Overriding Pipelines, Stages, and Steps and Implementing Loops

This lesson explains how to override complete pipelines and individual stages or steps of pipelines. We also learn how to implement loops.

We'll cover the following

Our pipeline is currently building a Linux binary of our application before adding it to a container image. **What if we want to distribute the application also as executables for different operating systems?** We could provide that same binary, but that would work only for Linux users since that is the architecture it is currently built for. We might want to extend the reach to Windows and macOS users as well, and that would mean that we'd need to build two additional binaries. **How could we do that?**

How can we create binaries for all three OS?

Since our pipeline is already building a Linux executable through a step inherited from the build pack, we can add two additional steps that would build for the other two operating systems. But that approach would result in *go-demo-6* binary for Linux, and our new steps would, let's say, build *go-demo-6_Windows* and *go-demo-6_darwin*. That, however, would result in "strange" naming. In that context, it would make much more sense to have *go-demo-6_linux* instead of *go-demo-6*. We

It would make much more sense to have *go-demo-6_linux* instead of *go-demo-6*. We could add yet another step that would rename it, but then we'd be adding

unnecessary complexity to the pipeline that would make those reading it wonder what we're doing. We could build the Linux executable again, but that would result in duplication of the steps.

A better solution is to remove the build step inherited from the build pack and add those that build the three binaries in its place. That would be a more optimum solution. One step removed, three steps added. But those steps would be nearly the same, the only difference would be an argument that defines each OS. Instead of having three steps, one for building a binary for each operating system, we'll create a loop that will iterate through values that represent operating systems and execute a step that builds the correct binary.

This might be too much to swallow at once, so we'll break it into two tasks. First, we'll try to figure out how to remove a step from the inherited build pack pipeline. If we're successful, we'll put the loop of steps in its place.

Let's get started.

We can use the `overrides` instruction to remove or replace any inherited element. We'll start with the simplest version of the instruction and improve it over time.

Overriding the `release` pipeline

Please execute the command that follows to create a new version of `jenkins-x.yml`.

```
echo "buildPack: go
pipelineConfig:
  env:
  - name: CODECOV_TOKEN
    valueFrom:
      secretKeyRef:
        key: token
        name: codecov
  pipelines:
    pullRequest:
      build:
        preSteps:
        - name: unit-tests
          command: make unittest
        - name: code-coverage
          command: codecov.sh
          agent:
            image: vfarcic/codecov
      promote:
        steps:
        - 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
        - name: functional-tests
          command: ADDRESS=\`jx get preview --current 2>&1\` make functest
    # This is new
    overrides:
    - pipeline: release
" | tee jenkins-x.yml
```

All we did was to add two lines at the end of the pipeline. We specified that we want to override the `release` pipeline.

Just as with the previous examples, we'll validate the syntax, push the changes to GitHub, and observe the result by watching the activities.

```
jx step syntax validate pipeline

git add .

git commit -m "Multi-architecture"

git push

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

The output of the last command, limited to the relevant parts, is as follows.

```
...
vfarcic/go-demo-6/master #3                          36s 30s Succeeded
  meta pipeline                                      36s 20s Succeeded
    Credential Initializer Bsggw                     36s  0s Succeeded
    Working Dir Initializer 5n6mx                    36s  1s Succeeded
    Place Tools                                      35s  1s Succeeded
    Git Source Meta Vfarcic Go Demo 6 Master R ...   34s  5s Succeeded https://github.com/vfarcic/go
    Git Merge                                        29s  1s Succeeded
    Merge Pull Refs                                  28s  1s Succeeded
    Create Effective Pipeline                        27s  3s Succeeded
    Create Tekton Crds                               24s  8s Succeeded
  from build pack                                    14s  8s Succeeded
    Credential Initializer Fw774                     14s  0s Succeeded
    Working Dir Initializer S7292                    14s  1s Succeeded
    Place Tools                                      13s  1s Succeeded
    Git Source Vfarcic Go Demo 6 Master Releas ...   12s  5s Succeeded https://github.com/vfarcic/go
    Git Merge                                         7s  1s Succeeded
    Setup Jx Git Credentials                          6s  0s Succeeded
```

Judging from the output of the latest activity, the number of steps dropped drastically. That's the expected behavior since we told Jenkins X to override the release pipeline with nothing. We have not specify replacement steps that should

be executed instead of those inherited from the build pack. So, the only steps executed are those related to Git since they are universal and not tied to any specific pipeline.

Please press *ctrl+c* to stop watching the activities.

Overriding the `build` stage of the `release` pipeline

In our case, overriding the whole `release` pipeline might be too much. We do not have a problem with all of the inherited steps, but only with the `build` stage inside the `release` pipeline. So, we'll override only that one.

Since we are about to modify the pipeline yet again, we might want to add the `rollout` command to the `release` pipeline as well. It'll notify us if a release cannot be rolled out.

Off we go.

