Extending Build Pack Pipelines

This lesson explains how we can extend the build pack pipelines to run tests.

We'll cover the following

We already saw that pipelines based on the new format have a single line: `buildPack: go`. To be more precise, those that are created based on a build pack are like that. While you can certainly create a pipeline from scratch, most use cases benefit from having a good base inherited from a build pack. For some, the base will be everything they need, but for others, it will not. There is a high probability that you will need to extend those pipelines by adding your own steps or even replacing a whole lifecycle (e.g., `promote`). Our next mission is to figure out how to accomplish that. We'll explore how to extend pipelines used with serverless Jenkins X.

Creating a new branch

As any good developer would (excluding those who work directly with the master branch), we'll start by creating a new branch.

```
git checkout -b extension
```

Increasing the value of `replicaCount` in `values.yaml`

Since we need to make a change to demonstrate how pipelines work, instead of making a minor modification like adding a random text to the `README.md` file, we'll do something we should have done a long time ago. We'll increase the number of replicas for our application. We'll ignore the fact that we should probably create a **HorizontalPodAutoscaler** and simply increase the `replicaCount` value in `values.yaml` from `1` to `3`.

Given that I want to make it easy for you, we'll execute a command that will make a change instead of asking you to open the `value.yaml` file in your favorite editor and update it manually.

```
cat charts/go-demo-6/values.yaml \
    | sed -e \
    's@replicaCount: 1@replicaCount: 3@g' \
    | tee charts/go-demo-6/values.yaml
```

Removing `http://` from the tests

We'll need to make another set of changes. The reason will become apparent later. For now, please note that the tests have a hard-coded `http://` followed with a variable that is used to specify the IP or the domain. As you'll see soon, we'll fetch a fully qualified address so we'll need to remove `http://` from the tests.

```
cat functional_test.go \
    | sed -e \
    's@fmt.Sprintf("http://@fmt.Sprintf("@g' \
    | tee functional_test.go

cat production_test.go \
    | sed -e \
    's@fmt.Sprintf("http://@fmt.Sprintf("@g' \
    | tee production_test.go
```

Now that we fixed the problem with the tests and increased the number of replicas (even though that was not necessary for the examples we'll run), we can proceed and start extending our out-of-the-box pipeline.

Running unit tests

As it is now, the pipeline inherited from the build pack does not contain validations. It does almost everything else we need it to do except to run tests. We'll start with unit tests.

Where should we add unit tests? We need to choose the pipeline, the lifecycle, and the mode. Typically, this would be the moment when I'd need to explain in great detail the syntax of the new pipeline format. Given that it would be a lengthy process, we'll add the step first, and explain what we did.

```
echo "buildPack: go
pipelineConfig:
  pipelines:
    pullRequest:
      build:
        preSteps:
        - command: make unittest" \
    | tee jenkins-x.yml
```

What did we do?

Unit tests should probably be executed when we create a pull request, so we defined a step inside the `pullrequest` pipeline. It could have been `release` or `feature` as well. We already explored under which conditions each of those are executed.

Each time a pipeline is executed, it goes through a series of lifecycles.

- **`setup`**: The `setup` lifecycle prepares the pipeline environment.
- **`setversion`**: The `setversion` configures the version of the release.
- **`prebuild`**: It prepares the environment for the build steps.
- **`build`**: It builds the binaries and other artifacts.
- **`postbuild`**: It is usually used for additional validations like security scanning.
- **`promote`**: It executes the process of installing the release to one or more environments.

In most cases, running unit tests does not require compilation nor a live application, so we can execute them early. The right moment is probably just before we build the binaries. That might compel you to think that we should select the `prebuild` lifecycle assuming that the building is done in the `build` phase. That would be true only if we are choosing the lifecycle we want to replace. While that

is possible as well, we'll opt for extending one of the phases. So, please defined our step inside the `build`.

We already commented that we want to extend the existing `build` lifecycle, so we chose to use `preSteps` mode. There are others that we'll discuss later. The `preSteps` mode will add a new step before those in the `build` lifecycle inherited from the build pack. That way, our unit tests will run before we build the artifacts.

