Limiting Serverless Deployments to Pull Requests

This lesson discusses if we can limit serverless deployments and if so, how to do that?

We'll cover the following

Can we limit serverless deployment?

I already explained that running applications as serverless deployments in preview environments is highly beneficial. As a result, you might have the impression that an application must be serverless in all environments. That is certainly not the case. We can, for example, choose to run some applications as serverless deployments in the preview environment and run them as normal apps in those that are permanent.

To be fair, we could have more complicated arrangements wit running a serverless application in the staging environment but non-serverless in production. However, that would mean that we are not testing what we’re deploying to production. After all, serverless applications are not behaving in the same way as other types of deployments.

When should the final testing be performed?

When should the final testing be performed?

Now, you could argue that preview environments are used for testing, so they should be the same as production. While it is true that testing is the primary function of preview environments, they are not used for the final round of testing. By their nature, preview environments are more lightweight and do not contain the whole system, only the parts required to validate pull requests. The real or final test is performed in the staging environment if we are performing continuous delivery, or in production, if we are practicing continuous deployment. The latter option would require some form of progressive delivery, which we might explore later. For now, I'll assume that you are following the continuous delivery model and, therefore, the staging environment is the one that should be production-like.

All in all, we'll explore how to make an application serverless only in preview environments, and continue being whatever it was before in permanent ones.

Since our *jx-knative* application is already serverless by default, we'll go with the least possible effort and leave it as is, but disable `knativeDeploy` in values for the staging in production environments.

Cloning the staging environment repository

Our first order of business is to clone the staging environment repository.

```
cd ..

rm -rf environment-jx-rocks-staging

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

cd environment-jx-rocks-staging
```

We removed a local copy of the staging repository just in case there are left-overs from one of the previous chapters, we cloned the repo, and we entered inside the local copy.

Modifying the value of the `knativeDeploy` variable

Now, changing the way an application is deployed to a specific repository is as easy as changing the value of the `knativeDeploy` variable. But, since an environment defines all the applications running in it, we need to specify for which one we're changing the value. To be more precise, since all the apps in an environment are

defined as dependencies in `requirements.yaml`, we need to prefix the value with the alias of the dependency. In our case, we have `jx-knative` and potentially a few other applications. We want to ensure that `jx-knative` is not running as serverless in the staging environment.

```
echo "jx-knative:
  knativeDeploy: false" \
    | tee -a env/values.yaml
```

Pushing the changes to GitHub

Now that we added the `jx-knaative.knativeDeploy` variable set to `false`, we can push the changes and let Jenkins X do the job for us.

```
git add .

git commit -m "Removed Knative"

git pull

git push
```

Making a (trivial) change to the application

Even though that push will trigger a new deployment, it will not recreate the required **Ingress** resource, so we'll need to make a trivial change to the application as well. That should result in the new deployment with everything we need for our *jx-knative* application to behave in the staging environment as if it is a normal application (not serverless).

```
cd ../jx-knative

git checkout master

echo "jx-knative rocks" \
    | tee README.md

git add .

git commit -m "Removed Knative"

git pull

git push
```

Checking activities of `jx-knative`

Just as before, we'll check the activities of the project pipeline to confirm that it

executed successfully.

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

Feel free to stop watching the activities with *ctrl+c*, and double-check that the activity triggered by making changes to the staging environment repository is finished as well.

Checking activities of the application in staging

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

You know what to do now. Press *ctrl+c* when the newly spun activity is finished.

Confirming the non-serverless deployment

Now, let's check whether the application was indeed deployed to staging as non-serverless.

```
kubectl \
    --namespace jx-staging \
    get pods
```

The output is as follows:

```
NAME              READY STATUS  RESTARTS AGE
jx-jx-knative-... 1/1   Running 0        78s
```

Checking all resources related to `jx-knative`

It's hard to see whether an application is serverless or not by looking at the Pods, so we'll check all the resources related to the `jx-knative` app.

```
kubectl \
    --namespace jx-staging \
    get all \
    | grep jx-knative
```

The output is as follows:

```
pod/jx-jx-knative-66df89fb74-mgg68   1/1     Running   0          3m29s
service/jx-knative   ClusterIP   10.31.254.208   <none>        80/TCP    3m29s
deployment.apps/jx-jx-knative   1/1     1            1           8m18s

replicaset.apps/jx-jx-knative-5cbd9d4799   0         0         0       8m18s
replicaset.apps/jx-jx-knative-66df89fb74   1         1         1       3m30s
release.jenkins.io/jx-knative-0.0.4   jx-knative   v0.0.4    https://github.com/vfarcic/jx-knative
```

As you can see, we have a bunch of resources. What matters, in this case, is not what we do have, but rather what we don't. There is no `ksvc`. Instead, we got the "standard" resources like **Service**, **Deployment**, **ReplicaSets**, etc.

Sending a request and observing the output

Since you know that I have a paranoid nature, you won't be surprised that we'll double-check whether the application works by sending a request and observing the output.

```
ADDR=$(kubectl \
    --namespace jx-staging \
    get ing jx-knative \
    --output jsonpath="{.spec.rules[0].host}")

echo $ADDR

curl "http://$ADDR"
```

If you got the familiar message, the application works and is back to its non-serverless form.

Right now, our preview and production environments feature serverless deployments of *jx-knative*, while staging is back to normal deployment. We should make a similar change to the production repository, but I will not provide instructions for that since they are the same as the staging environment. Think of that as a challenge that you should complete alone.

Let's wrap up the discussion of serverless deployment in the next lesson and free up the used resources.