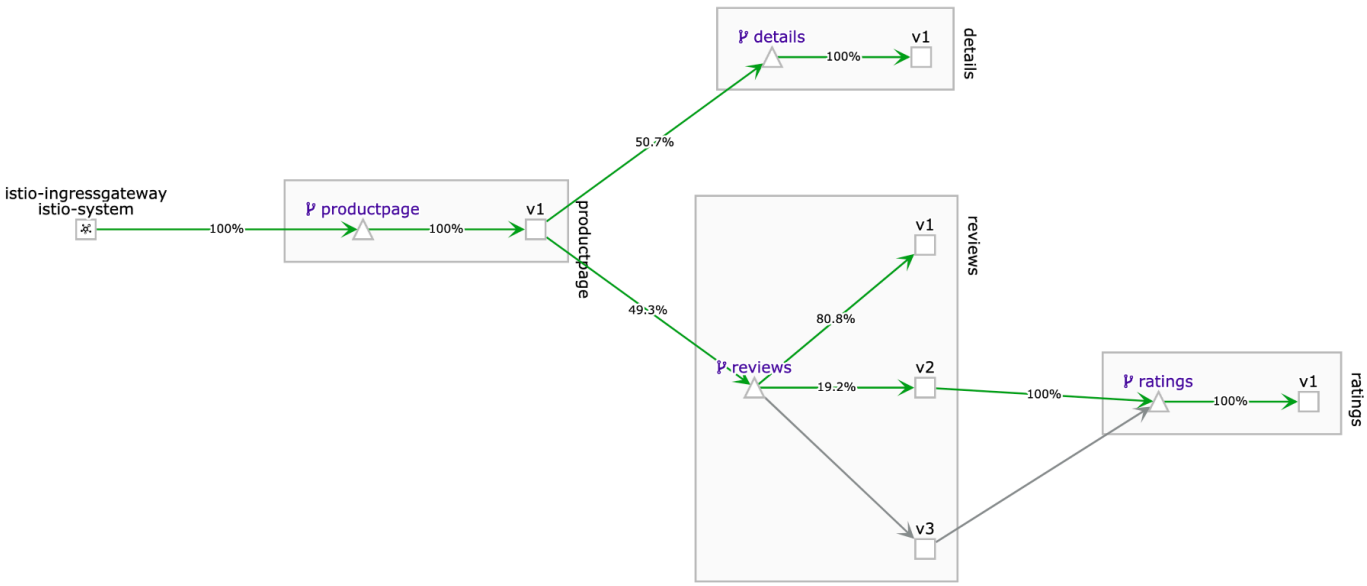
> Using `replace` should allow you to edit existing route-rules.

```Bash
  kubectl apply -f ~/training/istio/samples/bookinfo/networking/virtual-service-r
eviews-80-20.yaml
```

The new definition yaml file that we have just applied looks like this

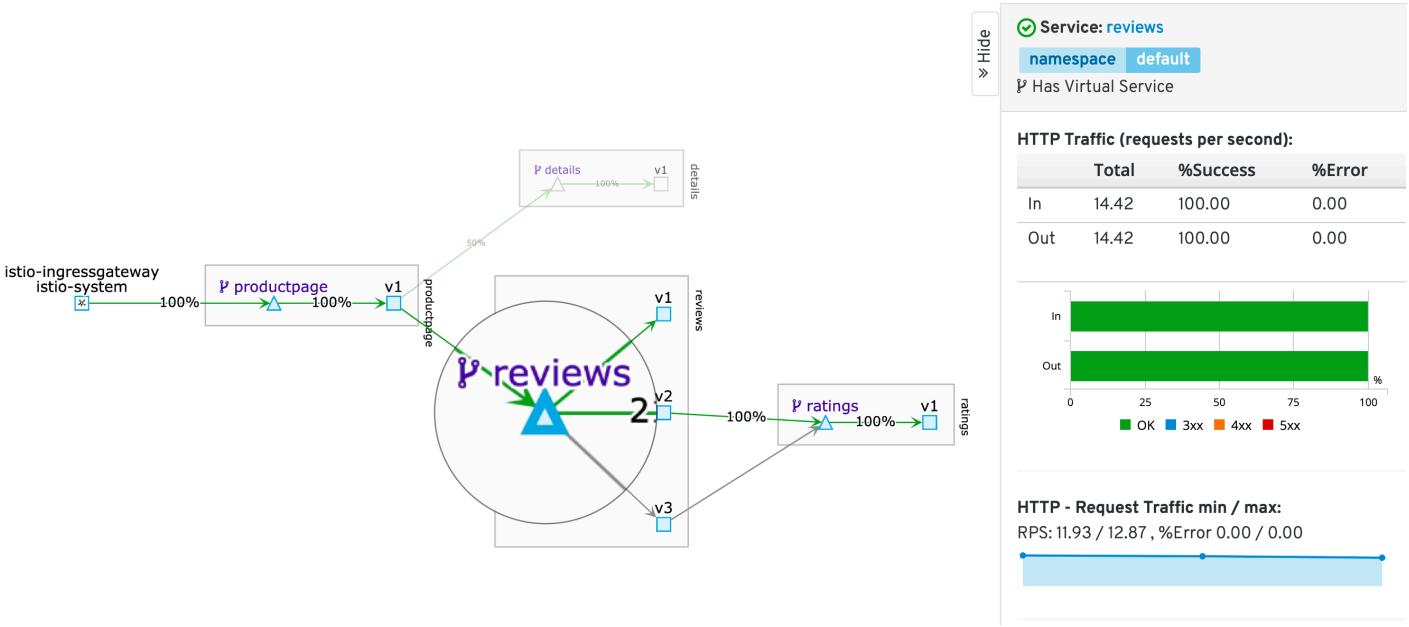
```yaml
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
    - reviews
  http:
  - route:
    - destination:
        host: reviews
        subset: v1
      weight: 80
    - destination:
        host: reviews
        subset: v2
      weight: 20
```

Observe in the Kiali Dashboard. After a short wile you should see that 80% of the traffic is going to V1 and 80% of the traffic is going to V2.

