The Relation Between Application and Environment Pipelines

This lesson explores the relationship between application and environment pipelines by describing the flow from a commit to deployment.

We'll cover the following

Keep in mind that I am aware we haven't gone into detail on the application pipeline, that's coming soon. Right now, we'll focus on the overall flow between the two.

The flow from a commit to the `master` branch to deployment to the staging environment

Everything starts with a push into the `master` branch of an application repository (e.g., *go-demo-6*). That push might be direct or through a pull request. For now, what matters is that something is pushed to the `master` branch.

A push to any branch initiates a webhook request to Jenkins X. It does not matter much whether the destination is Jenkins itself, **Prow**, or something else. We haven't gone through different ways we can define webhook endpoints. What matters is that the webhook might initiate activity that performs a set of steps defined in `jenkins-x.yml` residing in the repository that launched the process. Such an activity might do nothing if `jenkins-x.yml` ignores that branch, it might execute only a fraction of the steps, or it might run all of them. It all depends on the `branch` filters. In this case, we're concerned with the steps defined to run when a push is done on the `master` branch of an application.

From a very high level, a push from the `master` branch of an application initiates a build that checks out the code, builds binaries (e.g., container image, **Helm** chart), makes a release and pushes it to registries (e.g., container registry, **Helm** charts registry, etc.), and promotes the release. This last step is where GitOps principles are more likely to clash with what you're used to doing. More often than not, a promotion would merely deploy a new release to an environment. We're not doing that because we'd break at least four of the rules we defined.

If the build that was initiated through a Webhook of an application repository results in deployment, that change would not be stored in Git.

- We could not say that **Git is the only source of truth.**
- That deployment would **not be tracked**.
- The operation **would not be reproducible**.
- Not everything **would be idempotent**.
- Finally, we would also break the rule that **information about all the releases must be stored in environment-specific repositories or branches.**

Truth be told, we could fix these issues by pushing a change to the environment-specific repository after the deployment, but that would break the rule that **everything must follow the same coding practices**. Such a course of action would result in an activity that was not initiated by a push of a change to Git, and would not follow whichever coding practices we decided to follow. We have to follow all the rules, but keep in mind that the order matters as well. Simply put, we push a change to Git, and that ends with a change of the system, not the other way around.

Taking all that into account, the only logical course of action is for the promotion steps in the application pipeline to make a push to a branch of the environment-specific repository. Given that we choose to promote to the staging environment automatically, it should also create a pull request, it should approve it, and it should merge it to the `master` branch automatically. That is an excellent example of following a process, even when humans are not involved.

At this point, you might be asking yourself:

> **"What is the change to the environment-specific repository pushed by the application-specific build?"**

If you paid attention to the contents of the `requirements.yaml` file, you should already know the answer. Let's output it one more time as a refresher.

```
cat env/requirements.yaml
```

The relevant parts of the output are as follows.

```
dependencies:
...
- name: go-demo-6
  repository: http://jenkins-x-chartmuseum:8080
  version: 0.0.131
```

