Working with Environment Variables and Agents

This lesson shows how to create a new pipeline using environment variables and agents.

We'll cover the following

Let's say that we want to add code coverage to our pipeline. We could do that with many different tools. However, since the goal is not to teach you how to set up code coverage or to explore which tool is better, we'll skip the selection process and use the **Codecov** service. Keep in mind that I'm not saying that it is better than the others nor that you must use a service for that, but rather that I needed an example to demonstrate a few new pipeline instructions. **Codecov** seems like the right candidate.

Integrating our pipeline with Codecov

What do we need to do to integrate our pipeline with the **Codecov** service? If we check their instruction for Go applications, we'll see that we should output code coverage to a text file. Since I'm trying to make the examples as agnostic to programming languages as possible, we'll skip changing the `Makefile` that contains testing targets assuming that you'll read the **Codecov** instructions later on if you

choose to use it. So, instead of telling you to apply specific changes to `Makefile`, we'll download a Gist I prepared.

```
curl -o Makefile \
    https://gist.githubusercontent.com/vfarcic/313bedd36e863249cb01af1f459139c7/raw
```

Now that we put Go internals out of the way, there are a few other things we need to do. We need to run a script provided by **Codecov**. That script expects a token that will authenticate us. So, we need three things.

1. A container image with the script
2. An environment variable with the token.
3. A pipeline step that will execute the script that will send the code coverage results to **Codecov**.

Retrieving a Codecov token

Let's start by retrieving a **Codecov** token for our *go-demo-6* repository.

```
open "https://codecov.io/"
```

I will skip giving you instructions on how to add your *go-demo-6* fork into **Codecov**. I'm sure that you will be able to sign up and follow the instructions provided on the site. Optionally, you can install their GitHub App (the message will appear at the top). What matters the most is the token you'll receive once you add the fork of the *go-demo-6* repository to **Codecov**.

> ⚠ Please replace `[...]` with the **Codecov** token for the *go-demo-6* repository.

```
export CODECOV_TOKEN=[...]
```

Providing information to Codecov

There are a few ways we can provide the info **Codecov** needs to calculate code coverage. We'll use a Shell script they provide. To make things simple, I already created a container image that contains the script. It is a straightforward one, and you can explore `Dockerfile` used to create the image from the vfarcic/codecov repository.

```
open "https://github.com/vfarcic/codecov"
```

Open Dockerfile, and you'll see that the definition is as follows.

```
FROM alpine:3.9

RUN apk update && apk add bash curl git
RUN curl -o /usr/local/bin/codecov.sh https://codecov.io/bash
RUN chmod +x /usr/local/bin/codecov.sh
```

I already created a public image `vfarcic/codecov` based on that definition, so there is no action needed on your part. We can start using it right away.

So, what do we need to integrate Codecov with our pipeline?

- We need to define the environment variable `CODECOV_TOKEN` required by the `codecov.sh` script.
- We'll also need to add a new step that will execute that script.

> We already know how to add steps, but this time there is a twist. We need to make sure that the new step is executed inside a container created from the `vfarcic/codecov` image converted into a pipeline agent.

We need to figure out how to define environment variables as well as to define an agent based on a custom container image.

Creating the pipeline

Please execute the command that follows to create the full pipeline that contains the necessary changes.

```
echo "buildPack: go
pipelineConfig:
  # This is new
  env:
  - name: CODECOV_TOKEN
    value: \"$CODECOV_TOKEN\"
  pipelines:
    pullRequest:
      build:
        preSteps:
        - name: unit-tests
          command: make unittest
        # This is new
        - 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
" | tee jenkins-x.yml
```

Near the top of the pipeline is the `env` section that, in this case, defines a single variable `CODECOV_TOKEN`. We could have moved the `env` definition inside a pipeline, stage, or a step to limit its scope. As it is now, it will be available in all the steps of the pipeline.

We added a new step `code-coverage` inside the `pullRequest` pipeline. What makes it "special" is the `agent` section with the `image` set to `vfarcic/codecov`. As a result, that step will be executed inside a container based on that image. Just as with `env`, we could have defined `agent` in `pipelineConfig`, and then all the steps in all the pipelines would run in containers based on that image. Or we could have defined it on the level of a single pipeline or a stage.