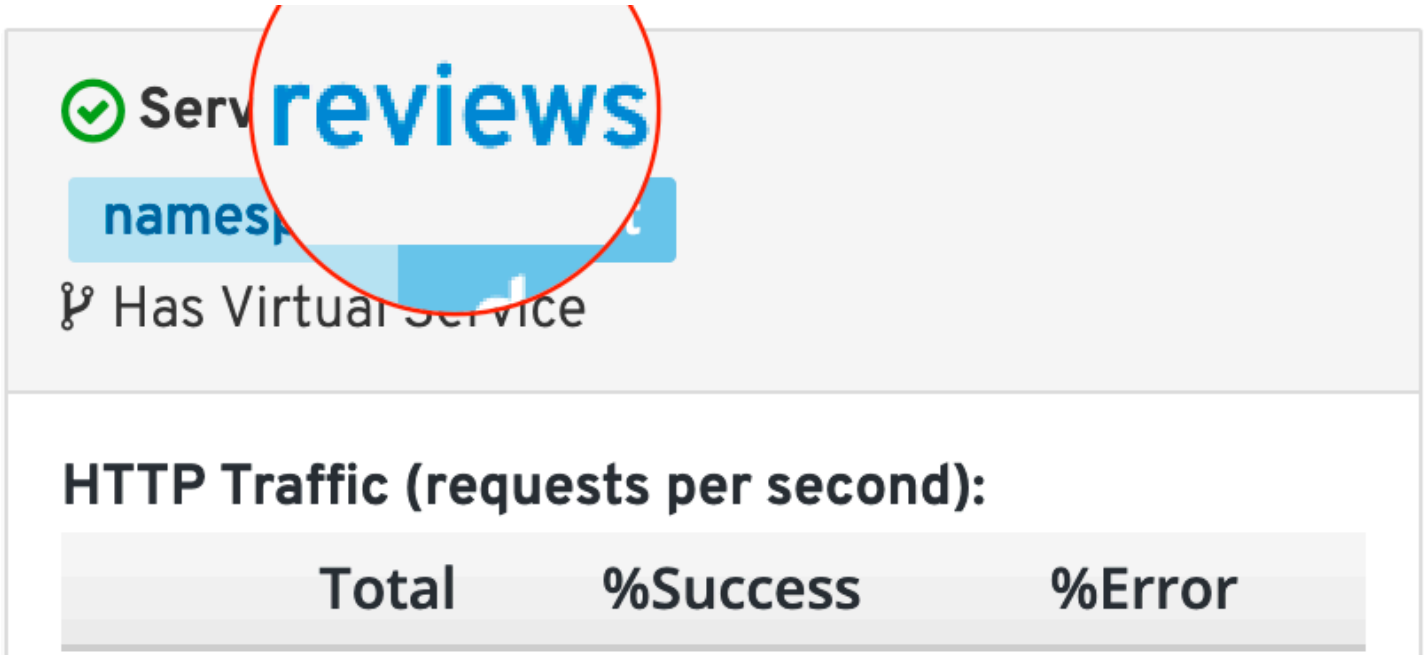


Navigating Kiali

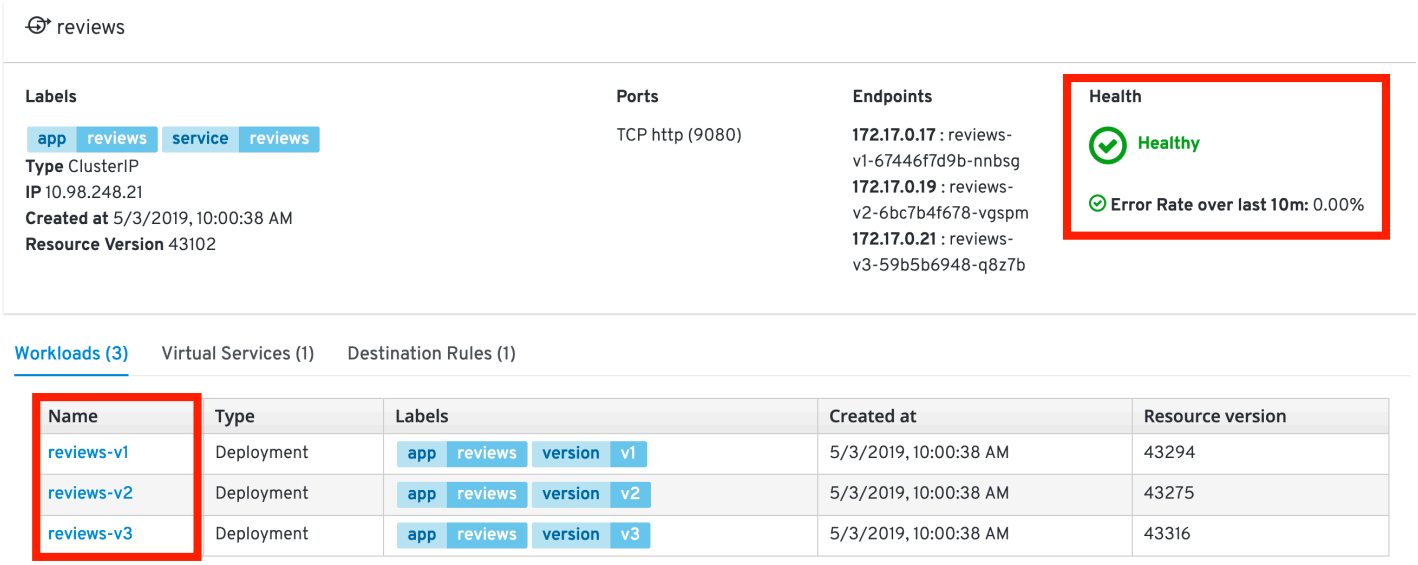
Go back to the Kiali Dashboard and select the `reviews` service in the graph.



