# Extending Environment Pipelines

This lesson explains how to extend the environment pipelines to add integration tests.

**We'll cover the following**

## Where to add integration tests? #

We could add integration tests in the pipelines of our applications, but that's probably not the right place. The idea behind integration tests is to validate whether the system is integrated (hence the name). So, pipelines used to deploy to environments are probably better candidates for such tests. We already did a similar change with the static Jenkins X by modifying `Jenkinsfile` in staging and production repositories. Let's see how we can accomplish a similar effect through the new format introduced in serverless Jenkins X.

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

```
cd ..

GH_USER=[...]

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

cd environment-jx-rocks-staging
```

We cloned the `environment-jx-rocks-staging` repository that contains the always-up-to-date definition of our staging environment.

up-to-date definition of our staging environment.

## Inspecting `jenkins-x.yml`

Let’s take a look at the `jenkins-x.yml` file that controls the processes executed whenever a change is pushed to that repo.

```
cat jenkins-x.yml
```

The output is as follows.

```
env:
- name: DEPLOY_NAMESPACE
  value: jx-staging
pipelineConfig:
  agent: {}
  env:
  - name: DEPLOY_NAMESPACE
    value: jx-staging
  pipelines: {}
```

This pipeline might be a bit confusing since there is no equivalent to `buildPack: go` we saw before. On the one hand, the pipeline is too short to be a full representation of the processes that result in deployments to the staging environment. On the other, there is no indication that this pipeline extends a pipeline from a buildpack. A pipeline is indeed inherited from a buildpack, but that is hidden by Jenkins X “magic” that, in my opinion, is not intuitive.

## Inspecting the `pipeline.yaml` of the build pack `environment`

When that pipeline is executed, it will run whatever is defined in the build pack `environment`. Let’s take a look at the details.

```
curl https://raw.githubusercontent.com/jenkins-x-buildpacks/jenkins-x-kubernetes/master/packs/envi
```

```
extends:
  import: classic
  file: pipeline.yaml
agent:
  label: jenkins-go
  container: gcr.io/jenkinsxio/builder-go
pipelines:
  release:
    build:
      steps:
        - dir: env
          steps:
            - sh: jx step helm apply
            name: helm-apply

  pullRequest:

    build:
      steps:
        - dir: env
          steps:
            - sh: jx step helm build
              name: helm-build
```

This should feel familiar. It is functionally the same as environment pipelines we explored before when we used `Jenkinsfile`. Just as with the `go` buildpack, it extends a common `pipeline.yaml` located in the root of the `jenkins-x-classic` repository. The "classic" pipeline is performing common operations like checking out the code during the `setup` lifecycle, as well as some cleanup at the end of pipeline runs. If you're interested in details, please visit the jenkins-x-buildpacks/jenkins-x-classic repository and open `packs/pipeline.yaml`.

If we go back to the `curl` output, we can see that there are only two steps. The first is running in the `build` lifecycle of the `release` pipeline. It applies the chart to the environment specified in the variable `DEPLOY_NAMESPACE`. The second step `jx step helm build` that is actually used to lint the chart and confirm that it is syntactically correct. That step is also executed during the `build` lifecycle but inside the `pullRequest` pipeline.

## Adding integration tests #

If our mission is to add integration tests, they should probably run after the application is deployed to an environment. That means that we should add a step to the `release` pipeline and that it must run after the current build step that executes `jx step helm apply`. We could add a new step as a `post` mode of the `build` lifecycle, or we can use any other lifecycle executed after `build`. In any case, the only important thing is that our integration tests run after the deployment performed during the `build` lifecycle. To make things more interesting, we'll choose the `postbuild` lifecycle.

Please execute the command that follows.

```
cat jenkins-x.yml \
    | sed -e \
    's@pipelines: {}@pipelines:\
    release:\
      postBuild:\
        steps:\
        - command: echo "Running integ tests!!!"@g' \
    | tee jenkins-x.yml
```

As you can see, we won't run "real" tests, but simulate them through a simple `echo`. Our goal is to explore serverless Jenkins X pipelines and not to dive into testing, so I believe that a simple message like that one should be enough.

Now that we added a new step we can take a look at what we got.

```
cat jenkins-x.yml
```

The output is as follows.

```
env:
- name: DEPLOY_NAMESPACE
  value: jx-staging
pipelineConfig:
  env:
  - name: DEPLOY_NAMESPACE
    value: jx-staging
  pipelines:
    release:
      postBuild:
        steps:
        - command: echo "Running integ tests!!!"
```

As you can see, the `pipelines` section was expanded to include our new step following the same pattern as the one we saw when we extended the *go-demo-6* pipeline.

All that's left is to push the change before we confirm that everything works as expected.

```
git add .

git commit \
    --message "Added integ tests"

git push
```

Just as before, we need to wait for a few moments until the new pipeline run starts, before we can retrieve the logs.

```
jx get build logs \
    --filter environment-jx-rocks-staging \
    --branch master
```

You should be presented with a choice with a run to select. If the new one is present (e.g., `#3`), please select it. Otherwise, wait for a few moments more and

repeat the `jx get build logs` command.

The last line of the output should display the message `Running integ tests!!!`, thus confirming that the change to the staging environment pipeline works as expected.

That's it! We created a PR that runs units and functional tests, and now we also have a simulation of integration tests that will be executed every time anything is deployed to the staging environment. If that were a real application with real tests, our next action would be to approve the pull request and let the system do the rest.

## Approving the PR #

```
open "$PR_ADDR"
```

Feel free to go down the "correct" route of adding a colleague to the `OWNERS` file in the *master* branch and to the collaborators' list. After you're done, let the colleague write a comment with a slash command `/approve` or `/lgtm`. We already did those things in the previous chapter so you should know the drill. Or, you can be lazy (as I am), and just skip all that and click the *Merge pull request* button. Since the purpose of this chapter is not to explore ChatOps, you'll be forgiven for taking a shortcut.

When you're done merging the PR to the master branch, please click the *Delete branch* button in GitHub's pull request screen. There's no need to keep it any longer.

---

Let's wrap up this discussion and free up the used resources in the next lesson.