Using Serverless Deployments with Pull Requests

This lesson explains the benefits of using serverless deployments with pull requests and the steps to achieve it.

We'll cover the following

So far, all the changes we made were pushed to a branch. That was on purpose since one of my goals was to show you the benefits of using serverless deployments with pull requests.

Benefits of using serverless deployments with pull requests

The percentage of the apps running as serverless inevitably varies. Some might have all the stateless applications running as serverless, while others might have none. Between those two extremes can be all shades of gray. While I cannot tell you how many apps you should run as serverless deployments, what I can tell you with a high level of certainty is that you'll use **Knative** much more in temporary environments like those created for pull requests than in permanent ones like staging and production. The reason for such a bold statement lies in the differences in purpose for running applications.

differences in purpose for running applications.

Resource usage in preview environments

An application running in a preview environment is to validate (automatically and/or manually) that a change we'd like to merge to the master branch is a good one and that it is relatively safe to merge it with production code. However, the validations we're running against a release in a preview environment are often not the final ones. A preview environment is often not the same as production.

- It might differ in size.
- It might not be integrated with all the other applications.
- It might not have all the data we need, and so on and so forth.

That's why we have the staging and other non-production permanent environments. Their primary purpose is often to provide a production-like environment where we can run the **final set of validations before promoting to production**. If we'd do that for each preview environment, *our resource usage would go through the roof*.

Imagine having hundreds or even thousands of open pull requests and that each is deployed to a preview environment. Now, imagine that each pull request is a full-blown system. *How much **CPU** and **memory** would that require?*

For that reason, our preview environments used with pull requests usually contain only the application in question. We might add one or two essential applications it integrates with, but we seldom deploy the full system there. Instead, we use mocks and stubs to test applications in those environments. That, by itself, should save a lot of resources, while still maintaining the advantages of preview environments. Nevertheless, we can do better than that.

Preview environments are one of the **best use-cases for serverless loads**. We might choose to run our application as serverless in production, or we might decide not to do so. The decision will depend on many factors, user experience being one of the most important ones. If our application scales to zero replicas due to lack of traffic when a request does come, it might take a while until the process in the newly spun replica is fully initialized. It should be obvious why forcing our users to wait for more than a second or two before receiving the first response is a bad idea. They are impatient and are likely to go somewhere else if they are unhappy with us. Also, our production traffic is likely going to be more demanding and less volatile than the one exercised over deployments in preview environments.

When a preview environment is created as a result of creating a pull request, our application is deployed to a unique Namespace. Typically, we would run some automated tests right after the deployment. *But what happens after that?* The answer is **“mostly nothing”**. A while later (a *minute*, an *hour*, a *day*, or even a *week*), someone might open the deployment associated with the pull request and do some manual validations. Those might result in the need for new issues or validations. As a result, that pull request is likely going to be idle for a while, before it is used again. So, we have mostly unused deployments wasting our memory and CPU.

Deployments initiated by pull requests

Given that we established how the preview environments represent a **colossal waste in resources**, we can easily conclude that deployments initiated by pull requests are one of the best candidates for serverless computing.

But, that would also assume that we do not mind waiting until an application is scaled from zero to one or more replicas. In my experience, that is rarely a problem. We cannot put users of our production releases and pull request reviewers on the same level. It should not be a problem if a person who decides to review a pull request and validate the release candidate manually has to wait for a second or even a minute until the application is scaled up. That loss in time is more than justified with better usage of resources.

Before and after the review, our app would use no resources unless it has dependencies (e.g., *database*) that cannot be converted into serverless deployments. We can gain a lot, even in those cases, when only some of the deployments are serverless. A hundred percent of deployments in preview environments running as serverless is better than fifty percent. *Still, fifty percent is better than nothing*.

Databases in preview environments

Databases in preview or even in permanent environments can be serverless as well. As long as their state is safely stored in a network drive, that should be able to continue operating when scaled from zero to one or more replicas. Nevertheless, databases tend to be slow to boot, especially when having a lot

> of data. Even though they could be serverless, they are probably not the right candidate. The fact that we can do something does not mean that we should.

Creating a pull request

Now, let's create a pull request and see how the `jx-knative` behaves.

```
git checkout -b serverless

echo "A silly change" | tee README.md

git add .

git commit -m "Made a silly change"

git push --set-upstream origin serverless

jx create pullrequest \
    --title "A silly change" \
    --body "What I can say?" \
    --batch-mode
```

We created a branch, we made "a silly change", and we created a new pull request.

Storing the name of the branch

Next, since Jenkins X treats pull requests as yet-another-branch, we'll store the name of that branch in an environment variable.

> ⚠ Please replace `[...]` with `PR-[PR_ID]` (e.g., PR-109). You can extract the ID from the last segment of the pull request address.

```
BRANCH=[...] # e.g., `PR-1`
```

Checking activities

Now, let's double check that the pull request was processed successfully.

```
jx get activities \
    --filter jx-knative/$BRANCH \
    --watch
```

Feel free to press *ctrl+c* to stop watching the activities when all the steps in the pipeline run were executed correctly.

Finding the name of the Namespace

To see what was deployed to the preview environment, we need to know the Namespace Jenkins X created for us. Fortunately, it uses a predictable naming convention, so we can reconstruct the Namespace easily using the base Namespace, GitHub user, application name, and the branch.

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

```
GH_USER=[...]

PR_NAMESPACE=$(\
  echo jx-$GH_USER-jx-knative-$BRANCH \
  | tr '[:upper:]' '[:lower:]')

echo $PR_NAMESPACE
```

In my case, the output of the last command was `jx-vfarcic-jx-knative-pr-1`. Yours should be different but follow the same logic.

Checking Pods in the preview Namespace

Now we can take a look at the Pods running in the preview Namespace.

```
kubectl --namespace $PR_NAMESPACE \
    get pods
```

The output is as follows:

```
NAME                             READY STATUS   RESTARTS AGE
jx-knative-ccqll-deployment-... 2/2   Running 0        109s
```

We can see two interesting details just by observing the number of containers in those Pods.

Figuring out the address

Just to be on the safe side, we'll confirm that the application deployed in the preview environment is indeed working as expected. To do that, we need to construct the auto-assigned address through which we can access the application.

```
PR_ADDR=$(kubectl \
    --namespace $PR_NAMESPACE \
    get ksvc jx-knative \
    --output jsonpath="{.status.url}")

echo $PR_ADDR
```

The output should be similar to the one that follows.

```
jx-knative.jx-vfarcic-jx-knative-pr-115.34.73.141.184.nip.io
```

> Please note that we did not have to *discover* the address. We could have gone to the GitHub pull request screen and clicked the *here* link. We'd need to add `/demo/hello` to the address, but that would still be easier than what we did. Still, I am a freak about automation and doing everything from a terminal screen, and I have the right to force you to do things my way, at least while you're following the exercises I prepared.

Sending a request to the application

Here comes the moment of truth.

```
curl "$PR_ADDR"
```

The output should already be the familiar `hello from` message. If by the time we sent the request, there was already a replica, it was simply forwarded there. If there wasn't, **Knative** created one.

In the next lesson, let's face a situation where we have to limit the serverless deployment and see how can that be achieved.