```
echo "buildPack: go
pipelineConfig:
  env:
  - name: CODECOV_TOKEN
    valueFrom:
      secretKeyRef:
        key: token
        name: codecov
  pipelines:
    pullRequest:
      build:
        preSteps:
        - name: unit-tests
          command: make unittest
        - name: code-coverage
          command: codecov.sh
          agent:
            image: vfarcic/codecov
      promote:
        steps:
        - 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
        - name: functional-tests
          command: ADDRESS=\`jx get preview --current 2>&1\` make functest
    overrides:
    - pipeline: release
      # This is new
      stage: build
    # This is new
```

```
    release:
      promote:
        steps:
        - name: rollout
          command: |
            sleep 30
            kubectl -n jx-staging rollout status deployment jx-go-demo-6 --timeout 3m
" | tee jenkins-x.yml
```

We added the `stage: build` instruction to the existing override of the `release` pipeline. We also added the `rollout` command as yet another step in the `promote` stage of the `release` pipeline.

You probably know what comes next. We'll validate the pipeline syntax, push the changes to GitHub, and observe the activities hoping that they will tell us whether the change was successful or not.

```
jx step syntax validate pipeline

git add .

git commit -m "Multi-architecture"

git push

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

The output, limited to the latest build, is as follows.

```
...
vfarcic/go-demo-6/master #5                          3m46s 2m45s Succeeded Version: 1.0.446
  meta pipeline                                      3m46s   21s Succeeded
    Credential Initializer L6kh9                     3m46s    0s Succeeded
    Working Dir Initializer Khkf6                    3m46s    0s Succeeded
    Place Tools                                      3m46s    1s Succeeded
    Git Source Meta Vfarcic Go Demo 6 Master R ... 3m45s    5s Succeeded https://github.com/vfarci
    Git Merge                                        3m40s    1s Succeeded
    Merge Pull Refs                                  3m39s    0s Succeeded
    Create Effective Pipeline                        3m39s    4s Succeeded
    Create Tekton Crds                               3m35s   10s Succeeded
  from build pack                                    3m23s 2m22s Succeeded
    Credential Initializer 5cw8t                     3m23s    0s Succeeded
    Working Dir Initializer D99p2                    3m23s    1s Succeeded
    Place Tools                                      3m22s    1s Succeeded
    Git Source Vfarcic Go Demo 6 Master Releas ... 3m21s    6s Succeeded https://github.com/vfarci
    Git Merge                                        3m15s    0s Succeeded
    Setup Jx Git Credentials                         3m15s    0s Succeeded
    Promote Changelog                                3m15s    8s Succeeded
    Promote Helm Release                              3m7s   18s Succeeded
    Promote Jx Promote                               2m49s 1m32s Succeeded
    Promote Rollout                                  1m17s   16s Succeeded
  Promote: staging                                   2m43s 1m26s Succeeded
```

```
PullRequest                                  2m43s 1m26s Succeeded  PullRequest: ...
Update                                       1m17s    0s Succeeded
Promoted                                     1m17s    0s Succeeded  Application ...
```

The first thing we can note is that the number of steps in the activity is closer to what we're used to. Now that we are not overriding the whole pipeline but only the `build` stage, almost all the steps inherited from the build pack are there. Only those related to the `build` stage are gone, simply because we limited the scope of the `overrides` instruction.

Please stop watching the activities by pressing *ctrl+c*.

We are getting closer to our goal. We just need to figure out how to override a specific step with the new one that will build binaries for all operating systems. But, **how are we going to override a particular step if we do not know which one it is?** We could find all the steps of the pipeline by visiting the repositories that host build packs. But that would be tedious. We'd need to go to a few repositories, check the source code of the related pipelines, and combine the result with the one we're rewriting right now. There must be a better way to get an insight into the pipeline related to *go-demo-6*.

Reverting the changes

Before we move on and try to figure out how to retrieve the full definition of the pipeline, we'll revert the current version to the state before we started "playing" with `overrides`. You'll see the reason for this soon.

```
echo "buildPack: go
pipelineConfig:
  env:
  - name: CODECOV_TOKEN
    valueFrom:
      secretKeyRef:
        key: token
        name: codecov
  pipelines:
    pullRequest:
      build:
        preSteps:
        - name: unit-tests
          command: make unittest
        - name: code-coverage
          command: codecov.sh
          agent:
            image: vfarcic/codecov
      promote:
        steps:
        - 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
        - name: functional-tests
          command: ADDRESS=\`jx get preview --current 2>&1\` make functest
    # Removed overrides
    release:
      promote:
        steps:
        - name: rollout
          command: |
            sleep 30
            kubectl -n jx-staging rollout status deployment jx-go-demo-6 --timeout 3m
" | tee jenkins-x.yml
```

The `jx step syntax effective` command

Now that we are back to where we were before we discovered `overrides`, we can learn about yet another command.

```
jx step syntax effective
```

The output is the “effective” version of our pipeline. You can think of it as a merge of our pipeline combined with those it extends (e.g., from build packs). It is the same final version of the YAML pipeline Jenkins X would use as a blueprint for creating Tekton resources.

