Creating Canary Resources with Flagger

In this lesson, we will learn how to create canary resources with Flagger.

We'll cover the following

- The Canary resource definition
  - canary.enabled
  - spec.provider
  - spec.targetRef
  - spec.progressDeadlineSeconds
  - spec.service
  - spec.analysis
  - Istio Gateway
- The Helm values
  - host
  - canaryAnalysis
- Adding staging-specific values
  - Cloning the repository
  - Enabling the Canary deployments and defining the address
  - Pushing the changes to the repository
  - Checking the activities to confirm the changes

Let’s say we want to deploy our new release only to 20% of the users and that we will monitor metrics for 60 seconds. During that period, we’ll be validating whether the error rate is within a predefined threshold and whether the time it takes to process requests is within some limits. If everything seems right, we’ll increase the percentage of users who can use our new release for another 20%, and continue monitoring metrics to decide whether to proceed. The process should repeat until the new release is rolled out to everyone, twenty percent at a time every thirty seconds.

Now that we have a general idea of what we want to accomplish and that all the tools are set up, all that's missing is to create **Flagger** `Canary` definition.

Fortunately, canary deployments with **Flagger** are available in Jenkins X build packs since January 2020. So, there's not much work to do to convert our application to use the canary deployment process.

The `Canary` resource definition

Let's take a look at the `Canary` resource definition already available in our project.

```
cat charts/jx-progressive/templates/canary.yaml
```

The output is as follows.

```
{{- if .Values.canary.enabled }}
apiVersion: flagger.app/v1alpha3
kind: Canary
metadata:
  name: {{ template "fullname" . }}
  labels:
    draft: {{ default "draft-app" .Values.draft }}
    chart: "{{ .Chart.Name }}-{{ .Chart.Version | replace "+" "_" }}"
spec:
  provider: istio
  targetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: {{ template "fullname" . }}
  progressDeadlineSeconds: {{ .Values.canary.progressDeadlineSeconds }}
  {{- if .Values.hpa.enabled }}
  autoscalerRef:
    apiVersion: autoscaling/v2beta1
    kind: HorizontalPodAutoscaler
    name: {{ template "fullname" . }}
  {{- end }}
  service:
    port: {{ .Values.service.externalPort }}
    targetPort: {{ .Values.service.internalPort }}
    gateways:
    - {{ template "fullname" . }}
    hosts:
    - {{ .Values.canary.host }}
  analysis:
    interval: {{ .Values.canary.canaryAnalysis.interval }}
    threshold: {{ .Values.canary.canaryAnalysis.threshold }}
    maxWeight: {{ .Values.canary.canaryAnalysis.maxWeight }}
    stepWeight: {{ .Values.canary.canaryAnalysis.stepWeight }}
    metrics:
    - name: request-success-rate
      threshold: {{ .Values.canary.canaryAnalysis.metrics.requestSuccessRate.threshold }}
      interval: {{ .Values.canary.canaryAnalysis.metrics.requestSuccessRate.interval }}
    - name: request-duration
      threshold: {{ .Values.canary.canaryAnalysis.metrics.requestDuration.threshold }}
      interval: {{ .Values.canary.canaryAnalysis.metrics.requestDuration.interval }}
```

```
---

apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: {{ template "fullname" . }}
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: {{ .Values.service.externalPort }}
      name: http
      protocol: HTTP
    hosts:
    - {{ .Values.canary.host }}
{{- end }}
```

`canary.enabled`

The whole definition is inside an `if` statement so that `Canary` is created only if the value `canary.enabled` is set to `true`. That way, by default, we will not use canary deployments at all. Instead, we'll have to specify when and under which circumstances they will be created. That might spark the question *"why do we go into the trouble of making sure that canary deployments are enabled only in certain cases?"* **Why not use them always?**

