Table of Contents

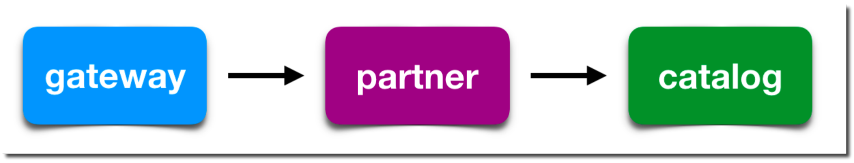
Dynamic Routing with Istio Lab

Goals

- Deploy version 2 of business services
- Configure service mesh route rules to dynamically route and shape traffic between services

1. Overview

There are three microservices in this lab, and they are chained together in the following sequence:

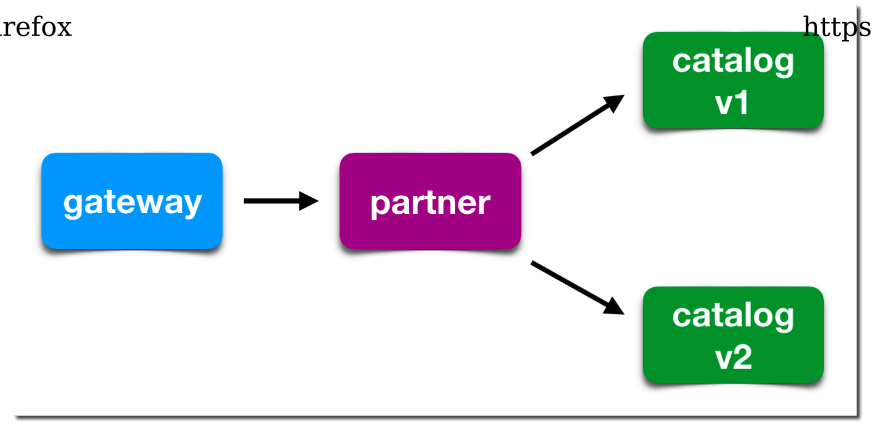


In this lab you dynamically alter routing between different versions of the catalog service.

You do so via routing mechanisms available from OpenShift. You then make use of routing mechanisms available from Red Hat Service Mesh.

2. Deploy Catalog Service Version 2

You can experiment with Istio routing rules by deploying a second version of the catalog service (`v2`).



You start by deploying the catalog service to RHOCP. The sidecar proxy is automatically injected.

1. Deploy the `catalog-v2` service:

   ```
   oc create \
         -f ~/lab/ocp-service-mesh-foundations/catalog-v2/kubernetes/catalog-service-
   template.yml \
         -n $OCP_TUTORIAL_PROJECT
   ```

   Sample Output

   ```
   deployment.extensions/catalog-v2 created
   ```

2. View both versions of the running `catalog` pods with the following commands:

   ```
   oc get pods -l application=catalog -n $OCP_TUTORIAL_PROJECT -w
   ```

3. For the `catalog-v2` service, wait until the Ready column displays `2/2` pods and the Status column displays `Running`:

   Sample Output

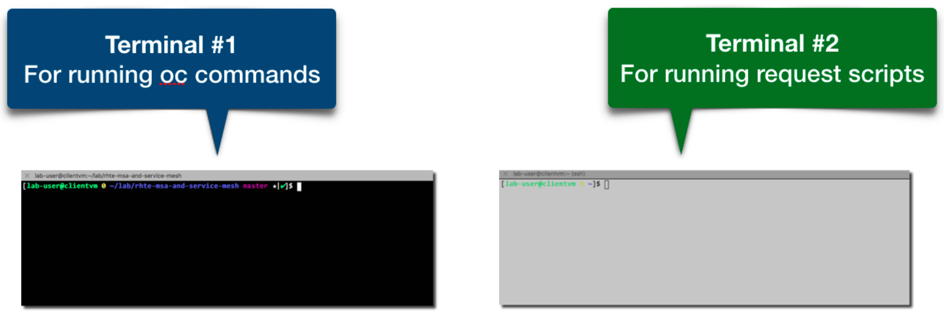
   ```
   NAME                            READY     STATUS     RESTARTS    AGE
   catalog-v1-6b576ffcf8-g6b48     2/2       Running    0           31m
   catalog-v2-7764964564-hj8xl     2/2       Running    0           49s
   ```

4. Press **Ctrl+C** to exit.

3. Set Up Multiple Terminal Windows

In order to help test your application, set up two terminal windows:

- Terminal 1: Running `oc` commands
- Terminal 2: Running request scripts



```
lab-user@clientvm:~/lab/rhte-msa-and-service-mesh
[lab-user@clientvm 0 ~/lab/rhte-msa-and-service-mesh master *|✓]$
```

```
lab-user@clientvm:~ (ssh)
[lab-user@clientvm ~]$
```

First, complete some prep work before setting up your terminals.

1. Open a new terminal window on your computer.
2. Use `ssh` to access your VM using the `ssh` command provided in your welcome email.
   - This new terminal is called "Terminal #2" for the remainder of this lab.

4. Test Loadbalancing using OCP Functionality

In this section of the lab, you test loadbalancing between v1 and v2 of the *catalog* service.