The reason we’re outputting the effective pipeline lies in our need to find the name of the step currently used to build the Linux binary of the application. If we find its name, we will be able to override it.

The output, limited to the relevant parts, is as follows.

```
buildPack: go
pipelineConfig:
  ...
  pipelines:
    ...
    release:
      pipeline:
        ...
        stages:
        - agent:
            image: go
          name: from-build-pack
          steps:
          ...
          - command: make build
            dir: /workspace/source
            image: go
            name: build-make-build
          ...
```

We know that the step we're looking for is somewhere inside the `release` pipeline, so that should limit the scope. If we take a look at the steps inside, we can see that one of them executes the command `make build`. That's the one we should remove or, to be more precise, override.

You'll notice that the names of the steps are different in the effective version of the pipeline. For example, the `rollout` step we created earlier is now called `promote-rollout`. In the effective version of the pipelines, the step names are always prefixed with the stage. As a result, when we see the activities retrieved from Tekton pipeline runs, we see the two (stage and step) combined.

There's one more explanation I promised to deliver.

> ### 🔍 Why did we revert the pipeline to the version before we added overrides?
>
> If we didn't revert the pipeline, we wouldn't be able to find the step we were looking for. The whole `build` stage from the `release` pipeline would be gone since we had it overridden to nothing.

Now, let's get back to our mission. We know that the step we want to override in the effective version of the pipeline is named `build-make-build`. Since we know that the names are prefixed with the stage, we can deduce that the stage is `build` and the name of the step is `make-build`.

Now that it's clear what to override, let's talk about loops.

Adding the loop

We can tell Jenkins X to loop between values and execute a step or a set of steps in each iteration. An example of the syntax could be as follows:

```
- loop:
    variable: COLOR
    values:
    - yellow
    - red
    - blue
    - purple
```

```
    - green
    steps:
    - command: echo "The color is $COLOR"
```

If we had that loop inside our pipeline, it would execute a single step five times, once for each of the `values` of the `loop`. What we put inside the `steps` section is up to us, and the only important thing to note is that `steps` in the `loop` use the same syntax as the `steps` anywhere else (e.g., in one of the stages).

Now, let's see whether we can combine `overrides` with `loop` to accomplish our goal of building a binary for each of the "big" three operating systems.

Please execute the command that follows to update `jenkins-x.yml` with the new version of the pipeline.

```
echo "buildPack: go
pipelineConfig:
  env:
  - name: CODECOV_TOKEN
    valueFrom:
      secretKeyRef:
        key: token
        name: codecov
  pipelines:
    pullRequest:
      build:
        preSteps:
        - name: unit-tests
          command: make unittest
        - name: code-coverage
          command: codecov.sh
          agent:
            image: vfarcic/codecov
      promote:
        steps:
        - name: rollout
          command: |
            NS=\`echo jx-\$REPO_OWNER-go-demo-6-\$BRANCH_NAME | tr '[:upper:]' '[:lower:]'\`
            sleep 30
            kubectl -n \$NS rollout status deployment preview-preview --timeout 3m
        - name: functional-tests
          command: ADDRESS=\`jx get preview --current 2>&1\` make functest
    overrides:
    - pipeline: release
      # This is new
      stage: build
      name: make-build
      steps:
      - loop:
          variable: GOOS
          values:
          - darwin
          - linux
          - windows
          steps:
          - name: build
            command: CGO_ENABLED=0 GOOS=\${GOOS} GOARCH=amd64 go build -o bin/go-demo-6 \${GOOS} m
```

```
    release:
      promote:
        steps:
        - name: rollout
          command: |
            sleep 15
            kubectl -n jx-staging rollout status deployment jx-go-demo-6 --timeout 3m
" | tee jenkins-x.yml
```

This time we are overriding the step `make-build` in the `build` stage of the `release` pipeline. The "old" step will be replaced with a `loop` that iterates over the values that represent operating systems. Each iteration of the loop contains the `GOOS` variable with a different value and executes the `command` that uses it to customize how we build the binary. The end result should be *go-demo-6_* that is executable with the unique suffix that tells us where it is meant to be used (e.g., `linux`, `darwin`, or `windows`)s.

> 🔍 If you're new to Go, the compiler uses environment variable `GOOS` to determine the target operating system for a build.