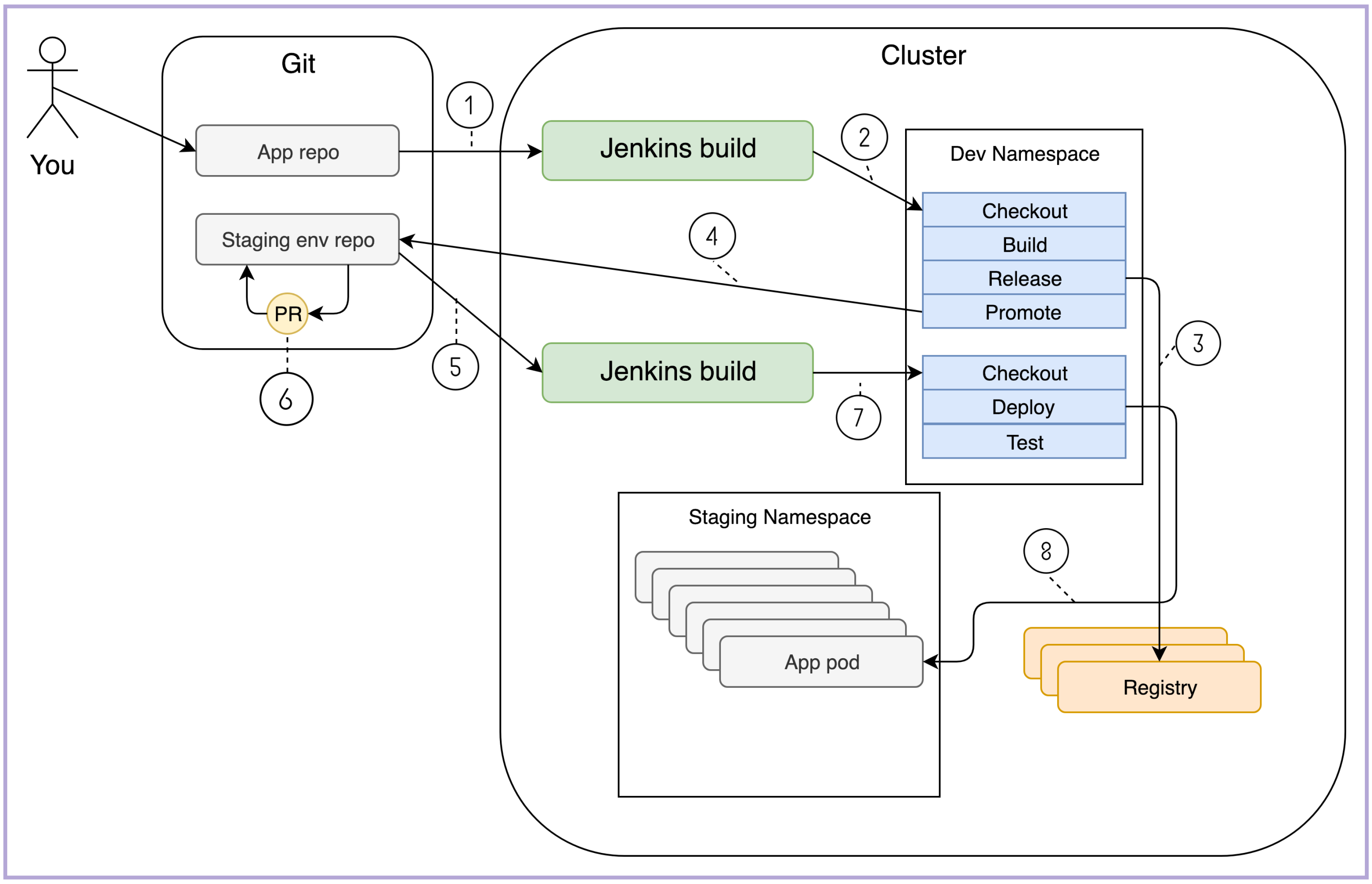
So, a promotion of a release from an application-specific build results in one of two things. If this is the first time we're promoting an application, Jenkins X will add a new entry to `requirements.yaml` inside the environment-specific repository. Otherwise, if a previous release of that application already runs in that environment, it'll update the `version` entry to the new release. As a result, `requirements.yaml` will always contain the complete and accurate definition of the whole environment and each change will be a new commit. That way, we're complying with GitOps principles. We are tracking changes, we have a single point of truth for the whole environment, we are following our coding principles (e.g., pull requests), and so on and so forth. Long story short, we're treating an environment in the same way we're treating an application. The only important difference is that we are not pushing changes to the repository dedicated to the staging environment. Builds of application-specific pipelines are doing that for us, simply because we decided to have automatic promotion to the staging environment.

What happens when we push something to `master` branch of a repository?

Git sends a webhook request which initiates yet another build. Actually, even the pull request initiates a build, so the whole process of automatically promoting a release to the staging environment results in two new builds; one for the PR, and the other after merging it to the `master` branch.

So, a pull request to the repository of the staging environment initiates a build that results in automatic approval and a merge of the pull request to the `master` branch. That launches another build that deploys the release to the staging environment.

With that process, we are fulfilling quite a few of the rules (commandments), and we are a step closer to have “real” GitOps continuous delivery processes that are, so far, fully automated. The only human interaction is a push to the application repository. That will change later on when we reach deployments to the production environment but, so far, we can say that we are fully automated.



The flow from a commit to the master branch to deployment to the staging environment

Please note that the previous diagram is not very accurate. We did not yet explore all the technologies behind the process, so it is intentionally vague.

Tests in the application-specific pipeline

One thing that we are obviously missing is tests in the application-specific pipeline. We’ll correct that in one of the next chapters. But, before we reach the point that we can promote to production, we should apply a similar set of changes to the repository of the production environment.

I’ll leave it to you to add tests there as well. The steps should be the same, and you should be able to reuse the same file with integration tests located here. You can skip doing that since production tests are not mandatory for the rest of the

exercises we'll do. If you choose not to add them, please use your imagination so that whenever we talk about an environment, you always assume that we can have tests, if we choose to.

Actually, we might even argue that we do not need tests in the production environment. If that statement confuses you or if you do not believe that's true, you'll have to wait for a few more chapters when we explore promotions to production.

We haven't explored what happens when we have multiple applications yet. I believe there's no need for exercises that will prove that all the apps are automatically deployed to the staging environment. The process is the same no matter if we have only one, or we have tens or hundreds of applications. The `requirements.yaml` file will contain an entry to each application running in the environment. No more, no less. We don't necessarily have to deploy all the applications to the same environment. That can vary from case to case, and it often depends on our Jenkins X team structure that we'll explore later.

Next, let's learn how to manipulate the environments.