

---

# THE DEVOPS 2.6 TOOLKIT



## Jenkins X

---

Viktor Farcic

CLOUD-NATIVE  
KUBERNETES-FIRST  
CONTINUOUS DELIVERY

# **The DevOps 2.6 Toolkit: Jenkins X**

## **Cloud-Native Kubernetes-First Continuous Delivery**

**Viktor Farcic**

This book is for sale at <http://leanpub.com/the-devops-2-6-toolkit>

This version was published on 2019-12-01



\* \* \* \* \*

This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

\* \* \* \* \*

© 2019 Viktor Farcic

# Table of Contents

[Preface](#)

[About the Author](#)

[Dedication](#)

[What Is Jenkins X?](#)

[Installing Prerequisites](#)

[Installing Jenkins X CLI](#)

[To Create A Cluster Or Not To Create A Cluster](#)

[Creating A Google Kubernetes Engine \(GKE\) Cluster With jx](#)

[Creating An Amazon Elastic Container Service for Kubernetes \(EKS\) Cluster With jx](#)

[Creating An Azure Kubernetes Service \(AKS\) Cluster With jx](#)

[Is Your Cluster Ready For Jenkins X?](#)

[Installing Jenkins X In An Existing Kubernetes Cluster](#)

[What Did We Get?](#)

[What Now?](#)

[Deleting GKE Cluster And Unused Resources](#)

[Deleting EKS Cluster And Related Resources](#)

[Deleting AKS Cluster And Related Resources](#)

[Uninstalling Jenkins X](#)

[Exploring Quickstart Projects](#)

[Creating A Kubernetes Cluster With Jenkins X](#)

[Creating A Quickstart Project](#)

[Exploring Quickstart Project Files](#)

[Retrieving Jenkins X Activities, Logs, Pipelines, Applications, And Environments](#)

[What Now?](#)

[Importing Existing Projects Into Jenkins X](#)

[Creating A Kubernetes Cluster With Jenkins X](#)

[Importing A Project](#)

[Fixing The Auto-Generated Helm Chart](#)

[Why Did We Do All That?](#)

[What Now?](#)

## Creating Custom Build Packs

Creating A Kubernetes Cluster With Jenkins X

Choosing What To Include In A Build Pack

Creating A Build Pack For Go Applications With MongoDB

Datastore

Testing The New Build Pack

Giving Back To The Community

What Now?

## Applying GitOps Principles

Ten Commandments Of GitOps Applied To Continuous Delivery

Creating A Kubernetes Cluster With Jenkins X And Importing The Application

Exploring Jenkins X Environments

Which Types Of Tests Should We Execute When Deploying To The Staging Environment?

Exploring And Adapting The Staging Environment

Understanding The Relation Between Application And Environment Pipelines

Controlling The Environments

Are We Already Following All The Commandments?

What Now?

## Improving And Simplifying Software Development

Exploring The Requirements Of Efficient Development Environment

Creating A Kubernetes Cluster With Jenkins X And Importing The Application

Creating a Remote Development Environment

Working With The Code In The DevPod Using Browser-Based IDE

Synchronizing Code From A Laptop Into A DevPod

Integrating IDEs With Jenkins X

What Now?

## Working With Pull Requests And Preview Environments

Creating A Kubernetes Cluster With Jenkins X And Importing The Application

Creating Pull Requests

Intermezzo

Merging a PR

## Exploring Jenkins X Garbage Collection

### What Now?

## Promoting Releases To Production

- Creating A Kubernetes Cluster With Jenkins X And Importing The Application
- Promoting A Release To The Production Environment
- What Now?

## Versioning Releases

- Semantic Versioning Explained
- Creating A Kubernetes Cluster With Jenkins X And Importing The Application
- Versioning Releases Through Tags
- Controlling Release Versioning From Jenkins X Pipelines
- Customizing Versioning Logic
- Versioning With Maven, NodeJS, And Other Build Tools
- What Now?

## Going Serverless

- Exploring Prow, Jenkins X Pipeline Operator, And Tekton

## Implementing ChatOps

- Creating A Kubernetes Cluster With Jenkins X
- Exploring The Basic Pull Request Process Through ChatOps
- Exploring Additional Slash Commands
- How Do We Know Which Slash Commands Are Available?
- What Now?

