

Installing Jenkins X Using GitOps Principles

In this lesson, we will see how to install Jenkins X in accordance with GitOps principles and using the Jenkins X Boot.

We'll cover the following



- Taking a look at the GitHub repository
- Defining variables
- Cloning the repository
- Exploring and modifying the jx-requirements.yml file
 - The cluster section
 - The gitops value
 - The environments section
 - The ingress section
 - The kaniko value
 - The secretStorage value
 - The storage section
 - The versionsStream section
 - The webhook value
- Alternatives for Prow
- Adding additional information for EKS
- Did we miss any values?
- Installing Jenkins X with jx boot
 - Usernames of approvers
 - TLS warnings
 - Upgrade Jenkins X version
 - Long-term storage
 - Secrets and CRDs
 - Nginx Ingress and domain
 - Certificates
 - Vault

- vault
- GitHub configurations
- External Docker Registry

How can we install Jenkins X in a better way than what we're used to? Jenkins X configuration should be defined as code and reside in a Git repository, and that's what the community created for us. It maintains a GitHub repository that contains the structure of the definition of the Jenkins X platform, together with a pipeline that will install it, as well as a requirements file that we can use to tweak it to our specific needs.

Taking a look at the GitHub repository

Let's take a look at the repository.

```
open "https://github.com/jenkins-x/jenkins-x-boot-config.git"
```



We'll explore the files in it a bit later. Or, to be more precise, we'll explore those that you are supposed to customize.

Defining variables

Next, we'll define a variable `CLUSTER_NAME` that will, as you can guess, hold the name of the cluster we created a short while ago.



In the commands that follow, please replace the first occurrence of [...] with the name of the cluster and the second with your GitHub user.

```
export CLUSTER_NAME=[...]
```

```
export GH_USER=[...]
```



Cloning the repository

Now that we forked the Boot repo and we know what our cluster is called, we can clone the repository with a proper name that will reflect the naming scheme of our soon-to-be-installed Jenkins X.

```
git clone \
https://github.com/jenkins-x/jenkins-x-boot-config.git \
environment-$CLUSTER_NAME-dev
```

Exploring and modifying the `jx-requirements.yml` file

The key file that contains (almost) all the parameters that can be used to customize the setup is `jx-requirements.yml`. Let's take a look at it.

```
cd environment-$CLUSTER_NAME-dev

cat jx-requirements.yml
```

The output is as follows.

```
cluster:
  clusterName: ""
  environmentGitOwner: ""
  project: ""
  provider: gke
  zone: ""
gitops: true
environments:
- key: dev
- key: staging
- key: production
ingress:
  domain: ""
  externalDNS: false
  tls:
    email: ""
    enabled: false
    production: false
kaniko: true
secretStorage: local
storage:
  logs:
    enabled: false
    url: ""
  reports:
    enabled: false
    url: ""
  repository:
    enabled: false
    url: ""
versionStream:
  ref: "master"
  url: https://github.com/jenkins-x/jenkins-x-versions.git
webhook: prow
```

As you can see, that file contains values in a format that resembles `requirements.yml` file used with **Helm** charts. It is split into a few sections.



The format of the `jx-requirements.yml` file might have changed since I wrote this section, so your output might be different. Nevertheless, what I'm describing should give you a good enough grip over the values you can tweak, and you should be able to extend that knowledge to those not represented here.

The `cluster` section `#`

First, there is a group of values that define our `cluster`. You should be able to figure out what it represents by looking at the variables inside it. It probably won't take you more than a few moments to see that we have to change at least some of those values, so that's what we'll do next.

Please open `jx-requirements.yml` in your favorite editor and change the following values.

- Set `cluster.clusterName` to the name of your cluster. It should be the same as the name of the environment variable `CLUSTER_NAME`. If you already forgot it, execute `echo $CLUSTER_NAME`.
- Set `cluster.environmentGitOwner` to your GitHub user. It should be the same as the one we previously declared as the environment variable `$GH_USER`.
- Set `cluster.project` to the name of your GKE project, only if that's where your Kubernetes cluster is running. Otherwise, leave that value intact (empty). If you used one of my Gists to create a GKE cluster, the name of the project should be in the environment variable `PROJECT`, so feel free to output it with `echo $PROJECT` if you are forgetful.
- Set `cluster.provider` to `gke` or to `eks` or to any other provider if you decided that you are brave and want to try currently unsupported platforms. Or, the things might have changed since I wrote this chapter, and your provider is indeed supported now.
- Set `cluster.zone` to whichever zone your cluster is running in. If you're running a regional cluster (as you should) then the value should be the region, not the zone. If, for example, you used my Gist to create a GKE cluster, the value should be `us-east1`. Similarly, the one for EKS is `us-east-1`.

