Verifying Jenkins X Boot Installation

In this lesson, we will verify the installation done by Jenkins X Boot by checking pipelines, Namespaces, activities and creating a new quickstart project.

We'll cover the following

- Checking the pipelines
- Triggering the dev pipeline
- Checking the Namespaces present in the cluster
- Checking the environments
- Creating a new quickstart project
- Inspecting the activities and Namespaces

Checking the pipelines

Let's take a quick look at the pipelines currently active in our cluster.

```
jx get pipelines --output yaml
```

The output is as follows.

```
- vfarcic/environment-jx-boot-dev/master
- vfarcic/environment-jx-boot-production/master
- vfarcic/environment-jx-boot-staging/master
```

We are already used to working with `production` and `staging` pipelines. What is new is the `dev` pipeline. That is the one we just executed locally. It is now available in the cluster as well, and we should be able to trigger it by pushing a change to the associated repository. Let's test that.

Triggering the `dev` pipeline

We'll explore the Jenkins X upgrade process later. For now, we just want to see whether the `dev` repository is indeed triggering pipeline activities. We'll do that by making a trivial change to the `README.md` file.

```
echo "A trivial change" \
    | tee -a README.md

git add .

git commit -m "A trivial change"

git push

jx get activities \
    --filter environment-$CLUSTER_NAME-dev \
    --watch
```

We pushed the changes to GitHub and started watching the activities of the `dev` pipeline. The output, when the activity is finished, should be as follows.

```
STEP                                      STARTED AGO DURATION STATUS
vfarcic/environment-jx-boot-dev/master #1       4m10s    3m52s Succeeded
  release                                       4m10s    3m52s Succeeded
    Credential Initializer 7jh9t                4m10s       0s Succeeded
    Working Dir Initializer Tr2wz               4m10s       2s Succeeded
    Place Tools                                  4m8s       2s Succeeded
    Git Source Vfarcic Environment Jx...         4m6s      36s Succeeded https://github.com/vfarci
    Git Merge                                   3m30s       1s Succeeded
    Validate Git                                3m29s       1s Succeeded
    Verify Preinstall                           3m28s      26s Succeeded
    Install Jx Crds                              3m2s      10s Succeeded
    Install Velero                              2m52s      12s Succeeded
    Install Velero Backups                      2m40s       2s Succeeded
    Install Nginx Controller                    2m38s      16s Succeeded
    Create Install Values                       2m22s       0s Succeeded
    Install External Dns                        2m22s      16s Succeeded
    Install Cert Manager Crds                    2m6s       0s Succeeded
    Install Cert Manager                         2m6s      16s Succeeded
    Install Acme Issuer And Certificate         1m50s       2s Succeeded
    Install Vault                               1m48s       8s Succeeded
    Create Helm Values                          1m40s       3s Succeeded
    Install Jenkins X                           1m37s     1m3s Succeeded
    Verify Jenkins X Environment                  34s       5s Succeeded
    Install Repositories                          29s       5s Succeeded
    Install Pipelines                             24s       1s Succeeded
    Update Webhooks                               23s       4s Succeeded
    Verify Installation                           19s       1s Succeeded
```

That's the first activity (`#1`) of the `dev` pipeline. To be more precise, it is the second one (the first was executed locally) but, from the perspective of Jenkins X inside the cluster, which did not exist at the time, that is the first activity. Those are the steps of Jenkins X Boot running inside our cluster.

We won't go through the changes that were created by that activity since there are none. We did not modify any of the files that matter. We already used those same files when we executed Jenkins X Boot locally. All we did was push the change of the `README` file from the local repository to GitHub. That triggered a webhook that the README file from the local repository to GitHub. That triggered a webhook that

notified the cluster that there are some changes to the remote repo. As a result, the first in-cluster activity was executed.

The reason I showed you that activity was not due to an expectation of seeing some change applied to the cluster (there were none), but to demonstrate that, from now on, we should let Jenkins X running inside our Kubernetes cluster handle changes to the `dev` repository, instead of running `jx boot` locally. That will come in handy later on when we explore how to upgrade or change our Jenkins X setup.