For the sake of diversity, we defined an environment variable available in all the steps, and an agent that will be used in a single step.

Validating the syntax of pipeline

Before we proceed, we'll check whether the syntax of the updated pipeline is correct.

```
jx step syntax validate pipeline
```

Pushing changes and watching activities

Next, we'll push the changes to GitHub and watch the activity that will be initiated by it.

```
git add .

git commit -m "Code coverage"

git push
```

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

After the unit tests are executed, we should see the `Code Coverage` step of the `build` stage/lifecycle change the status to `succeeded`.

Feel free to cancel the watcher by pressing *ctrl+c*.

Checking the logs

Just to be on the safe side, we'll take a quick look at the logs and confirm that **Codecov** script was indeed executed correctly.

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

The relevant parts of the output are as follows, you might need to scroll through your output to find it.

```
...


  _____          _
 / ____|        | |
| |     ___   __| | ___  ___ _____   __
| |    / _ \ / _` |/ _ \/ __/ _ \ \ / /
| |___| (_) | (_| |  __/ (_| (_) \ V /
 \_____\___/ \__,_|\___|\___\___/ \_/
                              Bash-8a28df4
...
==> Python coveragepy not found
==> Searching for coverage reports in:
    + .
    -> Found 1 reports
==> Detecting git/mercurial file structure
==> Reading reports
    + ./coverage.txt bytes=1289
==> Appending adjustments
    http://docs.codecov.io/docs/fixing-reports
    + Found adjustments
==> Gzipping contents
==> Uploading reports
    url: https://codecov.io
    query: branch=master&commit=eb67f2a869f16ce1a02d4903f6eec0af124300dc&build=&build_url=&name=&t
    -> Pinging Codecov
https://codecov.io/upload/v4?package=bash-8a28df4&token=4384f439-9da1-4be3-af60-ee80aed67bc8&branc
    -> Uploading
    -> View reports at https://codecov.io/github/vfarcic/go-demo-6/commit/eb67f2a869f16ce1a02d4903
...
```

> ⚠ You might see an error message `find: unrecognized: -execdir` . That's a

> bug in CodeCov, and you can ignore it, at least during the exercises in this chapter.

As you can see, the coverage report was uploaded to **Codecov** for evaluation, and we got a link where we can see the result. Feel free to visit it. We won't be using it in the exercises since there is a better way to see the results. I'll explain it soon. For now, there is an important issue we need to fix.

Creating a secret

We added the **Codecov** token directly to the pipeline. As you can imagine, that is very insecure. There must be a way to provide the token without storing it in Git. Fortunately, Jenkins X pipelines have a solution. We can define an environment variable that will get the value from a Kubernetes secret. So, our next step is to create the secret.

```
kubectl create secret \
    generic codecov \
    --from-literal=token=$CODECOV_TOKEN
```

Updating the pipeline

Now we can update the pipeline. Please execute the command that follows.

```
echo "buildPack: go
pipelineConfig:
  env:
  # This was modified
  - 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
" | tee jenkins-x.yml
```

This time, instead of creating an `env` with a `value`, we used `valuefrom` with reference to the Kubernetes secret `codecov` and the key `token`.