To the right you can observe specific metrics for this service. Then open the `reviews` service overview:

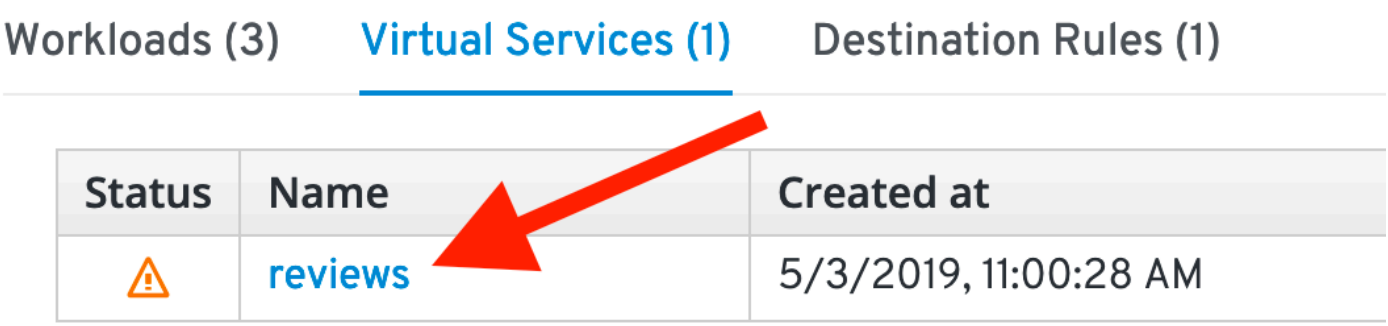


In this view you can get details about the service, like averall Health, assigned `Workloads` and much more.

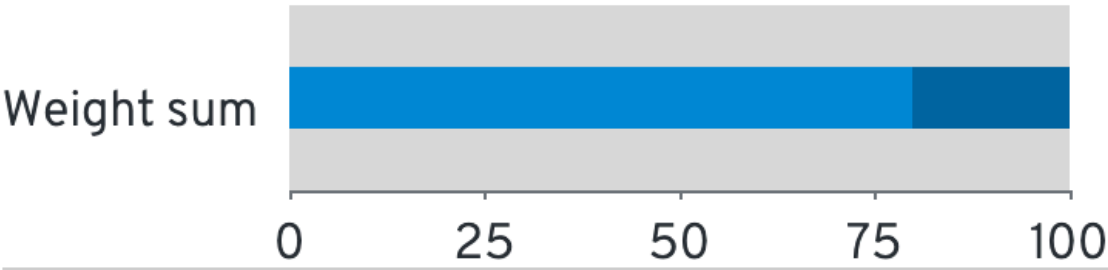


Feel free to browse some more to get familiar with the interface.

Now open the `reviews` service details by selecting the `Virtual Services` tab and selecting `reviews`.



Here you get more detailed information about the service, like the weight distribution:



Status	Destination			Weights
	Host	Subset	Port	
	reviews	v1	-	80
	reviews	v2	-	20

Gradual Rollout

In order to gradually roll out a new release we have to change the weight distribution.

- Click on the `YAML` tab
- Modify the weight to 20%/80%
- Click `Save`

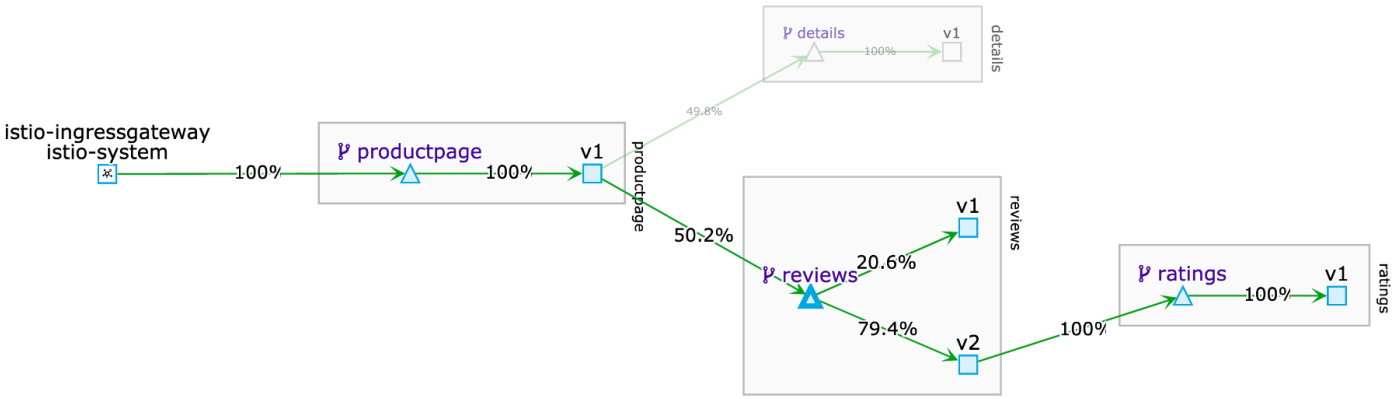
Istio Config > Namespace: default > Istio Object Type: virtualservices > Istio Object: reviews

Overview YAML *

```
  namespace: default
  selfLink: >-
    /apis/networking.istio.io/v1alpha3/namespaces/default/virtualservices/reviews
  uid: e5af95d6-6d81-11e9-8674-080027cac4d9
  resourceVersion: '49294'
  generation: 6
  creationTimestamp: '2019-05-03T09:00:28Z'
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: >
      {"apiVersion":"networking.istio.io/v1alpha3","kind":"VirtualService","metadata":{"annotation
spec:
  hosts:
    - reviews
  gateways: ~
  http:
    - route:
        - destination:
            host: reviews
            subset: v1
          weight: 20
        - destination:
            host: reviews
            subset: v2
          weight: 80
  tcp: ~
  tls: ~
```

Save Reload Cancel

Observe in the Kiali Dashboard. After a short wile you should see that about 20% of the traffic is going to V1 and 90% of the traffic is going to V2.