We're finished with the `cluster` section, and the next in line is the `gitops` value.

The `gitops` value

It instructs the system how to treat the Boot process. I don't believe it makes sense to change it to `false`, so we'll leave it as-is (`true`).

The `environments` section

The next section contains the list of the `environments` that we're already familiar with. The keys are the suffixes, and the final names will be a combination of `environment-` with the name of the cluster followed by the `key`. We'll leave them intact.

The `ingress` section

The `ingress` section defines the parameters related to external access to the cluster (`domain`, TLS, etc.). We won't dive into it just yet. That will be left for later (probably the next chapter).

The `kaniko` value

The `kaniko` value should be self-explanatory. When set to `true`, the system will build container images using **Kaniko** instead of, let's say, Docker. That is a much better choice since Docker cannot run in a container and, as such, poses a significant security risk (mounted sockets are evil), and it messes with Kubernetes scheduler given that it bypasses its API. In any case, **Kaniko** is the only supported way to build container images when using Tekton, so we'll leave it as-is (`true`).

The `secretStorage` value

Next, we have `secretStorage` currently set to `local`. The whole platform will be defined in this repository, except for secrets (e.g., passwords). Pushing them to Git would be childish, so Jenkins X can store the secrets in different locations. If we'd change it to `local`, that location is your laptop. While that is better than a Git repository, you can probably imagine why that is not the right solution. Keeping them locally complicates cooperation (they exist only on your laptop), is volatile, and is only slightly more secure than Git. A much better place for secrets is **HashiCorp Vault**. It is the most commonly used solution for secrets management in Kubernetes (and beyond), and Jenkins X supports it out of the box.

All in all, secrets storage is an easy choice.

- Set the value of `secretStorage` to `vault`.

The `storage` section

Below the `secretStorage` value is the whole section that defines `storage` for `logs`, `reports`, and `repository`. If enabled, those artifacts will be stored on a network drive. As you already know, containers and nodes are short-lived, and if we want to preserve any of those, we need to store them elsewhere. That does not necessarily mean that network drives are the best place, but rather that's what comes out of the box. Later on, you might choose to change that and, let's say, ship logs to a central database like **ElasticSearch**, **PaperTrail**, **CloudWatch**, **StackDriver**, etc.

For now, we'll keep it simple and enable network storage for all three types of artifacts.

- Set the value of `storage.logs.enabled` to `true`
- Set the value of `storage.reports.enabled` to `true`
- Set the value of `storage.repository.enabled` to `true`

For now, we'll keep it simple and keep the default values (`true`) that enable network storage for all three types of artifacts.

The `versionsStream` section

The `versionsStream` section defines the repository that contains versions of all the packages (charts) used by Jenkins X. You might choose to fork that repository and control versions yourself. Before you jump into doing just that, please note that Jenkins X versioning is quite complex, given that many packages are involved. Leave it be unless you have a very good reason to take over the control, and that you're ready to maintain it.

The `webhook` value

Finally, at the time of this writing (October 2019), `webhook` is set to `prowl`. It defines the end-point that receives webhooks and forwards them to the rest of the system, or back to Git.

Alternatives for Prowl

As you already know, **Prowl** supports only GitHub. If that's not your Git provider,

Prow is a no-go. As an alternative, we could set it to `jenkins`, but that's not the right solution either. Jenkins (without X) is not going to be supported for long, given that the future is in Tekton. It was used in the first generation of Jenkins X only because it was a good starting point and because it supports almost anything we can imagine. But, the community embraced Tekton as the only pipeline engine, and that means that static Jenkins X is fading away and that it is used mostly as a transition solution for those accustomed to the “traditional” Jenkins.