By their nature, it takes much more time to execute a canary deployment than most of the other strategies. The canaries roll out progressively and are pausing periodically to allow us to validate the result on a subset of users. That increased duration can be anything from seconds to hours or even days or weeks. In some cases, we might have an important release that we want to test with our users progressively over a whole week. Such prolonged periods might be well worth the wait in production and staging, which should use the same processes. But, waiting for more than necessary to deploy a release in a preview environment is a waste. Those environments are not supposed to provide the "final stamp" before promoting to production, but rather to flash out most of the errors. After all, not using canaries in preview environments might not be the only difference. As we saw in the previous chapter, we might choose to make them serverless while keeping permanent environments using different deployment strategies. Long story short, the `if` statement allows us to decide when we'll do canary deployments. We are probably not going to employ it in all environments.

The `apiVersion`, `kind`, and `metadata` should be self-explanatory if you have minimal experience with Kubernetes and **Helm** templating minimal experience with Kubernetes and **Helm** templating.

`spec.provider`

The exciting entry is `spec.provider`. As the name suggests, it allows us to specify which provider we'll use. In our case, it'll be **Istio**, but I wanted to make sure that you can easily switch to something else like, for example, **Gloo**. If you are using GKE, you already have **Gloo** running. While exploring different solutions is welcome while learning, later on, you should probably use one or the other, not necessarily both.

`spec.targetRef`

The `spec.targetRef` tells `Canary` which Kubernetes object it should manipulate. Unlike serverless Deployments with Knative, which replace Kubernetes Deployments, `Canary` runs on top of whichever Kubernetes resource we're using to run our software. For *jx-progressive*, that's `Deployment`, but it could just as well be a `StatefulSet` or something else.

`spec.progressDeadlineSeconds`

The next in line is `spec.progressDeadlineSeconds`. Think of it as a safety net. If the system cannot progress with the deployment for the specified period (expressed in seconds), it will initiate a rollback to the previous release.

`spec.service`

The `spec.service` entry provides the information on how to access the application, both internally (`port`) and externally (`gateways`), as well as the `hosts` through which the end-users can communicate with the app.

`spec.analysis`

The `spec.analysis` entries are probably the most interesting ones. They define the analysis that should be done to decide whether to progress with the deployment or to roll back. Earlier I mentioned that the interval between progress iterations is thirty seconds. That's specified in the `interval` entry. The `threshold` defines how many failed metric checks are allowed before rolling back. The `maxWeight` sets the percentage of requests routed to the canary deployment before it gets converted to the primary. After that percentage is reached, all users will see the new release. More often than not, we do not need to wait until the process reaches 100% through smaller increments. We can say that, for example, when 50% of users are using the new release, there is no need to proceed with validations of the metrics.

The system can move forward and make the new release available to everyone

right away. The `stepWeight` entry defines how the big roll out increments should be (e.g., 20% at a time). Finally, `metrics` can host an array of entries, one for each metric and threshold that should be evaluated continuously during the process.

Istio `Gateway`

The second definition is a "standard" **Istio** `Gateway`. We won't go into it in detail since that would derail us from our mission by leading us into a vast subject of **Istio**. For now, think of the `Gateway` as being equivalent to **nginx Ingress** we've been using so far. It allows **Istio**-managed applications to be accessible from outside the cluster.

As you noticed, many of the values are not hard-coded into the `Canary` and `Gateway` definitions. Instead, we defined them as **Helm** values. That way, we can change all those that should be tweaked from `charts/jx-progressive/values.yaml`. So, let's take a look at them.

The Helm values

```
cat charts/jx-progressive/values.yaml
```

The relevant parts of the output are as follows.

```
...
# Canary deployments
# If enabled, Istio and **Flagger** need to be installed in the cluster
canary:
  enabled: false
  progressDeadlineSeconds: 60
  analysis:
    interval: "1m"
    threshold: 5
    maxWeight: 60
    stepWeight: 20
    # WARNING: Canary deployments will fail and rollback if there is no traffic that will generate
    metrics:
      requestSuccessRate:
        threshold: 99
        interval: "1m"
      requestDuration:
        threshold: 1000
        interval: "1m"
  # The host is using Istio Gateway and is currently not auto-generated
  # Please overwrite the `canary.host` in `values.yaml` in each environment repository (e.g., stag
  host: acme.com
...
```

`host`

We can set the address through which the application should be accessible through the `host` entry at the bottom of the `canary` section. The feature of creating **Istio Gateway** addresses automatically, like Jenkins X is doing with **Ingress**, is not available. So, we'll need to define the address of our application for each of the environments. We'll do that later.