## Using The Pipeline Extension Model

- The Evolution Of Jenkins Jobs And How We Got To The YAML-Based jenkins-x.yml Format
- Getting Rid Of Repetition
- Creating A Kubernetes Cluster With Jenkins X
- Exploring Build Pack Pipelines
- Extending Build Pack Pipelines
- Extending Environment Pipelines
- What Now?

## Extending Jenkins X Pipelines

- What Are We Trying To Do?

[Creating A Kubernetes Cluster With Jenkins X And Importing The Application](#)

[Naming Steps And Using Multi-Line Commands](#)

[Working With Environment Variables And Agents](#)

[Overriding Pipelines, Stages, And Steps And Implementing Loops](#)

[Pipelines Without Buildpacks](#)

[Exploring The Syntax Schema](#)

[What Now?](#)

## [Using Jenkins X To Define And Run Serverless Deployments](#)

[What is Serverless Computing?](#)

[Serverless Deployments In Kubernetes](#)

[Which Types Of Applications Should Run As Serverless?](#)

[Why Do We Need Jenkins X To Be Serverless?](#)

[What Is Tekton And How Does It Fit Jenkins X?](#)

[Creating A Kubernetes Cluster With Jenkins X And Importing The Application](#)

[Installing Gloo and Knative](#)

[Creating A New Serverless Application Project](#)

[Using Serverless Deployments With Pull Requests](#)

[Limiting Serverless Deployments To Pull Requests](#)

[What Now?](#)

## [Choosing The Right Deployment Strategy](#)

[What Do We Expect From Deployments?](#)

[Creating A Kubernetes Cluster With Jenkins X And Creating A Sample Application](#)

[Using Serverless Strategy With Gloo And Knative \(GKE only\)](#)

[Using Recreate Strategy With Standard Kubernetes Deployments](#)

[Using RollingUpdate Strategy With Standard Kubernetes Deployments](#)

[Evaluating Whether Blue-Green Deployments Are Useful](#)

[About The World We Lived In](#)

[A Short Introduction To Progressive Delivery](#)

[A Quick Introduction To Istio, Prometheus, Flagger, And Grafana](#)

[Installing Istio, Prometheus, Flagger, And Grafana](#)

[Creating Canary Resources With Flagger](#)

[Using Canary Strategy With Flagger, Istio, And Prometheus](#)

[Rolling Back Canary Deployments](#)

[To Canary Or Not To Canary?](#)  
[Visualizing Rollouts Of Canary Deployments](#)  
[Which Deployment Strategy Should We Choose?](#)  
[What Now?](#)

[Applying GitOps Principles To Jenkins X](#)  
[Discussing The Cardinal Sin](#)  
[Creating A Kubernetes Cluster \(Without Jenkins X\)](#)  
[What Is Jenkins X Boot?](#)  
[Installing Jenkins X Using GitOps Principles](#)  
[Exploring The Changes Done By The Boot](#)  
[Verifying Jenkins X Boot Installation](#)  
[What Now?](#)

[Managing Third-Party Applications](#)  
[Creating A Kubernetes Cluster With Jenkins X](#)  
[Managing Application-Specific Dependencies](#)  
[Managing Third-Party Applications Through Permanent Environments](#)  
[Managing Third-Party Applications Running In The Development Environment](#)  
[Managing Third-Party Applications As Jenkins X Apps](#)  
[Using Any Helm Chart As A Jenkins X App](#)  
[Which Method For Installing and Managing Third-Party Applications Should We Use?](#)  
[What Now?](#)

[Now It's Your Turn](#)

[Contributions](#)

# Preface

When I finished the last book (“The DevOps 2.5 Toolkit: Monitoring, Logging, and Auto-Scaling Kubernetes”), I wanted to take a break from writing for a month or two. I thought that would clear my mind and help me decide which subject to tackle next. Those days were horrible. I could not make up my mind. So many cool and useful tech is emerging and being adopted. I was never as undecided as those weeks. Which should be my next step?

I could explore serverless. That’s definitely useful, and it might be considered the next big thing. Or I could dive into Istio. It is probably the biggest and the most important project sitting on top of Kubernetes. Or I could tackle some smaller subjects. Kaniko is the missing link in continuous delivery. Building containers might be the only thing we still do on the host level, and Kaniko allows us to move that process inside containers. How about security scanning? It is one of the things that are mandatory in most organizations, and yet I did not include it in “The DevOps 2.4 Toolkit: Continuous Deployment To Kubernetes”. Then there is Skaffold, Prow, Knative, and quite a few other tools that are becoming stable and very useful.