The last entry is the `command` that will be executed as a step. It will run `make unittest`.

That's it. We added a new step to our `jenkins-x.yml`.

Inspecting the `jenkins-x.yml` file

Let's see what we got.

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

The output is as follows.

```
buildPack: go
pipelineConfig:
  pipelines:
    pullRequest:
      build:
        preSteps:
        - command: make unittest
```

We can see that the `buildPack: go` is still there so our pipeline will continue doing whatever is defined in that build pack. Below is the `pipelineConfig` section that, in this context, extends the one defined in the build pack.

The `pipelines` section extends the `build` lifecycle of the `pullRequest` pipeline. By specifying `preStep`, we know that it will execute `make unittest` before any of the out-of-the-box steps defined in the same lifecycle.

Pushing the changes to GitHub

Now that we extended our pipeline, we'll push the changes to GitHub, create a pull request, and observe the outcome.

```
git add .

git commit \
    --message "Trying to extend the pipeline"

git push --set-upstream origin extension

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

The last command created a pull request, and the address should be in the output. We'll put that URL together with the name of the branch into environment variables since we'll need them later.

> ⚠ Please replace the first `[...]` with the full address of the pull request (e.g., https://github.com/vfarcic/go-demo-6/pull/56) and the second with `PR-[PR_ID]` (e.g., PR-56). You can extract the ID from the last segment of the pull request address.

```
PR_ADDR=[...] # e.g., `https://github.com/vfarcic/go-demo-6/pull/56`

BRANCH=[...] # e.g., `PR-56`
```

Inspecting the build logs

The easiest way to deduce if our extended pipeline works correctly is through logs.

```
jx get build logs \
    --filter go-demo-6 \
    --branch $BRANCH
```

If you get the `error: no Tekton pipelines have been triggered which match the current filter` message, you were probably too fast, and the pipeline did not yet start running. In that case, wait for a few moments and re-execute the `jx get build logs` command.

The output is too big to be presented here. What matters is the part that follows.

```
...
=== RUN   TestMainUnitTestSuite
=== RUN   TestMainUnitTestSuite/Test_HelloServer_Waits_WhenDelayIsPresent
=== RUN   TestMainUnitTestSuite/Test_HelloServer_WritesHelloWorld
```

```
=== RUN   TestMainUnitTestSuite/Test_HelloServer_WritesNokEventually
=== RUN   TestMainUnitTestSuite/Test_HelloServer_WritesOk
=== RUN   TestMainUnitTestSuite/Test_PersonServer_InvokesUpsertId_WhenPutPerson
=== RUN   TestMainUnitTestSuite/Test_PersonServer_Panics_WhenFindReturnsError
=== RUN   TestMainUnitTestSuite/Test_PersonServer_Panics_WhenUpsertIdReturnsError
=== RUN   TestMainUnitTestSuite/Test_PersonServer_WritesPeople
=== RUN   TestMainUnitTestSuite/Test_RunServer_InvokesListenAndServe
=== RUN   TestMainUnitTestSuite/Test_SetupMetrics_InitializesHistogram
--- PASS: TestMainUnitTestSuite (0.01s)
    --- PASS: TestMainUnitTestSuite/Test_HelloServer_Waits_WhenDelayIsPresent (0.00s)
    --- PASS: TestMainUnitTestSuite/Test_HelloServer_WritesHelloWorld (0.00s)
    --- PASS: TestMainUnitTestSuite/Test_HelloServer_WritesNokEventually (0.00s)
    --- PASS: TestMainUnitTestSuite/Test_HelloServer_WritesOk (0.00s)
    --- PASS: TestMainUnitTestSuite/Test_PersonServer_InvokesUpsertId_WhenPutPerson (0.00s)
    --- PASS: TestMainUnitTestSuite/Test_PersonServer_Panics_WhenFindReturnsError (0.00s)
    --- PASS: TestMainUnitTestSuite/Test_PersonServer_Panics_WhenUpsertIdReturnsError (0.00s)
    --- PASS: TestMainUnitTestSuite/Test_PersonServer_WritesPeople (0.00s)
    --- PASS: TestMainUnitTestSuite/Test_RunServer_InvokesListenAndServe (0.00s)
    --- PASS: TestMainUnitTestSuite/Test_SetupMetrics_InitializesHistogram (0.00s)
PASS
ok      go-demo-6       0.011s
...
```

We can see that our unit tests are indeed executed.

In our context, what really matters is the output after the tests. It shows that the application binary and container images were built. That proves that the new step didn't replace anything, but that it was added among those defined in the build pack. Since we specified the mode as part of `preSteps`, it was added before the existing steps in the `build` lifecycle.

As I'm sure you already know, unit tests are often not enough and we should add some form of tests against the live application.

Creating a target in `Makefile` for functional tests

We should probably add functional tests to the pipeline. But, before we do that, we need to create one more target in the `Makefile`.

> ⚠ Remember what we said before about `Makefile`. It expects tabs as indentation. Please make sure that the command that follows is indeed using tabs and not spaces if you're typing the commands instead of copying and pasting from the Gist.

