Deploy a Demo Web Service with Envoy Proxy

Install istio from this link first:

<https://istio.io/docs/setup/getting-started/>

Now we finally are at the fun part of the tutorial. Let’s check the routing capabilities of this service mesh. First, we will deploy two demo web services, “blue” and “green,” like in one of our previous tutorials. Copy the following into a yaml file named my-websites.yaml:

apiVersion: apps/v1beta1

kind: Deployment

metadata:

  name: web-v1

  namespace: default

spec:

  replicas: 1

  template:

    metadata:

      labels:

        app: website

        version: website-version-1

    spec:

      containers:

      - name: website-version-1

        image: kublr/kublr-tutorial-images:v1

        resources:

          requests:

            cpu: 0.1

            memory: 200

---

apiVersion: apps/v1beta1

kind: Deployment

metadata:

  name: web-v2

  namespace: default

spec:

  replicas: 1

  template:

    metadata:

      labels:

        app: website

        version: website-version-2

    spec:

      containers:

      - name: website-version-2

        image: kublr/kublr-tutorial-images:v2

        resources:

          requests:

            cpu: 0.1

            memory: 200

---

apiVersion: apps/v1beta1

kind: Deployment

metadata:

  name: web-v3

  namespace: default

spec:

  replicas: 1

  template:

    metadata:

      labels:

        app: website

        version: website-version-3

    spec:

      containers:

      - name: website-version-3

        image: kublr/kublr-tutorial-images:v3

        resources:

          requests:

            cpu: 0.1

            memory: 200

---

apiVersion: v1

kind: Service

metadata:

  name: website

spec:

  ports:

  - port: 80

    targetPort: 80

    protocol: TCP

    name: http

  selector:

    app: website

Note that when you want to use the Envoy sidecar with your pods, the label “app” should be present (it’s used in the request tracing feature), and “spec.ports.name” in service definition must be named properly (http, http2, grpc, redis, mongo) otherwise Envoy will act on that service traffic as if it was plain TCP, and you will not be able to use the layer 7 features with those services! Also, the pods must be targeted only by 1 “service” in the cluster. As you can see above, the definition file has three simple deployments each using a different version of the web service (v1/v2/v3), and three simple services, each pointing at the corresponding deployment.

Now we will add the needed Envoy proxy configuration to the pod definitions in this file, using “[istioctl kube-inject](https://istio.io/docs/reference/commands/istioctl.html" \l "istioctl%20kube-inject)” command. It will produce a new yaml file with additional components of the Envoy sidecar ready to be deployed by kubectl, run: istioctl kube-inject -f my-websites.yaml -o my-websites-with-proxy.yaml

The output file will contain extra configuration, you can inspect the “my-websites-with-proxy.yaml” file. This command took the pre-defined ConfigMap “istio-sidecar-injector” (that was installed earlier when we did istio installation), and added the needed sidecar configurations and arguments to our deployment definitions. When we deploy the new file “my-websites-with-proxy.yaml”, each pod will have two containers, one of our demo application and one Envoy proxy. Run the creation command on that new file: kubectl apply -f my-websites-with-proxy.yaml

You will see this output if it worked as expected:

deployment "web-v1" created

deployment "web-v2" created

deployment "web-v3" created

service "website" created

Let’s inspect the pods to see that the Envoy sidecar is present:  kubectl get pods

Let’s inspect the pods to see that the Envoy sidecar is present:  kubectl get pods.

We can see that each pod has two containers, one is the website container and another is the proxy sidecar:

How to Implement a Service Mesh with Istio

Also, we can inspect the logs of the Envoy proxy by running: kubectl logs <your pod name> istio-proxy

You will see a lot of output, with last lines similar to this:

add/update cluster outbound|80|version-1|website.default.svc.cluster.local starting warming

add/update cluster outbound|80|version-2|website.default.svc.cluster.local starting warming

add/update cluster outbound|80|version-3|website.default.svc.cluster.local starting warming

warming cluster outbound|80|version-3|website.default.svc.cluster.local complete

warming cluster outbound|80|version-2|website.default.svc.cluster.local complete

warming cluster outbound|80|version-1|website.default.svc.cluster.local complete

This means that the proxy sidecar is healthy and running in that pod.