4.1. View OpenShift configurations

1. In Terminal #1, execute the following command to view the value of the *Selector* in the *catalog* service:

   ```
   oc describe service catalog -n $OCP_TUTORIAL_PROJECT | grep
   Selector

   ...

   Selector:          application=catalog
   ```

2. Execute the following command to view the values of the labels of the **v1 version** of the catalog:

```
oc get deploy catalog-v1 -o json -n $OCP_TUTORIAL_PROJECT | jq
.spec.template.metadata.labels

...

{
  "app": "catalog",
  "application": "catalog",
  "deployment": "catalog",
  "version": "v1"
}
```

3. Execute the following command to view the values of the labels of the **v2 version** of the catalog:

```
oc get deploy catalog-v2 -o json -n $OCP_TUTORIAL_PROJECT | jq
.spec.template.metadata.labels

...

{
  "app": "catalog",
  "application": "catalog",
  "deployment": "catalog",
  "version": "v2"
}
```

4. Notice from the previous results that the value of the *catalog* service *Selector* (application=catalog) matches one of the labels found in both v1 and v2 of the *catalog* deployments.

   The implication of this is that OpenShift will loadbalance incoming requests in a round-robin manner to v1 and v2 of the *catalog* pods.

4.2. Verify OpenShift round-robin load-balancing

1. Move to your new terminal, **Terminal #2**.
2. Send the following request to the `gateway` service:

```
curl $GATEWAY_URL
```

Sample Output

```
gateway => partner => catalog v1 from '6b576ffcf8-g6b48': 2
```

- Here, `6b576ffcf8-g6b48` is the pod running `v1`, and `2` is the number of times the endpoint was hit.

3. Make another request to the `gateway` service:

```
curl $GATEWAY_URL
```

Sample Output

```
gateway => partner => catalog v2 from '7764964564-hj8xl': 1
```

- Here, `7764964564-hj8xl` is the pod running `v2`, and `1` is the number of times the endpoint was hit.
- By default, OpenShift round-robin load balancing is applied when there is more than one pod behind a service.

4. Send the following requests to the `gateway` service:

```
~/lab/ocp-service-mesh-foundations/scripts/run-all.sh
```

Sample Output

```
gateway => partner => catalog v1 from '6b576ffcf8-g6b48': 4
gateway => partner => catalog v2 from '7764964564-hj8xl': 3
gateway => partner => catalog v1 from '6b576ffcf8-g6b48': 5
gateway => partner => catalog v2 from '7764964564-hj8xl': 4
gateway => partner => catalog v1 from '6b576ffcf8-g6b48': 6
gateway => partner => catalog v2 from '7764964564-hj8xl': 5
gateway => partner => catalog v1 from '6b576ffcf8-g6b48': 7
gateway => partner => catalog v2 from '7764964564-hj8xl': 6
gateway => partner => catalog v1 from '6b576ffcf8-g6b48': 8
gateway => partner => catalog v2 from '7764964564-hj8xl': 7
...
```

- Let this script continue to run.

5. Define Service Mesh Routing Rules

Service Mesh *Route rules* control how requests are routed within an Istio service mesh.

Requests can be routed based on the source and destination, HTTP header fields, and weights associated with individual service versions. For example, a route rule could route requests to different versions of a service.

In addition to the usual OpenShift® object types like `BuildConfig`, `DeploymentConfig`, `Service` and `Route`, you also have new object types installed as part of Istio, like `VirtualService`. Adding these objects to the running OpenShift cluster is how you configure routing rules for Istio.

`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/version of it) defined in the registry. The source of traffic can also be matched in a routing rule. This allows routing to be customized for specific client contexts.

`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.

5.1. Route All Traffic to v2

1. Review the following Istio configuration file to route traffic to `v2`:
   - File name: `istiofiles/virtual-service-catalog-v2.yml`

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

- This definition allows you to configure a percentage of traffic and direct it to a specific version of the `catalog` service. In this case, 100% of traffic *(weight)* for the `catalog` service always goes to pods matching the labels version `v2`.
- The selection of pods here is very similar to the Kubernetes selector model for matching based on labels. So, any service within the service mesh that tries to communicate with the `catalog` service is always routed to `v2` of the `catalog` service.

2. Move back to **Terminal #1**.
3. Route traffic to `v2` using the configuration file:

```
oc create -f ~/lab/ocp-service-mesh-foundations/istiofiles/destination-rule-catalog-
v1-v2.yml -n $OCP_TUTORIAL_PROJECT

oc create -f ~/lab/ocp-service-mesh-foundations/istiofiles/virtual-service-catalog-
v2.yml -n $OCP_TUTORIAL_PROJECT
```

4. In Termail #2, view the output and confirm that requests are being routed to `catalog:v2`.

NOTE It takes a while for traffic to stabilize.

5.2. Route All Traffic to v1

In this section, you switch requests over to `v1`. You use the following configuration file:

- File name: `istiofiles/virtual-service-catalog-v1.yml`

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

1. Review the file and make note of the weight set to 100 for catalog v1.

2. Next, move everyone to catalog service `v1`, by entering the following command in **Terminal #1**:

```
oc replace -f ~/lab/ocp-service-mesh-foundations/istiofiles/virtual-service-catalog-
v1.yml -n $OCP_TUTORIAL_PROJECT
```

NOTE You use `oc replace` instead of `oc create` because you are overlaying the previous rule.

This completes the lab! In this lab you learned how to deploy microservices to form a *service mesh* using Istio. You also learned how to do traffic shaping and routing using *Route Rules*, which instruct the Istio sidecar proxies to distribute traffic according to a specified policy.

Last updated 2020-02-26 17:17:48 EST