Note: The sum of the weights must be equal to 100%

Traffic Steering / Dark Launch

Define certain conditions (Username, type of phone, ...) that will be using the new service.

Set Route to `reviews-v2` of **reviews microservice** for a specific user

This would set the route for the user `jason` (You can login as *jason* with any password in your deploy web application) to see the `version: v3` of the reviews microservice.

Run:

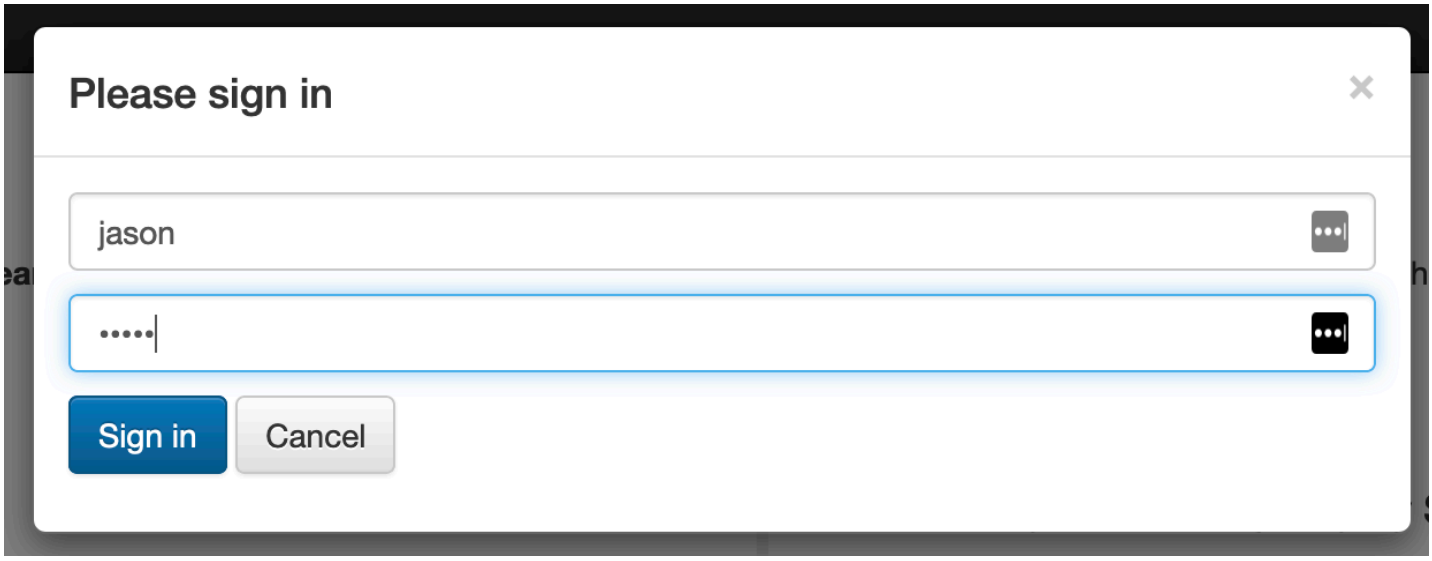
```bash
  kubectl apply -f ~/training/istio/samples/bookinfo/networking/virtual-service-reviews-jason-v2-v3.yaml
```

The new definition yaml file that we have just applied looks like this

```yaml
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
  - reviews
  http:
  - match:
    - headers:
        end-user:
          exact: jason
    route:
    - destination:
        host: reviews
        subset: v3
  - route:
    - destination:
        host: reviews
        subset: v2
```

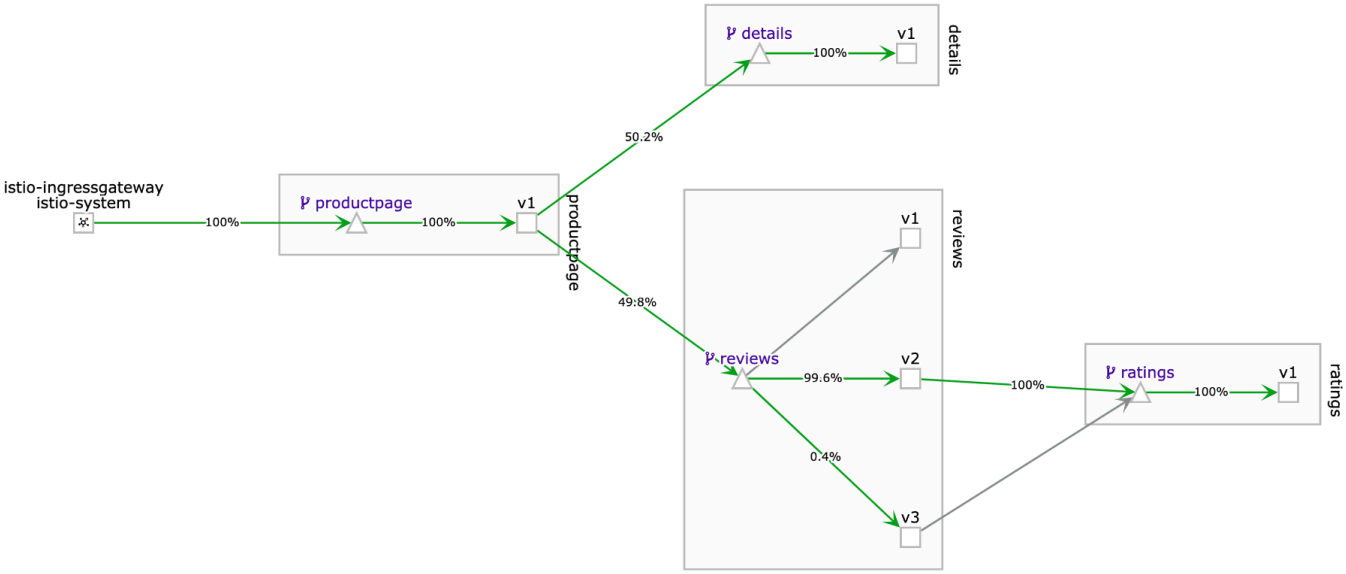
Go to the Bookinfo Application: `http://<MINIKUBE_IP>:30762/productpage` (replace with the address of your cluster).

