Naming Steps and Using Multi-Line Commands

In this lesson we will work on the jenkins-x.yml file and discuss how to name the steps and use multi-line commands.

We'll cover the following

A quick look at the pipeline

Let's take a quick look at the pipeline we have so far.

```
cd go-demo-6

cat jenkins-x.yml
```

```
buildPack: go
pipelineConfig:
  pipelines:
    pullRequest:
      build:
        preSteps:
        - command: make unittest
      promote:
        steps:
        - command: ADDRESS=`jx get preview --current 2>&1` make functest
```

We're extending the `go` pipeline defined as a build pack by adding two steps. We're executing unit tests (`make unittest`) before the build steps, and we added functional tests as a step after those pre-defined for the `promote` lifecycle. Both of those steps have issues we might want to fix those steps have issues we might want to fix.

So far, I have tried my best to hide a big problem with the execution of functional tests in our pipelines. They are executed after promotion, but there is no guarantee that our application is fully operational before we run the tests. If you create a pull request right now, without modifying the pipeline, you are likely going to experience failure. A pipeline run triggered by the creation of a pull request will fail because the functional tests are executed not when the application in a preview environment is fully up-and-running but after the “deploy” instruction is sent to Kubernetes. As you probably already know, when we execute `kubectl apply`, Kube API responds with the acknowledgment that it received the instruction, not with the confirmation that the actual state converged to the desired one.

There are quite a few ways to ensure that an application is rolled out before we run tests against it. We’re using **Helm** so you might be thinking that `--wait` should be enough. Usually, that would be the correct assumption if we were using **Tiller** (**Helm** server) in the cluster. But, Jenkins X does not use **Tiller** due to security and quite a few other issues. To be more precise, it does not use **Tiller** by default. You’ll need to specify `--no-tiller false` argument when installing Jenkins X. If you followed the instructions (Gists) as they are, your cluster does not have it, and Jenkins X uses **Helm** only to convert charts (templates) into “standard” Kubernetes `YAML` files. In that case, the simplified version of the deployment process consists of executing the `helm template` command followed by `kubectl apply`.

All in all, we do not have **Tiller** (**Helm** server) unless you customized the installation so we cannot use `--wait` to tell the deployment (promotion) process to exit only after the application rolls out. Instead, we’ll go back to basics and inject a step that will execute `kubectl rollout status`. It will serve two purposes:

1. First, it will make the pipeline wait until the application rolls out so that we can run functional tests without the fear that it will fail if the app is not yet fully up-and-running.
2. The second benefit is that the command will fail if rollout times out. If a command fails, pipeline fails as well so we’ll receive a notification that the new release could not roll out.

However, the problem is that we cannot merely execute `kubectl rollout status`. We need to know the namespace where our application is deployed, and each preview is deployed in a separate and unique one. Fortunately, namespaces are

created using a known pattern, so we will not have a problem figuring it out. Now we're getting to the real issue. We need to execute three commands, the first to figure out the namespace of a preview environment, the second to wait for a few seconds to ensure that the deployment was indeed created by the promotion process, and the third with `kubectl rollout status`. We cannot put them into separate commands because each is a different session in a separate container. If we store the namespace in an environment variable in one step (`command`), that variable would not be available in the next. We could solve that by converting those three commands into one, but that would result in a very long single line instruction that would be very hard to read. We want readable code, don't we?

All that brings us to something new we'll learn about pipelines. We'll try to specify a multi-line command.

Modifying `jenkins-x.yml`

Before we move on and implement what we just discussed, there is one more problem we'll try to solve. The few custom steps we added to the pipeline consisted only of `command` instructions. They are nameless. If you pay close attention, you probably noticed that Jenkins X auto-generated meaningless names for the steps we added. It doesn't have a crystal ball to figure out what we'd like to call each step. So, our second improvement will be to add names to our custom steps. Those coming from build packs are already named, so we only need to worry about those we add directly to our pipelines.

All in all, we'll add a step with a multi-line command that will make the pipeline wait until the application rolls out, and we'll make sure that all our steps have a name.

Here we go.

We'll create a new branch and replace the content of `jenkins-x.yml` with the command that follows.

> You'll see `# This is new` and `# This was modified` comments so that it's easier to figure out which parts of the pipeline are new or modified, and which are left unchanged.

> which are left unchanged.

```
git checkout -b better-pipeline

echo "buildPack: go
pipelineConfig:
  pipelines:
    pullRequest:
      build:
        preSteps:
        # This was modified
        - name: unit-tests
          command: make unittest
      promote:
        steps:
        # This is new
        - name: rollout
          command: |
            NS=\`echo jx-\$REPO_OWNER-go-demo-6-\$BRANCH_NAME | tr '[:upper:]' '[:lower:]'\`
            sleep 15
            kubectl -n \$NS rollout status deployment preview-preview --timeout 3m
        # This was modified
        - name: functional-tests
          command: ADDRESS=\`jx get preview --current 2>&1\` make functest
" | tee jenkins-x.yml
```

Multi-line command

We added the `rollout` step which contains a multi-line `command` that defines the namespace where the preview is deployed, sleeps for a while, and waits until the `deployment` is rolled out. All we had to do was specify pipe (`|`) and indent all the lines with the commands.