```
echo 'functest:
	CGO_ENABLED=$(CGO_ENABLED) $(GO) \\
	test -test.v --run FunctionalTest \\
```

```
        --cover
' | tee -a Makefile
```

Now we can add the `functest` target to the `pullrequest` pipeline. But, before we do that, we need to decide on the best moment to run them. Since they require a new release to be installed, and we already know from the past experience that installations and upgrades are performed through promotions, we should already have an idea where to put them. So, we'll add functional tests to the `pullrequest` pipeline, into the `promote` lifecycle, and with the `post` mode. That way, those tests will run after the promotion is finished, and our release based on the PR is up-and-running.

But, there is a problem that we need to solve or, to be more precise, there is an improvement we could do.

Modifying `jenkins-x.yml` to add the dynamic address of the current PR

Functional tests need to know the address of the application under tests. Since each pull request is deployed into its own namespace and the app is exposed through a dynamically created address, we need to figure out how to retrieve that URL. Fortunately, there is a command that allows us to retrieve the address of the current pull request. The bad news is that we cannot (easily) run it locally, so you'll need to trust me when I say that `jx get preview --current` will return the full address of the PR deployed with the `jx preview` command executed as the last step in the `promote` lifecycle of the `pullrequest` pipeline.

With that in mind, the command we will execute is as follows.

```
echo '      promote:
        steps:
        - command: ADDRESS=`jx get preview --current 2>&1` make functest' | \
    tee -a jenkins-x.yml
```

Just as before, the output confirms that `jenkins-x.yml` was updated, so let's take a peek at how it looks now.

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

The output is as follows.

```
buildPack: go
```

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

As you can see, the new step follows the same pattern. It is defined inside the `pullRequest` pipeline as the `promote` lifecycle and inside the `steps` mode. You can easily conclude that `preSteps` are executed before those defined in the same lifecycle of the build pack, and `steps` are running after.

Pushing the changes to GitHub

Now, let's push the change before we confirm that everything works as expected.

```
git add .

git commit \
    --message "Trying to extend the pipeline"

git push
```

Retrieving logs for functional tests

Please wait for a few moments until the new pipeline run starts, before we retrieve the logs.

```
jx get build logs \
    --filter go-demo-6 \
    --branch $BRANCH
```

You should be presented with a choice of pipeline runs. The choices should be as follows.

```
> vfarcic/go-demo-6/PR-64 #2 serverless-jenkins
  vfarcic/go-demo-6/PR-64 #1 serverless-jenkins
```

If you do not see the new run, please cancel the command with *ctrl+c*, wait for a while longer, and re-execute the `jx get build logs` command.

The result of the functional tests should be near the bottom of the output. You should see something similar to the output that follows.

