Controlling Release Versioning from Jenkins X Pipelines

This lesson shows how to control the release versioning from the Jenkins X pipelines.

We'll cover the following

- Adding the VERSION variable in the Makefile
- Pushing changes to GitHub
- Getting all applications in the staging environment
- Confirming the version of the release on GitHub
- Testing patch versioning

Adding the `VERSION` variable in the `Makefile`

One way to take better control of versioning is to add the variable `VERSION` to the project's `Makefile`.

Please open it in your favorite editor and add the snippet that follows as the first line.

```
VERSION := 1.0.0
```

> ⚠ Just as before, do not use `1.0.0` blindly. Make sure that the major version is higher than the current version.

If we execute `jx-release-version`, the output would be `1.0.0` (or whatever value was put into the `Makefile`). We won't do that because we do not have `jx-release-version` on our laptop, and we already deleted the DevPod. For now, you'll need to trust me on that one.

By now, you might be wondering why we are exploring `jx-release-version`. *What is its relation to Jenkins X pipelines?* That binary is used in every pipeline available

in Jenkins X build packs. We'll see that part of definitions later when we explore

Jenkins X pipelines. For now, we'll focus on the effect it produces, rather than how and where its usage is defined.

Pushing changes to GitHub

Let's see what happens when we push the changes to GitHub.

```
git add .

git commit \
    --message "Finally 1.0.0"

git push

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

We pushed the change to `Makefile`, and now we are watching *go-demo-6* activities. Soon, a new activity will start, and the output should be similar to the one that follows.

```
...
vfarcic/go-demo-6/master #2  1m20s  Running Version: 1.0.0
...
```

We can almost immediately see that in my case the `go-demo-6` activity `#2` is using version `1.0.0`. Since I'm paranoid by nature, we'll make a couple of other validations to confirm that versioning indeed works as expected.

Please wait until the new release is promoted to staging and press *ctrl+c* to stop the activity watcher.

Getting all applications in the staging environment

Next, we'll list the applications and confirm that the correct version was deployed to the staging environment.

```
jx get applications --env staging
```

The output is as follows.

```
APPLICATION STAGING PODS URL
```

```
go-demo-6   1.0.0   1/1  http://go-demo-6.jx-staging.52.42.36.56.nip.io
```

We can see that in my case the `go-demo-6` release running in the staging environment is `1.0.0`.

Confirming the version of the release on GitHub

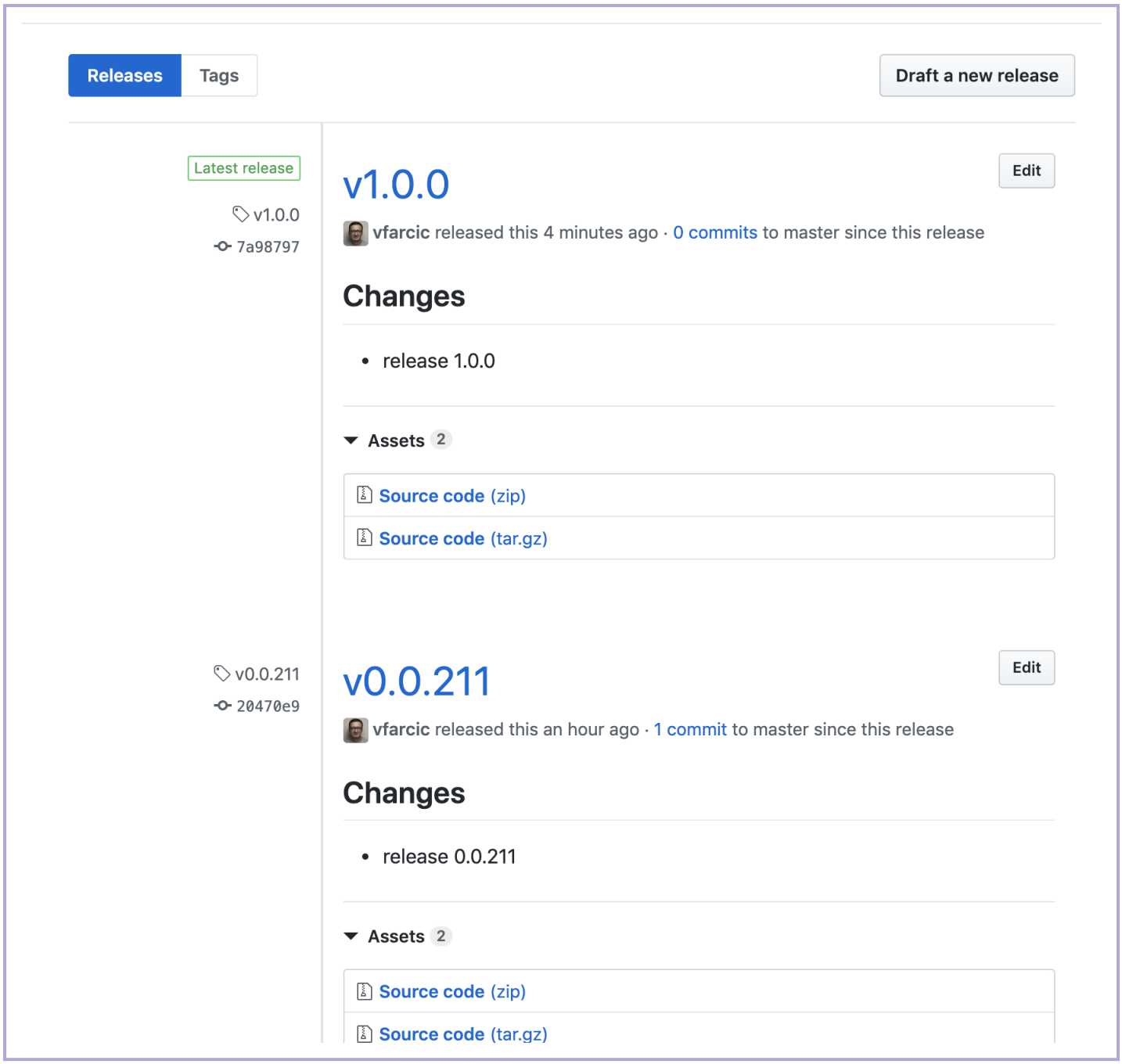
Finally, the last thing we'll do is validate that the release stored in GitHub is also based on the new major version.

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

```
GH_USER=[...]

open "https://github.com/$GH_USER/go-demo-6/releases"
```

We can see that the release in GitHub is also based on the new major version, and we can conclude that everything works as expected.



GitHub releases

Testing patch versioning

Let's make one more change and confirm that only the patch version will increase.

```
echo "A silly change" | tee README.md

git add .

git commit \
    --message "A silly change"

git push

jx get activity \
    --filter go-demo-6 \
```

```
    --watch
```

In my case, the output of the new activity showed that the new release is `1.0.1`. Please stop the activity watcher by pressing *ctrl+c*.

The next change will be `1.0.2`, the one after that `1.0.3`, and so on and so forth until the minor or the major version change again in `Makefile`. It's elegant and straightforward, isn't it?

But what should we do when we don't want to use semantic versioning?

Before we proceed, we'll get out of the `go-demo-6` directory.

```
cd ..
```

Next, let's explore how to customize our versioning logic.