And then it struck me. Jenkins X does all those things and many more. I intentionally excluded it from “The DevOps 2.4 Toolkit: Continuous Deployment To Kubernetes” because at that time (the first half of 2018) it was still too green and it was changing all the time. It was far from stable. But the situation in early 2019 is different. While the project still evolves at a rapid pace and there are quite a few things left to do and polish, Jenkins X is being adopted by many. It has proved its usefulness. Its community is rising, its popularity is enormous, and it is one of the Kubernetes darlings.

So, the decision was made. This book will be dedicated to Jenkins X.

As with other books, the idea is to go deep into the subject. While the first few chapters might (intentionally) seem very basic, we’ll explore Jenkins X and many related tools in-depth, and we’ll try to see the bigger picture.

What is it? What does it do, and how does it do it? How does it affect our processes? How can we combine it with the things we already have, and which tools should be changed? Is it only about continuous delivery? Does it affect local development? Does it change how we operate Kubernetes?

As with all other books, I do not know in advance where this subject will lead me. I do not plan (much) in advance, and I did not write an index of everything I want to cover. Time will tell what will be the final scope.

What matters is that I want you to be successful, and I hope that this book will help you with your career path.



If you explore [jenkins-x.io](https://jenkins-x.io), you might notice some similarities between the content there and in this book. What you read here is not a copy from what's there. Instead, I decided to contribute part of the chapters to the community.

Eventually, you might get stuck and will need help. Or you might want to write a review or comment on the book's content. Please join the [DevOps20](#) Slack workspace and post your thoughts, ask questions, or participate in a discussion. If you prefer a more one-on-one conversation, you can use Slack to send me a private message or send an email to [viktor@farcic.com](mailto:viktor@farcic.com). All the books I have written are very dear to me, and I want you to have a good experience reading them. Part of that experience is the option to reach out to me. Don't be shy.

Please note that this one, just as the previous books, is self-published. I believe that having no intermediaries between the writer and the reader is the best way to go. It allows me to write faster, update the book more frequently, and have more direct communication with you. Your feedback is part of the process. No matter whether you purchased the book while only a few or all chapters were written, the idea is that it will never be truly finished. As time passes, it will require updates so that it is aligned with the change in technology or processes. When possible, I will try to keep it up to date and release updates whenever that makes sense. Eventually, things might change so much that updates are not a good option anymore, and that will be a sign that a whole new book is required. **I will keep writing as long as I continue getting your support.**

# About the Author

Viktor Farcic is a Developer Advocate at [CloudBees](#), a member of the [Docker Captains](#) group, and author.

He has coded using a plethora of languages, starting with Pascal (yes, he is old), Basic (before it got Visual prefix), ASP (before it got .NET suffix), C, C++, Perl, Python, ASP.NET, Visual Basic, C#, JavaScript, Java, Scala, etc. He never worked with Fortran. His current favorite is Go.

His big passions are containers, distributed systems, microservices, continuous delivery (CD), continuous deployment (CDP), and test-driven development (TDD).

He often speaks at community gatherings and conferences.

He wrote [The DevOps Toolkit Series](#) and [Test-Driven Java Development](#).

His random thoughts and tutorials can be found in his blog [TechnologyConversations.com](#).

# **Dedication**

To Sara and Eva.

## What Is Jenkins X?

To understand the **intricacies and inner workings** of Jenkins X, we need to understand Kubernetes. However, you do not need to understand Kubernetes to **use Jenkins X**. That is one of the main contributions of the project. Jenkins X allows us to harness the power of Kubernetes without spending an eternity learning the ever-growing list of the things Kubernetes does. Jenkins X helps us by simplifying complex processes into concepts that can be adopted quickly and without spending months in trying to figure out “the right way to do stuff.” It helps by removing and simplifying some of the problems caused by the overall complexity of Kubernetes and its ecosystem. If you are indeed a Kubernetes ninja, you will appreciate all the effort put into Jenkins X. If you’re not, you will be able to jump right in and harness the power of Kubernetes without ripping your hair out of frustration caused by Kubernetes complexity.