Since the branch is part of the namespace and it is in upper case (e.g., `PR-27`), the namespace is converted to lower case letters to comply with the standard. The second line of the `rollout` command sleeps for fifteen seconds. The reason for that wait is to ensure that the promotion build initiated by changing the repositories associated with automatic promotion to environments has started. Finally, the third line executes `kubectl rollout status`, thus forcing the pipeline to wait until the app is fully up and running before executing functional tests. Additionally, if the rollout fails, the pipeline will fail as well.

Names of the steps

In addition to the new step, we also added a `name` to all the steps we created so far. Now we have:

- `unit-tests`

- `rollout`
- `functional-tests`

Please note that you should not use spaces or “special” characters in names. Even though this is not a requirement, we prefer to have the names use lower case letters and words separated with dashes (`-`). We’ll see, later on, what Jenkins X does with those names.

Validating the changes

Before we proceed and push the updates to the new branch, we should validate whether our changes to the pipeline comply with the schema.

```
jx step syntax validate pipeline
```

Assuming that you didn’t have any typos, the output should claim that the format of the pipeline was `successfully validated`.

Pushing the changes to GitHub

Now we can push the changes to GitHub.

```
git add .

git commit -m "rollout status"

git push --set-upstream origin \
    better-pipeline
```

Since all the changes we did so far were related to the `pullRequest` pipeline, so we need to create a PR if we’re going to test that everything works as expected.

```
jx create pullrequest \
    --title "Better pipeline" \
    --body "What I can say?" \
    --batch-mode
```

Exploring the activities related to the PR

Next, we’ll explore the activities related to this pull request. We’ll do that by limiting the activities for the branch that corresponds with the PR.

> ⚠ Please replace `[...]` with `PR-[PR_ID]` (e.g., PR-72). You can extract the ID

> from the last segment of the pull request address.

```
export BRANCH=[...] # e.g., PR-72
```

Now we can, finally, take a look at the activities produced with the newly created pull request.

```
jx get activities \
    --filter go-demo-6/$BRANCH \
    --watch
```

The output is as follows.

```
...
vfarcic/go-demo-6/PR-103 #1        1m36s 1m28s Succeeded
  from build pack                  1m36s 1m28s Succeeded
    Credential Initializer Qtb2s   1m36s    0s Succeeded
    Working Dir Initializer Tchx7  1m35s    0s Succeeded
    Place Tools                    1m34s    0s Succeeded
    Git Source Vfarcic Go Demo ... 1m33s    1s Succeeded https://github.com/vfarcic/go-demo-6
    Git Merge                      1m32s    2s Succeeded
    Build Unit Tests               1m32s   25s Succeeded
    Build Make Linux               1m32s   27s Succeeded
    Build Container Build          1m31s   30s Succeeded
    Postbuild Post Build           1m31s   31s Succeeded
    Promote Make Preview           1m31s   51s Succeeded
    Promote Jx Preview             1m30s 1m17s Succeeded
    Promote Rollout                1m30s 1m18s Succeeded
    Promote Functional Tests       1m30s 1m22s Succeeded
  Preview                            18s         https://github.com/vfarcic/go-demo-6/pull/103
    Preview Application              18s         http://go-demo-6.jx-vfarcic-go-demo-6-pr-103.35.19
```

You'll notice that now we have a new step `Promote Rollout`. It is a combination of the name of the stage (`promote`) and the `name` of the step we defined earlier. Similarly, you'll notice that our unit and functional tests are now adequately named as well.

Feel free to stop watching the activities by pressing *ctrl+c*.

Inspecting the build logs

Just to be on the safe side, we'll take a quick look at the logs to confirm that the `rollout status` command is indeed executed.

```
jx get build logs --current
```

Even though we retrieved `build logs` quite a few times before, this time we used a new argument `--current`. With it, we do not need to specify the repository. Instead, `jx` assumed that the current folder is the repository name.

The relevant section of the output is as follows:

```
...
Showing logs for build vfarcic-go-demo-6-pr-159-server-1 stage from-build-pack and container step-p
deployment "preview-preview" successfully rolled out
...
```

We can see that the pipeline was `Waiting for deployment "preview-preview" rollout to finish` and that the pipeline continued executing only after all three replicas of the application were rolled out. We can also see that the functional tests were executed only after the rollout, thus removing potential failure that could be caused by running tests before the application is fully up-and-running or, even worse, running them against the older release if the new one is still not rolled out.

Now that we've seen how to define multi-line commands as well as how to name our steps, we'll explore how to work with environment variables and agents, in the next lesson.