Versioning Releases Through Tags

This lesson explains how to version the releases in Jenkins X using tags.

We'll cover the following

Getting all running applications

Before we start "playing" with versions, we'll take a look at what we're currently running in the staging environment.

```
jx get applications
```

```
APPLICATION STAGING PODS URL
go-demo-6   0.0.211 1/1  http://go-demo-6.jx-staging.52.42.36.56.nip.io
```

In my case, `go-demo-6` version `0.0.211` is running in the staging environment. Your version is likely going to be different, and if you did not destroy the cluster at the end of the previous chapter, you are probably going to see that a release is running in production as well.

> If you forked *go-demo-6* after I wrote this chapter (April 2019), your version is likely going to be `1.x.x` or `2.x.x`. That's OK. It does not matter which version you're running right now.

Before we see how we can control versions through Jenkins X pipelines, we'll take a quick look at the `jx-release-version` CLI. It is already used inside our pipelines,

and understanding how it works will help us get a better grip at how we can combine it with our processes.

There are two ways we can proceed with the examples that involve `jx-release-version`.

1. You can visit the releases and install one that matches your operating system. You are free to do that, but I won't be providing a detailed walkthrough.
2. The alternative is to create a DevPod. This is simpler in the sense that the same commands will work no matter the operating system you're using. The instructions that follow assume that you prefer to use a DevPod and act as a refresher of what we learned in the Improving And Simplifying Software Development chapter.

Creating a DevPod

> ⚠ Please make sure that you are inside the local copy of the *go-demo-6* repository before executing the commands that follow.

```
jx create devpod --label go --batch-mode

jx rsh -d

cd go-demo-6
```

We created a DevPod, entered inside it, and navigated to the `go-demo-6` directory that contains the source code of our application.

Installing `jx-release-version-linux`

Next, we'll install `jx-release-version-linux`.

```
curl -L \
  -o /usr/local/bin/jx-release-version \
  https://github.com/jenkins-x/jx-release-version/releases/download/v1.0.17/jx-release-version-linu

chmod +x \
  /usr/local/bin/jx-release-version
```

We downloaded the `jx-release-version-linux` binary into the `/usr/local/bin/`

We downloaded the `jx-release-version-linux` binary into the `/usr/local/bin/` directory so that it is inside the `PATH`, and we added the executable permissions.

Inspecting git tags

Now, let's take a look at the Git tags we made so far.

```
git tag
```

Depending on when you forked the *go-demo-6* repository and how many tags you created so far, the list of the existing tags might be quite extensive. Please note that you can scroll through the tags using arrow keys. In my case, the output limited to the few latest entries is as follows.

```
...
v0.0.210
v0.0.211
...
```

We can see that the last tag I created is `v0.0.211`. Yours is likely going to be different.

Please press *q* to exit the list of the tags.

Executing `jx-release-version` command

Now that we know the version of the last release, let's see what we'll get if we execute `jx-release-version`.

```
jx-release-version
```

In my case, the output is `0.0.212`. `jx-release-version` examined the existing tags, found the latest one, and incremented the patch version. If we made a change that is not a new functionality and if we did not break compatibility with the previous release, the result is correct since only the patch version was incremented.

Now that we know that `jx-release-version` increments patch versions, we might wonder how to increase minor or major versions. After all, not everything we do consists of fixing bugs.

Let's say that we did make a breaking change, and therefore, we want to increase the major version. Since we already know that `jx-release-version` will find the latest tag, we can accomplish our goal by creating a tag manually.

> ⚠ Please do not run the command that follows as-is. Instead, increment the major version. If the current major is `0`, the new tag should be `v1.0.0`. If the current major is `1`, the new tag should be `v2.0.0`. And so on, and so forth. The command that follows works in my case because my current major version is `0`. However, if you forked the repository after I wrote this chapter (April 2019), you are likely to have the major version `1` or greater.

```
git tag v1.0.0
```

Now that we created a new *dummy* tag, we'll take another look at the output of `jx-release-version`.

```
jx-release-version
```

In my case, the output is `1.0.1`. The result is still an increment of the patch version. Since the last tag was `v1.0.0`, the new release should be `v1.0.1`.

> 📝 Please note that we could accomplish a similar effect by creating a tag that bumps a minor version instead.

Typically, our next step would be to push the newly created tag `v1.0.0` and let the future build of the *go-demo-6* pipeline continue generating new releases based on the new major version. However, we won't push the new tag because there is a better way to increment our major and minor versions.

Deleting the DevPod

We do not need the DevPod anymore, so we'll exit and remove it.

```
exit

jx delete devpod
```

Please make sure to type `y` and press the enter key when you are asked whether you want to delete the DevPod.

There are two potential problems we encountered so far.

1. Creating dummy tags only for the sake of bumping future major and minor releases is not very elegant.
2. We cannot have a bump with patch version `0`. For example, we can have a release `v1.0.1`, but we cannot have `v1.0.0` because that version is reserved for the dummy tag.

We will learn how to fix these potential problems in the next lesson.