`canaryAnalysis`

The `analysis` sets the interval to `1m`. So, it will progress with the rollout every minute. Similarly, it will roll back if it encounters failures (e.g., reaches metrics thresholds) `5` times. It will finish rollout when it reaches `60` percent of users (`maxWeight`), and it will increase the number of requests forwarded to the new release with increments of `20` percent (`stepWeight`).

Finally, it will use two metrics to validate rollouts and decide whether to proceed, to halt, or to roll back. The first metric is `requestSuccessRate` (`request-success-rate`) calculated throughout `1m`. If less than `99` percent of requests are successful (are not 5xx responses), it will be considered an error. Remember, that does not necessarily mean that it will rollback right away since the `analysis.threshold` is set to `5`. There must be five failures for the rollback to initiate. The second metric is `requestDuration` (`request-duration`). It is also measured throughout `1m` with the threshold of a second (`1000` milliseconds). It does not take into account every request but rather the 99th percentile.

We added the `istio-injection=enabled` label to the staging and the production environment namespaces. As a result, everything running in those Namespaces will automatically use **Istio** for networking.

Adding staging-specific values

Assuming that the default values suit our needs, there are only two changes we need to make to the values. We need to define the `host` for staging (and later for production), and we'll have to enable canary deployment. Remember, it is disabled by default.

The best place to add staging-specific values is the staging repository. We'll have to clone it, so let's get out of the local copy of the *jx-progressive* repo.

```
cd ..
```

Cloning the repository

Now we can clone the staging repository.

> ⚠ Please replace `[...]` with your GitHub user before executing the commands that follow.

```
rm -rf environment-jx-rocks-staging

GH_USER=[...]

git clone \
    https://github.com/$GH_USER/environment-jx-rocks-staging.git

cd environment-jx-rocks-staging
```

We removed the local copy of the environment repo just in case it was a leftover from the exercises in the previous chapters. After that, we cloned the repository and entered inside the local copy.

Enabling the `Canary` deployments and defining the address

Next, we need to enable `Canary` deployments for the staging environment and to define the address through which *jx-progressive* would be accessible.

```
STAGING_ADDR=staging.jx-progressive.$ISTIO_IP.nip.io

echo "jx-progressive:
  canary:
    enabled: true
    host: $STAGING_ADDR" \
    | tee -a env/values.yaml
```

All we did was to add a few variables associated with `jx-progressive`.

Pushing the changes to the repository

Now we can push the changes to the staging repository.

```
git add .

git commit \
    -m "Added progressive deployment"
```

```
git push
```

With the staging-specific values defined, we can go back to the *jx-progressive* repository and push the changes we previously did over there.

```
cd ../jx-progressive

git add .

git commit \
    -m "Added progressive deployment"

git push
```

Checking the activities to confirm the changes

We should not see a tangible change to the deployment process with the first release. So, all there is to do, for now, is to confirm that the activities initiated by pushing the changes to the repository were executed successfully.

```
jx get activities \
    --filter jx-progressive/master \
    --watch
```

Press *ctrl+c* to cancel the watcher once you confirm that the newly started build is finished.

```
jx get activities \
    --filter environment-jx-rocks-staging/master \
    --watch
```

Press *ctrl+c* to cancel the watcher once you confirm that the newly started build is finished.

Now we're ready with the preparations for `Canary` deployments. In the next lesson, we can finally take a closer look at the process itself as well as the benefits it brings.