Refresh several times - You should see only black stars, meaning that you are using V2.



Now login to the Web Application as user jason with password jason and **refresh several times**. You should see only red stars, meaning that you are using V3.

Observe in the Kiali Dashboard. After a short wile you should see that a small percentage of traffic is going to V3, which corresponds to your page refreshes.



Lab 6 - Telemetry data aggregation - metrics, logs and trace spans

Challenges with microservices

We all know that microservice architecture is the perfect fit for cloud native applications and it increases the delivery velocities greatly. Envision you have many microservices that are delivered by multiple teams, how do you observe the the overall platform and each of the service to find out exactly what is going on with each of the services? When something goes wrong, how do you know which service or which communication among the few services are causing the problem?

Istio telemetry

Istio's tracing and metrics features are designed to provide broad and granular insight into the health of all services. Istio's role as a service mesh makes it the ideal data source for observability information, particularly in a microservices environment. As requests pass through multiple services, identifying performance bottlenecks becomes increasingly difficult using traditional debugging techniques. Distributed tracing provides a holistic view of requests transiting through multiple services, allowing for immediate identification of latency issues. With Istio, distributed tracing comes by default. This will expose latency, retry, and failure information for each hop in a request.

You can read more about how Istio mixer enables telemetry reporting.

Collect metrics and logs using Prometheus and Grafana

This step shows you how to configure Istio Mixer to gather telemetry for services in your cluster.

1. Make sure you still send traffic to that service. You can renew the `for` loop from earlier.

```bash
for i in `seq 1 200000`; do curl http://<MINIKUBE_IP>:30762/productpage; done
```

2. Open the Grafana Dashboard

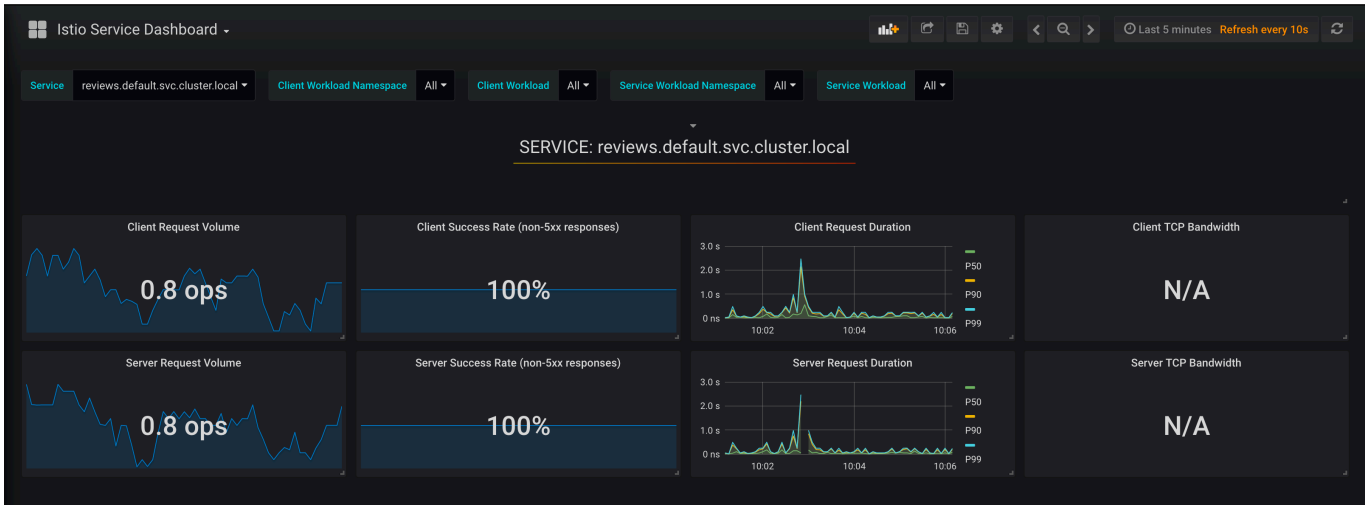
```bash
istioctl dashboard grafana > /dev/null &
```

> Just hit enter a few times to get back the prompt if neeed.

3. Click on the `Home` button in the upper left hand corner

4. Select `Istio Service Dashboard`

Your dashboard should look like this:



Play around and observe the different metrics being collected.

[Collecting Metrics on Istio]

[Collecting Logs on Istio]

Collect request traces using Jaeger

Jaeger is a distributed tracing tool that is available with Istio.