```
...
CGO_ENABLED=0 GO15VENDOREXPERIMENT=1 go \
test -test.v --run FunctionalTest \
--cover
=== RUN   TestFunctionalTestSuite
=== RUN   TestFunctionalTestSuite/Test_Hello_ReturnsStatus200
2019/04/26 10:58:31 Sending a request to http://go-demo-6.jx-vfarcic-go-demo-6-pr-57.34.74.193.252
=== RUN   TestFunctionalTestSuite/Test_Person_ReturnsStatus200
2019/04/26 10:58:31 Sending a request to http://go-demo-6.jx-vfarcic-go-demo-6-pr-57.34.74.193.252
--- PASS: TestFunctionalTestSuite (0.26s)
    --- PASS: TestFunctionalTestSuite/Test_Hello_ReturnsStatus200 (0.13s)
    --- PASS: TestFunctionalTestSuite/Test_Person_ReturnsStatus200 (0.13s)
PASS
coverage: 1.4% of statements
ok      go-demo-6       0.271s
```

What happens if the run fails?

While we are still exploring the basics of extending build pack pipelines, we might just as well take a quick look at what happens if a pipeline run fails. Instead of deliberately introducing a bug in the code of the application, we'll add another round of tests, but this time in a way that will certainly fail.

```
echo '        - command: ADDRESS=http://this-domain-does-not-exist.com make functest' | \
    tee -a jenkins-x.yml
```

As you can see, we added the execution of the same tests. The only difference is that the address of the application under test is now `http://this-domain-does-not-exist.com`. That will surely fail and allow us to see what happens when something goes wrong.

Let's push the changes.

```
git add .

git commit \
    --message "Added sully tests"

git push
```

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

```
jx get build logs \
    --filter go-demo-6 \
    --branch $BRANCH
```

If the previous run was #2, you should be presented with a choice to select run

If the previous run was `#2`, you should be presented with a choice to select run `#3`. Please do so.

At the very bottom of the output, you should see that the pipeline run failed. That should not be a surprise since we specified an address that does not exist. We wanted it to fail, but not so that we can see it in the logs. Instead, the goal is to see how we would be notified that something went wrong.

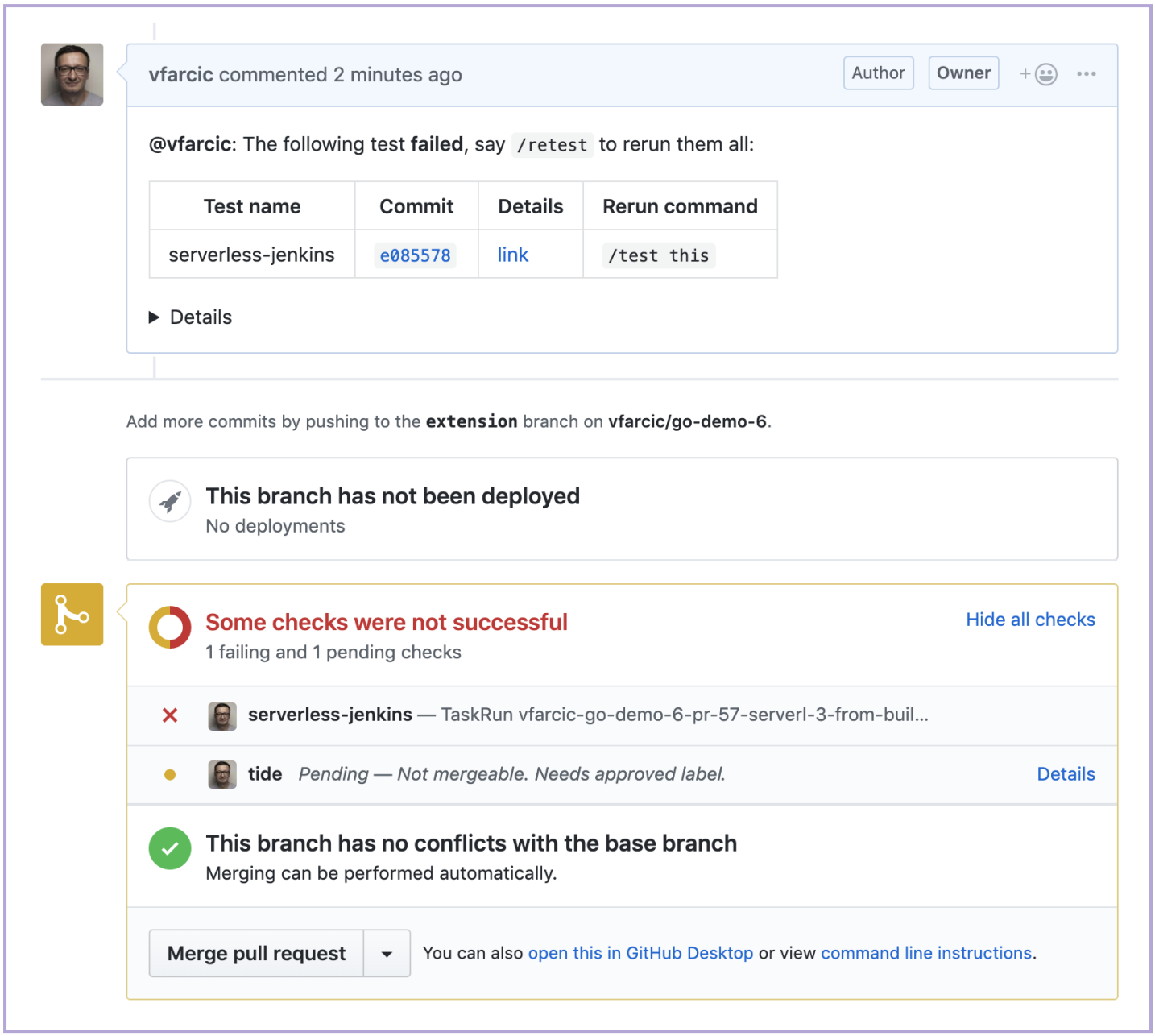
Let's see what we get if we open pull request in GitHub.