Validating changes

Let's see whether or not our updated pipeline works correctly.

```
jx step syntax validate pipeline

git add .

git commit -m "Code coverage secret"

git push

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

We validated the pipeline syntax, pushed the change to GitHub, and started watching the activities related to the pull request. The output should be the same as before since we did not change any of the steps. What matters is that all the steps of the newly executed activity should be successful.

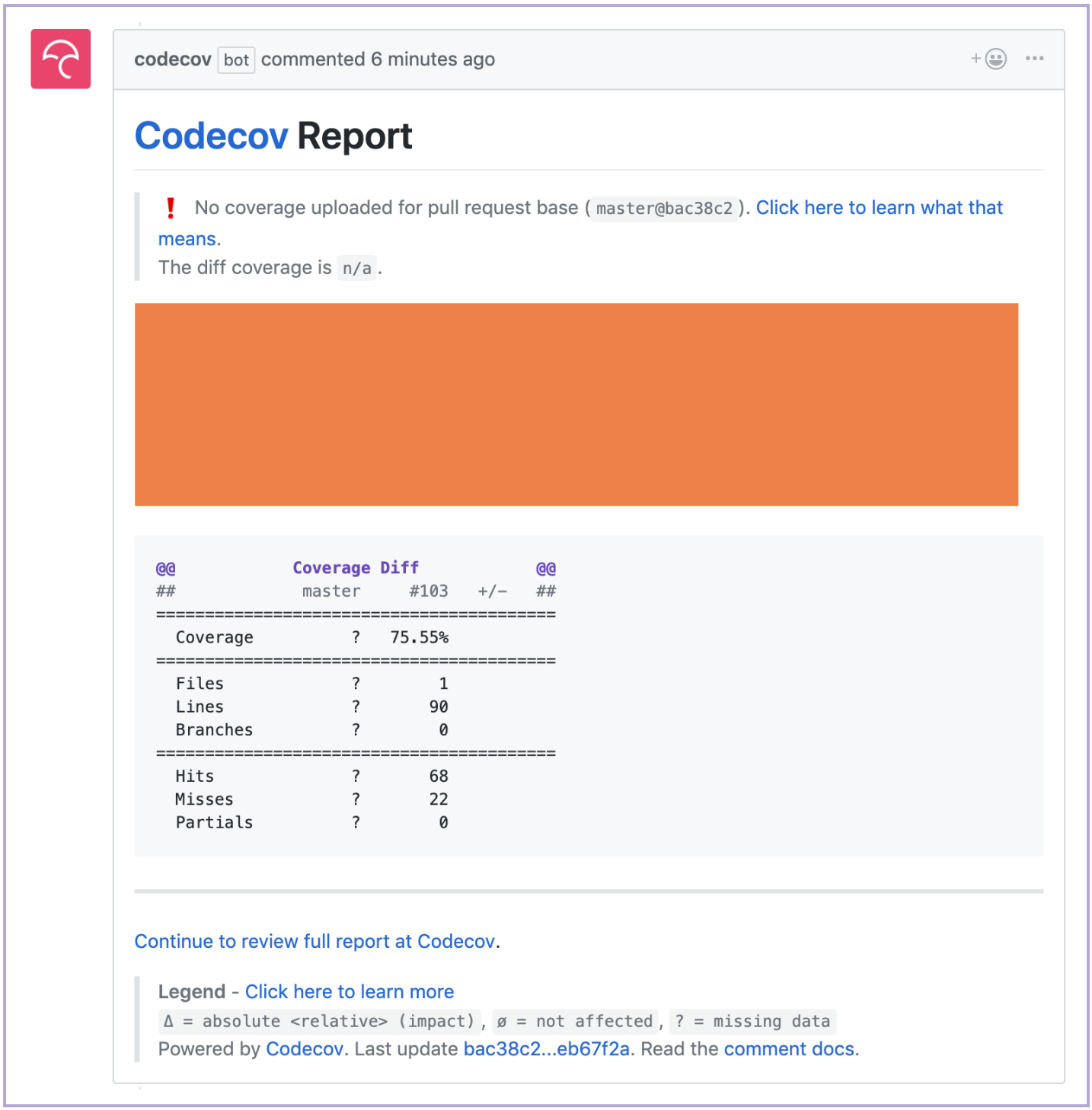
Please stop watching the activity by pressing *ctrl+c*.

Inspecting Codecov entry in the PR

Now that we are securely calculating code coverage, we can take a look at the pull request. If the integration was successful, we should see **Codecov** entries.

Please click on the `Preview` link from the last activity to open the pull request in your favorite browser.

You should see a comment in the pull request similar to the screenshot that follows.



Codecov report in GitHub

Don't be alarmed if you see a warning; **Codecov** could not compare pull request coverage with the one from the master branch because we haven't merged anything to master since we started using **Codecov**. That'll be fixed by itself when we merge with PR.

Additionally, you should see **Codecov** activity in the "checks" section of the pull request.

Since delaying is not a good practice, let's merge the pull request right away. That

way we'll give **Codecov** something to compare future pull request coverage

against. The next challenge will require that we work with the master branch anyway.

Please click *Merge pull request* followed by the *Confirm merge* button. Click the *Delete branch* button.

All that's left, before we move on, is to check out the master branch locally, to pull the latest version of the code from GitHub, and to delete the local copy of the `better-pipeline` branch.

```
git checkout master

git pull

git branch -d better-pipeline
```

Off we go to the next challenge.