1. Make sure you still send traffic to that service. You can renew the `for` loop from earlier.

   ```bash
   for i in `seq 1 200000`; do curl http://<MINIKUBE_IP>:30762/productpage; done
   ```

2. Access the Jaeger Dashboard

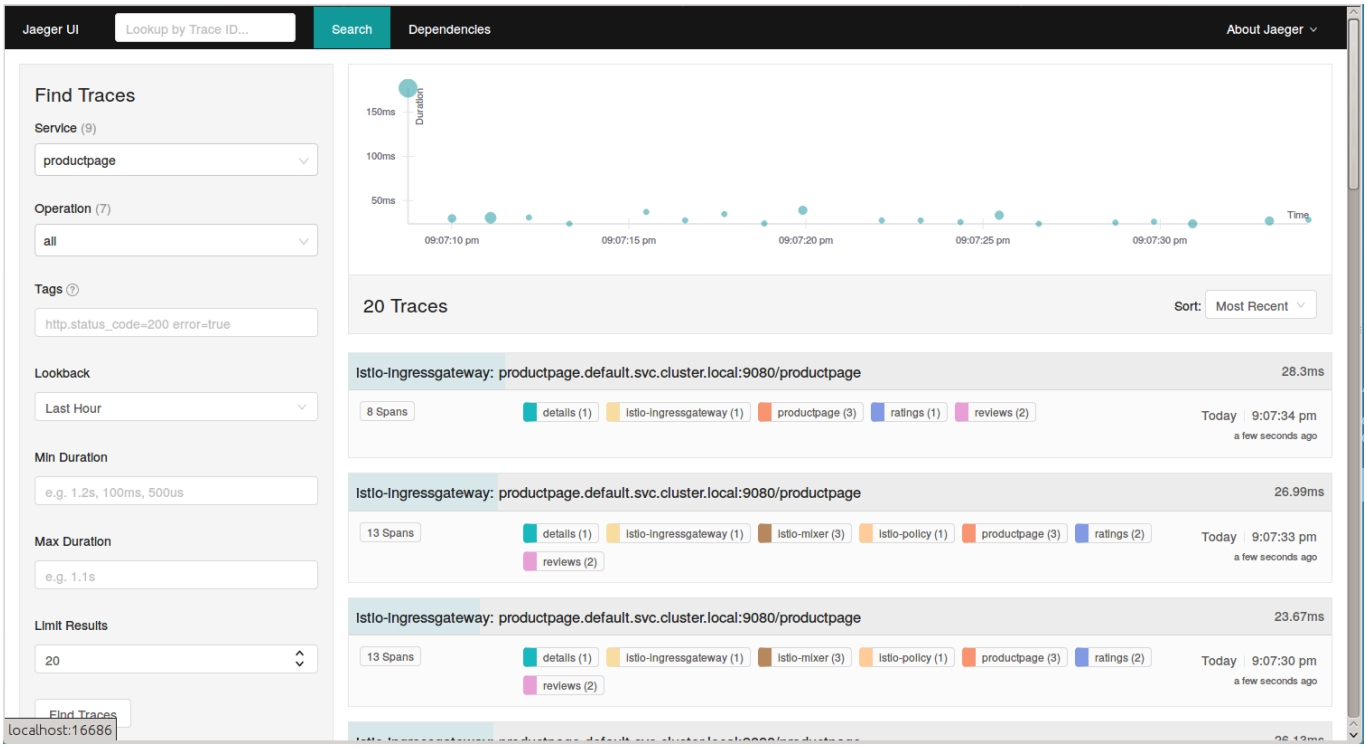
   ```bash
   istioctl dashboard jaeger > /dev/null &
   ```

   > Just hit enter a few times to get back the prompt if neeed.

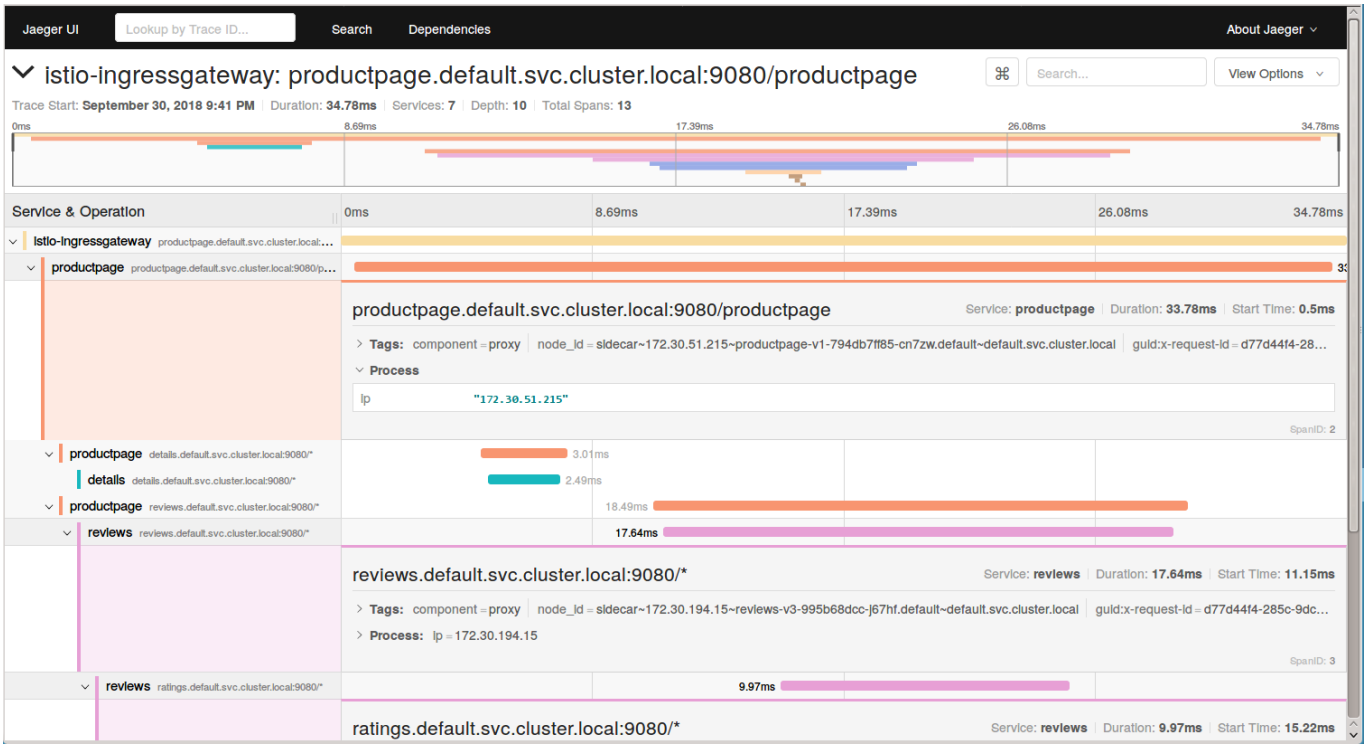
3. Select `productpage.default` in the left hand `Service` dropdown