Now we need to deploy the minimal Istio configuration resources, needed to route the traffic to our service and pods, save the following manifests into a file named “website-routing.yaml”:

---

apiVersion: networking.istio.io/v1alpha3

kind: Gateway

metadata:

  name: website-gateway

spec:

  selector:

    # Which pods we want to expose as Istio router

    # This label points to the default one installed from file istio-demo.yaml

    istio: ingressgateway

  servers:

  - port:

      number: 80

      name: http

      protocol: HTTP

    # Here we specify which Kubernetes service names

    # we want to serve through this Gateway

    hosts:

    - "\*"

---

apiVersion: networking.istio.io/v1alpha3

kind: VirtualService

metadata:

  name: website-virtual-service

spec:

  hosts:

  - "\*"

  gateways:

  - website-gateway

  http:

  - route:

    - destination:

        host: website

        subset: version-1

---

apiVersion: networking.istio.io/v1alpha3

kind: DestinationRule

metadata:

  name: website

spec:

  host: website

  subsets:

  - name: version-1

    labels:

      version: website-version-1

  - name: version-2

    labels:

      version: website-version-2

  - name: version-3

    labels:

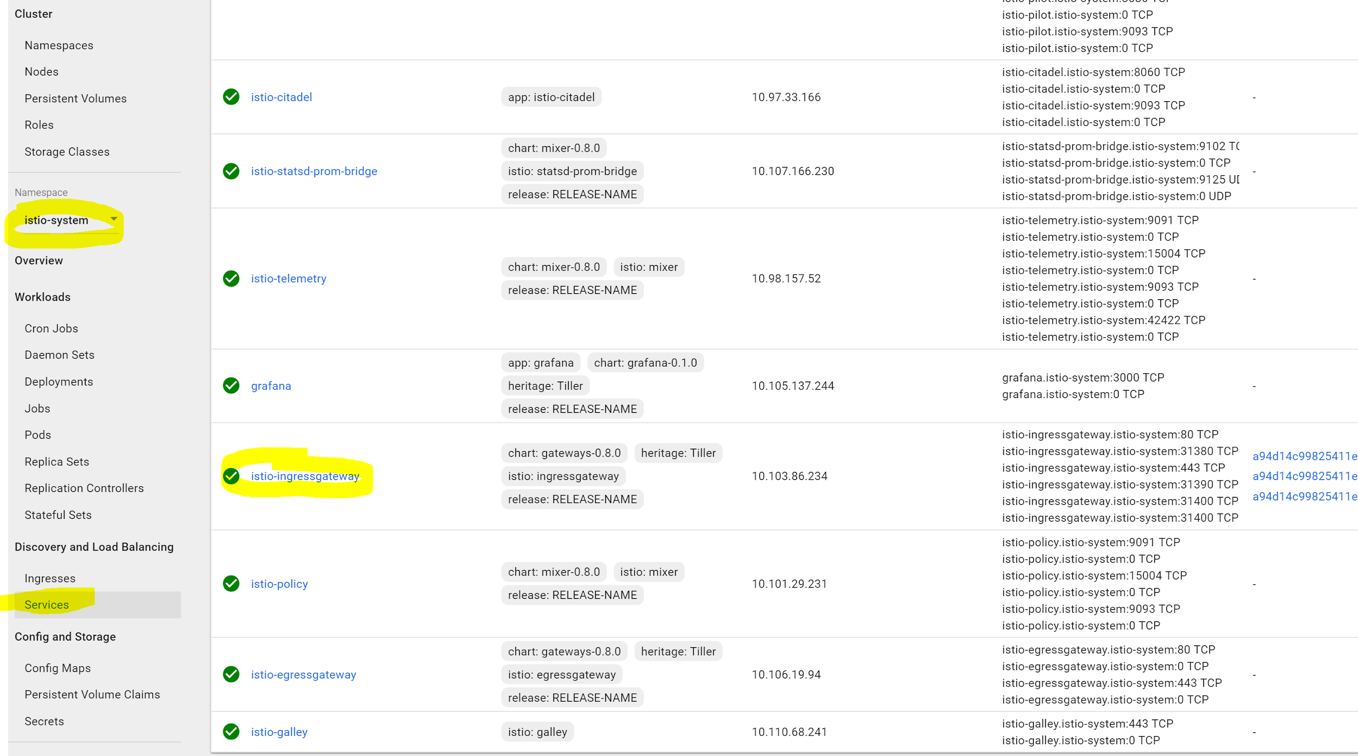
      version: website-version-3

These are Gateway, VirtualService, and DestinationRule. Those are custom Istio resources that manage and configure the ingress behavior of istio-ingressgateway pod. We will describe them more in-depth in the next tutorial which gets to the technical details of Istio configuration. For now, deploy these resources to be able to access our example website: kubectl create -f website-routing.yaml

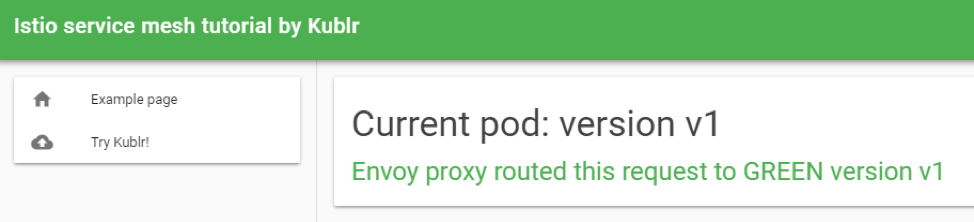
Next step is to visit our demo website. We deployed three “versions”, each shows different page text and color, but at the moment we can reach only version 1 through the Istio ingress. Let’s visit our endpoint just to be sure there is a web service deployed.

Find your external endpoint by running: kubectl get services istio-ingressgateway -n istio-system

Or find it by browsing to the istio-ingressgateway service as shown below (we also saw it at the beginning of the tutorial):



Visit the external endpoint by clicking it. You may see several links because one link points to HTTPS and another to HTTP port of the load balancer. If so, use only HTTP link, because we did not setup TLS for this tutorial, you should see the v1 page of the demo website:



The exact configuration which makes our “website” Kubernetes service point only to single deployment is the Istio VirtualService we created for the website. It tells the Envoy proxy to route requests of “website” service only to pods with label “version: website-version-1” (you probably noticed that the manifest of service “website” selects only one label “app: website” from our pods but says nothing about the “version” label to pick from – so without Envoy logic the Kubernetes service itself would do round robin between all pods with “app: website” label, both version one, two and three). You can change the version of the website that we see by changing the following section of the VirtualService manifest and redeploying it:

  http:

  - route:

    - destination:

        host: website

        subset: version-1

The “subset” is where we chose the correct section of DestinationRule to route to, and we will learn in depth about these resources in the next tutorial.

Usually when new version of an application needs to be tested with a small amount of traffic (canary deployment), the vanilla Kubernetes approach would be to create a second deployment that uses a new Docker image but the same pod label, causing the “service” that sends traffic to this pod label, while also balancing between the newly plugged pods from the second deployment. It is not as flexible as an Istio solution. You cannot easily point 10% of traffic to the new deployment (in order to reach a precise 10% you will need to keep the pod replicas ratio between two deployments according to the needed percentage, like 9 “v1 pods” and 1 “v2 pod”, or 18 “v1 pods” and 2 “v2 pods”), and cannot use HTTP header for example to route requests to particular version.

Istio solves this limitation through its flexible VirtualService configuration. For instance, if you want to route traffic using the 90/10 rule, it can easily do it like this:

apiVersion: networking.istio.io/v1alpha3

kind: VirtualService

metadata:

  name: website-virtual-service

spec:

  hosts:

  - "\*"

  gateways:

  - website-gateway

  http:

  - route:

    - destination:

        host: website

        subset: version-1

      weight: 90

    - destination:

        host: website

        subset: version-2

      weight: 10

The source code for the article is available on github: https://github.com/kublr/istio-blog-sample.