I’ll skip telling you that Kubernetes is a container orchestrator, how it manages our deployments, and how it took over the world by storm. You hopefully already know all that. Instead, I’ll define Kubernetes as a platform to rule them all. Today, most software vendors are building their next generation of software to be Kubernetes-native or, at least, to work better inside it. A whole ecosystem is emerging and treating Kubernetes as a blank canvas. As a result, new tools are being added on a daily basis, and it is becoming evident that Kubernetes offers near-limitless possibilities. However, with that comes increased complexity. It is harder than ever to choose which tools to use. How are we going to develop our applications? How are we going to manage different environments? How are we going to package our applications? Which process are we going to apply for application lifecycles? And so on and so forth. Assembling a Kubernetes cluster with all the tools and processes takes time and learning how to use what we assembled feels like a never-ending story. Jenkins X aims to remove those and other obstacles.

Jenkins X is opinionated. It defines many aspects of the software development lifecycle, and it makes decisions for us. It tells us what to do and how. It is like a tour guide on your vacation that shows you where to go, what to look at, when to take a photo, and when it’s time to take a

break. At the same time, it is flexible and allows power users to tweak it to fit their own needs.

The real power behind Jenkins X is the process, the selection of tools, and the glue that wraps everything into one cohesive unit that is easy to learn and use. We (people working in the software industry) tend to reinvent the wheel all the time. We spend countless hours trying to figure out how to develop our applications faster and how to have a local environment that is as close to production as possible. We dedicate time searching for tools that will allow us to package and deploy our applications more efficiently. We design the steps that form a continuous delivery pipeline. We write scripts that automate repetitive tasks. And yet, we cannot escape the feeling that we are likely reinventing things that were already done by others. Jenkins X is designed to help us with those decisions, and it helps us to pick the right tools for a job. It is a collection of the industry's best practices. In some cases, Jenkins X is the one defining those practices, while in others, it helps us in adopting those made by others.

If we are about to start working on a new project, Jenkins X will create the structure and the required files. If we need a Kubernetes cluster with all the tools selected, installed, and configured, Jenkins X will do that. If we need to create Git repositories, set webhooks, and create continuous delivery pipelines, all we need to do is execute a single `jx` command. The list of what Jenkins X does is vast, and it grows every day.

I won't go into details of everything Jenkins X does. That will come later. For now, I hope I got your attention. The critical thing to note is that you need to clear your mind from any Jenkins experience you might already have. Sure, Jenkins is there, but it is only a part of the package. Jenkins X is very different from the "traditional Jenkins". The differences are so massive that the only way for you to embrace it is to forget what you know about Jenkins and start from scratch.

I don't want to overwhelm you from the start. There's a lot of ground to cover, and we'll take one step at a time. For now, we need to install a few things.

## Installing Prerequisites

Before we jump into Jenkins X, we'll need a few tools that will be used throughout this book. I'm sure that you already have most (if not all) of

them, but I'll list them anyway.

I'm sure that you already have [Git](#). If you don't, you and I are not living in the same century. I would not even mention it, if not for GitBash. If you are using Windows, please make sure that you have GitBash (part of the Git setup) and to run all the commands from it. Other shells might work as well. Still, I tested all the commands on Windows with GitBash, so that's your safest bet. If, on the other hand, you are a macOS or Linux user, just fire up your favorite terminal.

Jenkins X CLI (we'll install it soon) will do its best to install [kubectl](#) and [Helm](#). However, the number of permutations of what we have on our laptops is close to infinite, and you're better off installing those two yourself.



At the time of this writing, December 2019, Jenkins X does not yet support Helm v3+. Please make sure that you're using Helm CLI v2+.

We'll need a Kubernetes cluster. I'll assume that you already have CLIs provided by your hosting vendor. You should be able to use (almost) any Kubernetes flavor to run Jenkins X, so the choice is up to you. I won't force you to use a particular vendor. Just as with kubectl and Helm, Jenkins X will try to install the appropriate CLI, but you might be better off installing it yourself. If you're planning on using an AWS EKS cluster, you probably already have the [AWS CLI](#) and [eksctl](#). If your preference is Google GKE, I'm sure that you have [gcloud](#). Similarly, if you prefer Azure, you likely have [Azure CLI](#) on your laptop. Finally, if you prefer something else, I'm sure you know which CLI fits your situation.

There is one restriction though. You can use (almost) any Kubernetes cluster, but it needs to be publicly accessible. The main reason for that lies in GitHub triggers. Jenkins X relies heavily on GitOps principles. Most of the events will be triggered by GitHub webhooks. If your cluster cannot be accessed from GitHub, you won't be able to trigger those events, and you will have difficulty following the examples.