```
open "$PR_ADDR"
```

We can see the whole history of everything that happened to that pull request, including the last comment that states that the `serverless-jenkins` test failed. Later on, we'll explore how to run multiple pipelines in parallel. For now, it might be important to note that GitHub does not know which specific test failed, but that it treats the entire pipeline as a single unit. Nevertheless, it is recorded in GitHub that there is an issue. Its own notification mechanism should send emails to all interested parties, and we can see from the logs what went wrong. Later on we'll explore the new UI designed specifically for Jenkins X. For now, we can get all the information we need without it.

You will also notice that the comment states that we can re-run tests by writing `/retest` or `/test this` as a comment. That's how we can re-execute the same pipeline in case a failure is not due to a "real" problem but caused by flaky tests or some temporary issue instead.

> ⚠ At the time of this writing the `/retest` and `/test this` commands do not yet work. But that does not mean that they won't by the time you're reading this, so please try it out.



A pull request with failed tests

Removing the failing test

Before we move into the next subject, we'll remove the step with the silly test that always fails, and leave the repository in a good state.

```
cat jenkins-x.yml \
  | sed '$ d' \
  | tee jenkins-x.yml

git add .

git commit \
    --message "Removed the silly test"

git push
```

The first command removed the last line from `jenkins-x.yml`, and the rest is the "standard" `push` to GitHub.

Now that our repository is back into the "working" state, we should explore the last available mode.

We used the `pre` mode to inject steps before those inherited from a build pack. Similarly, we saw that we can inject them after using the `post` mode. If we'd like to replace all the steps from a build pack lifecycle, we could select the `replace` mode. There's probably no need to go through an exercise that would show that in action since the process is the same as for any other mode. The only difference is in what is added to `jenkins-x.yml`.

Before you start replacing lifecycles, be aware that you'd need to redo them completely. If, for example, you replace the steps in the `build` lifecycle, you'd need to make sure that you are implementing all the steps required to build your application. We rarely do that since Jenkins X build packs come with good defaults that are seldom removed. Nevertheless, there are cases when we do NOT want what Jenkins X offers and we might wish to reimplement a complete lifecycle by specifying steps through the `replace` mode.

Now that we added unit and functional tests, we should probably add some kind of integration tests as well.