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

Checking the Namespaces present in the cluster

Let's take a look at the Namespaces we have in our cluster.

```
kubectl get namespaces
```

The output is as follows.

```
NAME         STATUS AGE
cert-manager Active 28m
default      Active 74m
jx           Active 34m
kube-public  Active 74m
kube-system  Active 74m
velero       Active 33m
```

As you can see, there are no Namespaces for staging and production environments. **Does that mean that we do not get them with Jenkins X Boot?**

Unlike other types of setup, the Boot creates environments lazily. That means that they are not created in advance, but rather when used for the first time. In other words, the `jx-staging` Namespace will be created the first time we deploy something to the staging environment. The same logic is applied to any other environment, the production included.

Checking the environments

To put your mind at ease, we can output the environments and confirm that staging and production were indeed created.

```
jx get env
```

The output is as follows.

```
NAME       LABEL       KIND        PROMOTE NAMESPACE     ORDER CLUSTER SOURCE
dev        Development Development Never   jx            0             https://github.com/vfarcic/
staging    Staging     Permanent   Auto    jx-staging    100           https://github.com/vfarcic/
production Production  Permanent   Manual  jx-production 200           https://github.com/vfarcic/
```

`Staging` and `production` environments do indeed exist, and they are associated with corresponding Git repositories, even though Kubernetes Namespaces were not yet created.

While we're on the subject of environments, we can see something that did not exist in the previous setups. What's new, in this output, is that the `dev` environment also has a repo set as the source. That was not the case when we were creating clusters with `jx create cluster` or installing Jenkins X with `jx install`. In the past, the `dev` environment was not managed by GitOps principles. Now it is.

The associated `source` is the repository we forked, and it contains the specification of the whole `dev` environment and a few other things. From now on, if we need to modify Jenkins X or any other component of the platform, all we have to do is push a change to the associated repository. A pipeline will take care of converging the actual with the desired state. From this moment onward, everything except infrastructure is managed through GitOps. I intentionally said *"except infrastructure"* because I did not show you explicitly how to create a pipeline that will execute **Terraform** or whichever tool you're using to manage your infra. However, that is not an excuse for you not to do it.

By now, you should have enough knowledge to automate that last piece of the puzzle by creating a Git Repository with `jenkins-x.yml` and importing it into Jenkins X. If you do that, everything will be using GitOps, and everything will be automated, except writing code and pushing it to Git. Do not take my unwillingness to force a specific tool (e.g., **Terraform**) as a sign that you shouldn't walk that last mile.

Creating a new quickstart project

To be on the safe side, we'll create a new quickstart project and confirm that the system is behaving correctly. We still need to verify that lazy creation of environments works as expected.

```
cd ..

jx create quickstart \
    --filter golang-http
```

We went back from the local copy of the `dev` repository and started the process of creating a new `quickstart`.

You might be wondering why we didn't use the `--batch-mode` as we did countless times before. This time, we installed Jenkins X using a different method, and the local cache does not have the information it needs to create a `quickstart` automatically. So, we'll need to answer a series of questions, just as we did at the beginning of the book. That is only a temporary issue. The moment we create the first quickstart manually, the information will be stored locally, and every consecutive attempt to create new quickstarts can be done with the `--batch-mode`, as long as we're using the same laptop.

Please answer the questions any way you like. Just bear in mind that I expect you to name the repository `jx-boot`. If you call it differently, you'll have to modify the commands that follow.

Inspecting the activities and Namespaces

All that's left now is to wait until the activity of the pipeline created by the quickstart process is finished.

```
jx get activity \
    --filter jx-boot/master \
    --watch
```

Please press *ctrl+c* to stop watching the activity once it's finished.

We'll also confirm that the activity of the staging environment is finished as well.

```
jx get activity \
    --filter environment-$CLUSTER_NAME-staging/master \
    --watch
```

You know what to do. Wait until the new activity is finished and press *ctrl+c* to stop watching.

Let’s retrieve the Namespaces again.

```
kubectl get namespaces
```

The output is as follows.

```
NAME         STATUS AGE
cert-manager Active 34m
default      Active 80m
jx           Active 40m
jx-staging   Active 55s
kube-public  Active 80m
kube-system  Active 80m
velero       Active 39m
```

As you can see, the `jx-staging` Namespace was created the first time an application was deployed to the associated environment, not when we installed the platform. The `jx-production` Namespace is still missing since we did not yet promote anything to production. Once we do, that Namespace will be created as well.

I won’t bother you with the commands that would confirm that the application is accessible from outside the cluster, that we can create preview (pull request) environments, and that we can promote releases to production. I’m sure that, by now, you know how to do all that by yourself.

The next lesson will wrap up this chapter and give instructions to free up the used resources.