Installing Istio, Prometheus, and Flagger

This lesson gives step-by-step instructions for installing Istio, Prometheus, and Flagger.

We'll cover the following

We'll install all the tools we need as Jenkins X add-ons. They are an excellent way to install and integrate tools. However, add-ons might not provide you with all the options you can use to tweak those tools to your specific needs. Later on, once you adopt Jenkins X in production, you should evaluate whether you want to continue using the add-ons or you prefer to set up those tools in some other way. The latter might give you more freedom. For now, add-ons are the easiest way to set up what we need, so we'll roll with them.

Installing Istio and Prometheus

Before we install **Istio** in our cluster, we'll need to install `istioctl` (its command-line tool). Instead of showing you the instructions on how to do that, please go to the Istio Releases, download the package for your operating system, unpack it, move it to the folder that is in your `PATH`, and make sure that it is executable. If you already have `istioctl`, make sure that it is version 1.5+. If it's not, please upgrade it.

Now we're ready to install Istio in our cluster.

```
istioctl manifest apply \
    --set profile=demo
```

Istio is resource-heavy. It is the reason why we increased the size of the VMs that compose our cluster.

When installing **Istio**, a new **Ingress** gateway service is created. It is used for sending all the incoming traffic to services based on **Istio** rules (`VirtualServices`). That achieves a similar functionality as the one provided by **Ingress**. While **Ingress** has the advantage of being simple, it is also very limited in its capabilities. **Istio**, on the other hand, allows us to create advanced rules for incoming traffic, and that's what we'll need for canary deployments.

For now, we need to find the external IP address of the **Istio Ingress** gateway service. Just as with **Ingress**, the way to retrieve the IP differs depending on whether you're using EKS or some other Kubernetes flavor.

Please run the command that follows only if you are **NOT** using **EKS** (e.g., GKE, AKS, etc.).

```
ISTIO_IP=$(kubectl \
    --namespace istio-system \
    get service istio-ingressgateway \
    --output jsonpath='{.status.loadBalancer.ingress[0].ip}')
```

Please run the commands that follow only if you are using **EKS**.

```
ISTIO_HOST=$(kubectl \
    --namespace istio-system \
    get service istio-ingressgateway \
    --output jsonpath='{.status.loadBalancer.ingress[0].hostname}')

export ISTIO_IP="$(dig +short $ISTIO_HOST \
    | tail -n 1)"
```

To be on the safe side, we'll output the environment variable to confirm that the IP does indeed looks to be correct.

```
echo $ISTIO_IP
```

When we installed **Istio**, **Prometheus** was installed alongside it.

Installing Flagger

The only tool left for us to add is **Flagger**. Later on, we'll see why I skipped **Grafana**.

```
kubectl apply \
    --kustomize github.com/weaveworks/flagger/kustomize/istio
```

Now, let's take a quick look at the Pods that were created through those two add-ons.

```
kubectl --namespace istio-system \
    get pods
```

The output is as follows.

```
NAME                         READY STATUS   RESTARTS AGE
flagger-...                  0/1   Running 0        10s
grafana-...                  1/1   Running 0        98s
istio-egressgateway-...      1/1   Running 0        110s
istio-ingressgateway-...     1/1   Running 0        108s
istio-tracing-...            1/1   Running 0        98s
istiod-...                   1/1   Running 0        2m12s
kiali-...                    1/1   Running 0        97s
prometheus-...               2/2   Running 0        97s
```

We won't go into details of what each of those Pods does. I expect you to consult the documentation if you are curious. For now, we'll note that **Flagger**, **Istio**, and **Prometheus** Pods were created in the `istio-system` Namespace and that, by the look of it, they are all running. If any of those are in the pending state, you either need to increase the number of nodes in your cluster or none of the nodes is big enough to meet the demand of the requested resources. The former case should be solved with the Cluster Autoscaler if you have it running in your Kubernetes cluster. The latter, on the other hand, probably means that you did not follow the instructions to create a cluster with bigger VMs. In any case, the next step would be to describe the pending Pod, see the root cause of the issue, and act accordingly.

Telling Istio to auto-inject sidecar proxy containers

There's still one thing missing. We need to tell Istio to auto-inject sidecar proxy containers to all the Pods in the `jx-production` Namespace.

```
kubectl label namespace jx-production \
    istio-injection=enabled \
    --overwrite
```

We got a new label, `istio-injection=enabled`. That one tells **Istio** to inject the sidecar containers into our Pods. Without it, we'd need to perform additional manual steps, and you already know that's not something I like doing.

Whichever deployment strategy we use, it should be the same in all the permanent environments. Otherwise, we do not have parity between applications running in production and those running in environments meant to be used for testing (e.g., staging). The more similar, if not the same, those environments are, the more confident we are to promote something to production. Otherwise, we can end up in a situation where someone could rightfully say that what was tested is not the same as what is being deployed to production.

Checking labels for the staging environment

So, let's take a look at the labels in the staging environment.

```
kubectl describe namespace \
    jx-staging
```

The output, limited to the relevant parts, is as follows.

```
Name:         jx-staging
Labels:       env=staging
              team=jx
...
```

As we can see, the staging environment does not have the `istio-injection=enabled` label, so **Istio** will not inject sidecars and, as a result, it will not work there. Given that we already elaborated that staging and production should be as similar as possible, if not the same, we'll add the missing label so that **Istio** works in both.

Adding the missing label to the staging environment

```
kubectl label namespace jx-staging \
    istio-injection=enabled \
    --overwrite
```

Let's have another look at the staging environment to confirm that the label was indeed added correctly.

```
kubectl describe namespace \
    jx-staging
```

The output, limited to the relevant parts, is as follows.

```
Name:         jx-staging
Labels:       env=staging
              istio-injection=enabled
              team=jx
...
```

The `istio-injection=enabled` is there, and we can continue while knowing that whatever **Istio** will do in the staging environment will be the same as in production.

In the next lesson, we will learn to create canary resources with **Flagger**.