So, what can we do if `prow` is not a choice if you do not use GitHub, and `jenkins` days are numbered? To make things more complicated, even **Prow** will be deprecated sometime in the future (or past depending when you read this). It will be replaced with *Lighthouse*, which, at least at the beginning, will provide similar functionality as **Prow**. Its primary advantage when compared with **Prow** is that Lighthouse will (or already does) support all major Git providers (e.g., **GitHub**, **GitHub Enterprise**, **Bitbucket Server**, **Bitbucket Cloud**, **GitLab**, etc.). At some moment, the default value of `webhook` will be `lighthouse`. But, at the time of this writing (October 2019), that's not the case since `Lighthouse` is not yet stable and production-ready. It will be soon. Or, maybe it already is, and I did not yet rewrite this chapter to reflect that.

In any case, we'll keep `prow` as our `webhook` (for now).

Adding additional information for EKS

⚠ Please execute the commands that follow only if you are using **EKS**. They will add additional information related to Vault, namely the IAM user that has sufficient permissions to interact with it. Make sure to replace `[...]` with your IAM user that has sufficient permissions (being admin always works).

```
export IAM_USER=[...] # e.g., jx-boot

echo "vault:
  aws:
    autoCreate: true
    iamUserName: \"\$IAM_USER\" \" \"
    | tee -a jx-requirements.yml
```

⚠ Please execute the command that follows only if you are using **EKS**. The `jx-requirements.yml` file contains `zone` entry and for AWS we need a

`jx-requirements.yml` file contains `zone` entry and for AWS we need a `region`. That command will replace one with the other.

```
cat jx-requirements.yml \  
| sed -e \  
's@zone@region@g' \  
| tee jx-requirements.yml
```

Let's take a peek at how `jx-requirements.yml` looks like now.

```
cat jx-requirements.yml
```

In my case, the output is as follows (yours is likely going to be different).

```
cluster:  
  clusterName: "jx-boot"  
  environmentGitOwner: "vfarcic"  
  project: "devops-26"  
  provider: gke  
  zone: "us-east1"  
gitops: true  
environments:  
- key: dev  
- key: staging  
- key: production  
ingress:  
  domain: ""  
  externalDNS: false  
  tls:  
    email: ""  
    enabled: false  
    production: false  
kaniko: true  
secretStorage: vault  
storage:  
  logs:  
    enabled: true  
    url: ""  
  reports:  
    enabled: true  
    url: ""  
  repository:  
    enabled: true  
    url: ""  
versionStream:  
  ref: "master"  
  url: https://github.com/jenkins-x/jenkins-x-versions.git  
webhook: prow
```



Feel free to modify some values further or to add those that we skipped.

If you used my Gist to create a cluster, the current setup will work. On the other hand, if you created a cluster on your own, you will likely need to


change some values.

Did we miss any values?

Now, you might be worried that we missed some of the values. For example, we did not specify a domain. Does that mean that our cluster will not be accessible from outside? We also did not specify `url` for storage. Will Jenkins X ignore it in that case?

The truth is that we specified only the things we know. For example, if you created a cluster using my Gist, there is no **Ingress**, so there is no external load balancer that it was supposed to create. As a result, we do not yet know the IP through which we can access the cluster, and we cannot generate a `.nip.io` domain. Similarly, we did not create storage. If we did, we could have entered addresses into `url` fields.

Those are only a few examples of the unknowns. We specified what we know, and we'll let Jenkins X Boot figure out the unknowns. Or, to be more precise, we'll let the Boot create the resources that are missing and thus convert the unknowns into knowns.

 In some cases, Jenkins X Boot might get confused with the cache from the previous Jenkins X installations. To be on the safe side, delete the `.jx` directory by executing `rm -rf ~/.jx`.

Installing Jenkins X with `jx boot`

Off we go. Let's install Jenkins X.

```
jx boot
```



Now we need to answer quite a few questions. In the past, we tried to avoid answering questions by specifying all answers as arguments to commands we were executing. That way, we had a documented method for doing things that do not end up in a Git repository. Someone else could reproduce what we did by running the same commands. This time, however, there is no need to avoid questions since everything we'll do will be stored in a Git repository. Later on, we'll see where exactly will Jenkins X Boot store the answers. For now, we'll do our best

to provide the information it needs.