4. Click `Find Traces`

   Your dashboard should look something like this:



5. Click on one of those traces and you will see the details of the traffic you sent to your BookInfo App. It shows how much time it took for the request on `productpage` to finish. It also shows how much time it took for the requests on the `details`, `reviews`, and `ratings` services.



[Jaeger Tracing on Istio]

Lab 7 - End-user authentication

Istio provides two types of authentication:

- **Peer authentication**: used for service-to-service authentication to verify the client making the connection. Istio offers mutual TLS for transport authentication, which can be enabled without requiring service code changes.
    - Provides each service with a strong identity representing its role to enable interoperability across clusters and clouds.
    - Secures service-to-service communication.
    - Provides a key management system to automate key and certificate generation, distribution, and rotation.
- **Request authentication**: Used for end-user authentication to verify the credential attached to the request. Istio enables request-level authentication with JSON Web Token (JWT) validation and custom authentication via any OpenID Connect providers, for example:
    - Keycloak
    - Auth0
    - Google Auth

Istio stores the authentication policies in Customresources.

Request authentication policies specify the values needed to validate a JSON Web Token (JWT). These values include, among others, the following:

- The location of the token in the request
- The issuer or the request
- The public JSON Web Key Set (JWKS)

Istio checks the presented token, if presented against the rules in the request authentication policy, and rejects requests with invalid tokens. **When requests carry no token, they are accepted by default**. To reject requests without tokens, you have to provide authorization rules that specify the restrictions for specific operations.

You can find mor information here: https://istio.io/latest/docs/tasks/security/authentication/authn-policy/

For this lab we use the test tokens from https://raw.githubusercontent.com/istio/istio/release-1.6/security/tools/jwt/samples/.

Lab 7 - Request Authentication

1. Try to access the productpage:

   ```bash
   export INGRESS_HOST=http://$(minikube ip):30762
   curl $INGRESS_HOST/productpage -s -o /dev/null -w "%{http_code}\n"

   > 200
   ```

   You should get an HTTP Code 200 - which means that the access is working. You can also try to refresh the productpage, it should load perfectly.

2. Create the `RequestAuthentication` Object

   ```bash
   kubectl apply -f - <<EOF
   apiVersion: "security.istio.io/v1beta1"
   kind: "RequestAuthentication"
   metadata:
     name: "jwt-example"
     namespace: istio-system
   spec:
     selector:
       matchLabels:
         istio: ingressgateway
     jwtRules:
     - issuer: "testing@secure.istio.io"
       jwksUri: "https://raw.githubusercontent.com/istio/istio/release-1.6/security
   EOF

   > requestauthentication.security.istio.io/jwt-example created
   ```

   Here we create the RequestAuthentication Object based on the test/demo tokens provided by Istio: https://raw.githubusercontent.com/istio/istio/release-1.6/security/tools/jwt/samples/.

   The JSON Web Key Set (JWKS) is a set of keys which contains the public keys used to verify any JSON Web Token (JWT).

3. Try to access the productpage again:

```bash
curl $INGRESS_HOST/productpage -s -o /dev/null -w "%{http_code}\n"

> 200
```

You should still get an HTTP Code 200 - which means that the access is working, because requests that carry no token, are accepted by default.

4. However trying to access the productpage with an invalid token should not work:

```bash
curl --header "Authorization: Bearer my-invalid-token" $INGRESS_HOST/productpa
ge -s -o /dev/null -w "%{http_code}\n"

> 401
```

You should get an HTTP Code 401 (Unauthorized) - which means that the access is blocked if you use an invalid token.

Lab 7 - Autorisation Policy

1. Create the `AutorisationPolicy` So to reject requests that carry no token, we have to provide authorization rules that specify the restrictions for specific operations, for example paths or actions. The following `AuthorizationPolicy` contains a rule specifying a DENY action for requests without request principals:

   Bash

   ```
   kubectl apply -f - <<EOF
   apiVersion: "security.istio.io/v1beta1"
   kind: "AuthorizationPolicy"
   metadata:
     name: "frontend-ingress"
     namespace: istio-system
   spec:
     selector:
       matchLabels:
         istio: ingressgateway
     action: DENY
     rules:
     - from:
       - source:
           notRequestPrincipals: ["*"]
   EOF

   > authorizationpolicy.security.istio.io/frontend-ingress created
   ```

2. Try to access the productpage again:

   Bash

   ```
   curl $INGRESS_HOST/productpage -s -o /dev/null -w "%{http_code}\n"

   > 403
   ```

   You should get an HTTP Code 403 (Forbidden) - which means that the access is now blocked if you provide no token.