Next, we'll validate the pipeline and confirm that we did not introduce a typo incompatible with the supported syntax.

```
jx step syntax validate pipeline
```

Changing the reference in the `Dockerfile`

There's one more thing we should fix. In the past, our pipeline was building the *go-demo-6* binary, and now we changed that to *go-demo-6_linux*, *go-demo-6_darwin*, and *go-demo-6_windows*. Intuition would tell us that we might need to change the reference to the new binary in `Dockerfile`, so let's take a quick look at it.

```
cat Dockerfile
```

The output is as follows.

```
FROM scratch
EXPOSE 8080
ENTRYPOINT ["/go-demo-6"]
COPY ./bin/ /
```

The last line will copy all the files from the `bin/` directory to the container root

This would introduce at least two problems. First of all, there is no need to have all three binaries inside the container images we're building. That would make them bigger for no good reason. The second issue with the way binaries are copied is the `ENTRYPOINT`. It expects `/go-demo-6`, instead of `go-demo-6_linux` that we are building now. Fortunately, the fix for both of the issues is straightforward. We can change the `COPY` instruction in `Dockerfile` so that only `go-demo-6_linux` is copied and that it is renamed to `go-demo-6` during the process. That will help us avoid copying unnecessary files and will still fulfill the `ENTRYPOINT` requirement.

```
cat Dockerfile \
    | sed -e \
    's@/bin/ /@/bin/go-demo-6_linux /go-demo-6@g' \
    | tee Dockerfile
```

Pushing changes and observing the activities

Now we're ready to push the change to GitHub and observe the new activity that will be triggered by that action.

```
git add .

git commit -m "Multi-architecture"

git push

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

The output, limited to the latest build, is as follows.

```
...
vfarcic/go-demo-6/master #6                          5m32s 5m18s Succeeded Version: 1.0.447
  meta pipeline                                      5m32s   24s Succeeded
    Credential Initializer Pg5cf                     5m32s    0s Succeeded
    Working Dir Initializer Lzpdb                    5m32s    2s Succeeded
    Place Tools                                      5m30s    1s Succeeded
    Git Source Meta Vfarcic Go Demo 6 Master R ...   5m29s    4s Succeeded https://github.com/vfarci
    Git Merge                                        5m25s    1s Succeeded
    Merge Pull Refs                                  5m24s    0s Succeeded
    Create Effective Pipeline                        5m24s    4s Succeeded
    Create Tekton Crds                               5m20s   12s Succeeded
  from build pack                                     5m6s 4m52s Succeeded
    Credential Initializer P5wrz                      5m6s    0s Succeeded
    Working Dir Initializer Frrq2                     5m6s    0s Succeeded
    Place Tools                                       5m6s    1s Succeeded
    Git Source Vfarcic Go Demo 6 Master Releas ...    5m5s    9s Succeeded https://github.com/vfarci
    Git Merge                                        4m56s    1s Succeeded
    Setup Jx Git Credentials                         4m55s    0s Succeeded
```

```
    Build1                                  4m55s   42s Succeeded
    Build2                                  4m13s   16s Succeeded
    Build3                                  3m57s   33s Succeeded

    Build Container Build                   3m24s    5s Succeeded
    Build Post Build                        3m19s    0s Succeeded
    Promote Changelog                       3m19s    7s Succeeded
    Promote Helm Release                    3m12s   16s Succeeded
    Promote Jx Promote                      2m56s 1m31s Succeeded
    Promote Rollout                         1m25s 1m11s Succeeded
  Promote: staging                          2m50s 1m25s Succeeded
    PullRequest                             2m50s 1m24s Succeeded  PullRequest: ...f157af83
    Update                                  1m25s    0s Succeeded
    Promoted                                1m25s    0s Succeeded  Application is at: ...
```

We can make a few observations. The `Build Make Build` step is now gone, so the override worked correctly. We have `Build1`, `Build2`, and `Build3` in its place. Those are the three steps created as a result of having the loop with three iterations. Those are the steps that are building `windows`, `linux`, and `darwin` binaries. Finally, we can observe that the `Promote Rollout` step is now shown as `succeeded`, thus providing a clear indication that the new building process (steps) worked correctly. Otherwise, the new release could not roll out, and that step would fail.

Please stop watching the activities by pressing *ctrl+c*.

Before we move on, I must confess that I wouldn't make the same implementation as the one we just explored. Instead, I'd rather change the `build` target in `Makefile`. That way, there would be no need for any change to the pipeline. The build pack step would continue building by executing that `Makefile` target so there would be no need to override anything, and there would certainly be no need for a loop. Now, before you start throwing stones at me, I must also state that `overrides` and `loop` can come in handy in some other scenarios. I had to come up with an example that would introduce you to `overrides` and `loop`, and that ended up being the need to cross-compile binaries, even if it could be accomplished in an easier and a better way. Remember, the "real" goal was to learn those constructs, and not how to cross-compile with Go.

Next, let's see how we can work with pipelines without buildpacks.