The Boot process might change by the time you read this. If that happens, do your best to answer by yourself the additional questions that are not covered here.

Username of approvers

The first input is asking for a `comma-separated git provider usernames of approvers for development environment repository`. That will create the list of users who can approve pull requests to the development repository managed by Jenkins X Boot. For now, type your GitHub user and hit the enter key.

TLS warnings

We can see that, after a while, we were presented with two warnings stating that TLS is not enabled for `Vault` and `webhooks`. If we specified a “real” domain, Boot would install Let’s Encrypt and generate certificates. But, since I couldn’t be sure that you have a domain at hand, we did not specify it, and, as a result, we will not get certificates. While that would be unacceptable in production, it is quite OK as an exercise.

As a result of those warnings, the Boot is asking us whether we `wish to continue`. Type `y` and press the enter key to continue.

Upgrade Jenkins X version

Given that Jenkins X creates multiple releases a day, the chances are that you do not have the latest version of `jx`. If that’s the case, the Boot will ask, `would you like to upgrade to the jx version?`. Press the enter key to use the default answer `Y`. As a result, the Boot will upgrade the CLI, but that will abort the pipeline. That’s OK. No harm’s done. All we have to do is repeat the process but, this time, with the latest version of `jx`.

```
jx boot
```



The process started again. We’ll skip commenting on the first few questions to `jx boot the cluster` and to `continue` without TLS. Answers are the same as before (`y` in both cases).

Long-term storage

The next set of questions is related to `long term storage` for logs, reports, and repository. Press the enter key to all three questions, and the Boot will create buckets with auto-generated unique names.

Secrets and CRDs

From now on, the process will create the secrets and install CRDs (Custom Resource Definitions) that provide custom resources specific to Jenkins X.

Nginx Ingress and domain

Then, it'll install **Nginx Ingress** (unless your cluster already has one) and set the domain to `.nip.io` since we did not specify one.

Certificates

Further on, it will install **CertManager**, which will provide **Let's Encrypt** certificates. Or, to be more precise, it would provide the certificates if we specified a domain. Nevertheless, it's installed just in case we change our minds and choose to update the platform by changing the domain and enabling TLS later on.

Vault

The next in line is Vault. The Boot will install it and attempt to populate it with the secrets. But, since it does not know them just yet, the process will ask us another round of questions. The first one in this group is the `Admin Username`. Feel free to press the enter key to accept the default value `admin`. After that comes `Admin Password`. Type whatever you'd like to use (we won't need it today).

GitHub configurations

The process will need to know how to access our GitHub repositories, so it asks us for the Git `username`, `email address`, and `token`. I'm sure that you know the answers to the first two questions. As for the token, if you did not save the one we created before, you'll need to create a new one. Do that, if you must, and feed it to the Boot. Finally, the last question related to secrets is `HMAC token`. Feel free to press the enter key, and the process will create it for you.

External Docker Registry

Finally comes the last question `Do you want to configure an external Docker`

Finally comes the last question. `Do you want to configure an external Docker Registry?` Press the enter key to use the default answer (`N`) and the Boot will

create it inside the cluster or, as in case of most cloud providers, use the registry provided as a service. In case of GKE, that would be GCR, for EKS that's ECR, and if you're using AKS, that would be ACR. In any case, by not configuring an external Docker Registry, the Boot will use whatever makes the most sense for a given provider.

The rest of the process will install and configure all the components of the platform. We won't go into all of them since they are the same as those we used before. What matters is that the system will be fully operational a while later.

The last step will verify the installation. You might see a few warnings during this last step of the process. Don't be alarmed. The Boot is most likely impatient. Over time, you'll see the number of `running` Pods increasing and those that are `pending` decreasing, until all the Pods are `running`.

That's it. Jenkins X is now up-and-running. On the surface, the end result is the same as if we used the `jx install` command but, this time, we have the whole definition of the platform with complete configuration (except for secrets) stored in a Git repository. That's a massive improvement by itself. Later on, we'll see additional benefits like upgrades performed by changing any of the configuration files and pushing those changes to the repository. But, for now, what matters is that Jenkins X is up-and-running. Or, at least, that's what we're hoping for.

In the next lesson, let's explore the changes done by boot.