Lab 7 - Autorisation with JWT

Now, to validate that the access works with a valid JWT token, we call the URL with the JWT token provided by Istio: https://raw.githubusercontent.com/istio/istio/release-1.6/security/tools/jwt/samples/ that will be validated by the JWKS public key defined in the `RequestAuthentication` .

Bash

```
TOKEN=$(curl https://raw.githubusercontent.com/istio/istio/release-1.6/security/tools/jwt/samples/demo.jwt -s)
curl --header "Authorization: Bearer $TOKEN" $INGRESS_HOST/productpage -s -o /dev/null -w "%{http_code}\n"

> 200
>
```

You should get an HTTP Code 200 - which means that the access is working, because the JWT token has been validated againts the JWKS.

Lab 7 - Autorisation with custom JWT

In the previous example we have been using a pre-created JWT token provided by Istio. In this Lab we are going to create a custom JWT token based on the same private key that has been used for the previous one and that is being validated by the JWKS.

1. Install the needed componenets and libraries

```Bash
sudo apt install python3-pip
pip3 install jwcrypto
```

2. Install the python-based tool to generate valid JWT tokens

```Bash
wget https://raw.githubusercontent.com/istio/istio/release-1.6/security/tools/jwt/samples/gen-jwt.py
chmod +x gen-jwt.py
```

3. Get the private key from the Istio Git repository that we will use to sign the token

```Bash
wget https://raw.githubusercontent.com/istio/istio/release-1.6/security/tools/jwt/samples/key.pem
```

4. Create a JWT token to access the application

```Bash
TOKEN=$(./gen-jwt.py ./key.pem --expire 500000)
echo $TOKEN

curl --header "Authorization: Bearer $TOKEN" $INGRESS_HOST/productpage -s -o /dev/null -w "%{http_code}\n"

> 200
```

   You should get an HTTP Code 200 - which means that the access is working, because our JWT token has been validated againts the JWKS.

5. Create a JWT token with 5 seconds expiry to access the application

```Bash
TOKEN=$(./gen-jwt.py ./key.pem --expire 5)
for i in `seq 1 10`; do curl --header "Authorization: Bearer $TOKEN" $INGRESS_
HOST/productpage -s -o /dev/null -w "%{http_code}\n"; sleep 1; done

> 200
> 200
> 200
> 200
> 200
> 401
> 401
> 401
> 401
...
```

You should get an HTTP Code 200 for about 5 seconds until the token expires, after which you will get a 401 error.

Lab 8 - Traffic Mirroring

Traffic mirroring, also called shadowing, is a powerful concept that provides a risk-free method of testing your releases in the production environment without impacting your end users.

Instead of using traditional pre-production environments which used to be a replica of production, mirroring can provide synthetic traffic to mimic the live environment.

Traffic monitoring works in the following way:

- You deploy a new version of your component (v2)
- The existing version (v1) works loke before but sends an asynchronous copy to the new version
- The new version (v2) processes the incoming traffic but does not respond to the user (fire-and-forget)
- The Dev and Ops teams can monitor the new version in order to identify potential problems before starting the rollout

Let's start the Lab.

1. Create a Virtual Service that sends all traffic to v1 of the `reviews` service

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

   Run the following:

   Bash
   ```
   kubectl apply -f ~/training/istio/samples/bookinfo/networking/virtual-service-all-v1.yaml

   > virtualservice.networking.istio.io/reviews configured
   ```

2. Create a Virtual Service that sends all traffic to v1 and mirrors it to v2 of the `reviews` service

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
  - reviews
  http:
  - route:
    - destination:
        host: reviews
        subset: v1
    mirror:
      host: nginx
      subset: v2
mirror_percent: 100
```

Run the following:

```bash
kubectl apply -f ~/training/istio/samples/bookinfo/networking/virtual-service-mirror-v1-v2.yaml

> virtualservice.networking.istio.io/reviews configured
```

After a short while you should see traffic starting to flow from `reviews` (v2) to `ratings`, because `reviews` (v2) is receiving the mirrored traffic.

Lab 9 - Fault Injection

To test microservices for resiliency, Istio allows us to inject delays and errors between services.

Let's start the Lab.

1. Let's create a VirtualService that creates 50% of 501 errors when the `details` service is called.

   YAML
   ```yaml
   apiVersion: networking.istio.io/v1alpha3
   kind: VirtualService
   metadata:
     name: details
   spec:
     hosts:
     - details
     http:
     - fault:
         abort:
           httpStatus: 500
           percentage:
             value: 50
       route:
       - destination:
           host: details
           subset: v1
   ...
   ```

   Run the following:

   Bash
   ```
   kubectl apply -f ~/training/istio/samples/bookinfo/networking/fault-injection-details-v1.yaml

   > virtualservice.networking.istio.io/details created
   > destinationrule.networking.istio.io/details created
   ```

2. In Kiali click on the productpage (orange) box and after a short while you should see 500 errors appearing. In the right side dtail panel you will see additional information about the problem in your communication between microservices.

Lab 10 - Clean-up

- To delete the BookInfo app and its route-rules:
  `~/training/istio/samples/bookinfo/platform/kube/cleanup.sh`
- To delete Istio from your cluster

```Bash
kubectl delete -f ~/training/istio/samples/bookinfo/platform/kube/bookinfo.yaml
kubectl delete -f  ~/training/istio/samples/bookinfo/networking/bookinfo-gateway.yaml
kubectl delete -f ~/training/istio/samples/bookinfo/networking/destination-rule-reviews.yaml
kubectl delete -f ~/training/istio/samples/bookinfo/networking/virtual-service-reviews-80-20.yaml

istioctl manifest generate --set profile=demo | kubectl delete -f -

kubectl delete ns istio-system
```

Congratulations!!! This concludes the Istio Lab