Now, that might pose two significant issues. You might prefer to practice locally using Minikube or Docker Desktop. Neither of the two is

accessible from outside your laptop. You might have a corporate cluster that is inaccessible from the outside world. In those cases, I suggest you use a service from AWS, GCP, Azure, or from anywhere else. Each chapter will start with the instructions to create a new cluster, and it will end with instructions on how to destroy it (if you choose to do so). That way, the costs will be kept to a bare minimum. If you sign up with one of the Cloud providers, they will give you much more credit than what you will spend on the exercises from this book, even if you are the slowest reader in the world. If you’re not sure which one to pick, I suggest [Google Cloud Platform \(GCP\)](#). At the time of this writing, their managed Kubernetes offering called Google Kubernetes Engine (GKE) is the best cluster on the market.

Moving on to the final set of requirements...

A few examples will use [jq](#) to filter and format JSON output. Please install it.

Finally, we’ll perform some GitHub operations using [hub](#). Install it, if you don’t have it already.

That’s it. I’m not forcing you to use anything but the tools you should have anyway.

For your convenience, the list of all the tools we’ll use is as follows.

- [Git](#)
- GitBash (if using Windows)
- [kubectl](#)
- [Helm](#)
- [AWS CLI](#) and [eksctl](#) (if using AWS EKS)
- [gcloud](#) (if using Google GKE)
- [Azure CLI](#) (if using Azure AKS)
- [jq](#)
- [hub](#)

Now, let’s install Jenkins X CLI.

## Installing Jenkins X CLI



All the commands from this chapter are available in the [02-intro.sh](#) Gist.

If you are a **macOS** user, please install `jx` using `brew`.

```
1 brew tap jenkins-x/jx
2
3 brew install jx
```

If you are a **Linux** user, the instructions are as follows.



Please visit the [releases](#) page and replace `v1.3.634` in the command that follows with the latest version.

```
1 mkdir -p ~/.jx/bin
2
3 curl -L https://github.com/jenkins-x/jx/releases/download/v1.3.634/jx-linux-amd64
4 r.gz \
5   | tar xzv -C ~/.jx/bin
6
7 export PATH=$PATH:~/jx/bin
8
9 echo 'export PATH=$PATH:~/jx/bin' \
10    >> ~/.bashrc
```

Finally, Windows users can install the CLI using [Chocolatey](#).

```
1 choco install jenkins-x
```

Now we are ready to install Jenkins X.

## To Create A Cluster Or Not To Create A Cluster

I already mentioned that Jenkins X is much more than a tool for continuous integration or continuous delivery. One of the many features it has is to create a fully operational Kubernetes cluster and install the tools we might need to operate it efficiently. On the other hand, `jx` allows us to install Jenkins X inside an existing Kubernetes cluster as well.

So, we need to make a choice.

Do we want to let `jx` create a cluster for us, or are we going to install Jenkins X inside an existing cluster? The decision will largely depend on your current situation, as well as the purpose of the cluster.

If you plan to create a cluster only for the purpose of the exercises in this book, I recommend using `jx` to create a cluster, assuming that your favorite hosting vendor is one of the supported ones. Another reason for letting `jx` handle creation of the cluster lies in potential desire to have a dedicated cluster for continuous delivery. In both cases, we can use `jx create cluster` command.

On the other hand, you might already have a cluster where other applications are running and might want to add Jenkins X to it. In that case, all we have to do is install Jenkins X by executing `jx install`.

Let's go through the help of both `jx create cluster` and `jx install` commands and see what we've got.

```
1 jx create cluster help
```

Judging from the output, we can see that Jenkins X works with quite a few different providers. We can use it inside Azure AKS, AWS with `kops`, AWS EKS, Google GKE, Oracle OKE, IBM ICP, IBM IKS, Minikube, Minishift, OpenShift, and Kubernetes. Now, the last provider I mentioned is curious. Aren't all other providers just different flavors of Kubernetes? If they are, why do we have a provider called `kubernetes` on top of more specific providers (e.g., GKE)? The `kubernetes` provider allows us to run Jenkins X in (almost) any Kubernetes flavor. The difference between the `kubernetes` provider and all the others lies in additions that are useful only for those providers. Nevertheless, you can run Jenkins X in (almost) any Kubernetes flavor. If your provider is on the list, use it. Otherwise, pick `kubernetes` provider instead.

As a side note, do not trust the list I presented as being final. By the time you read this, Jenkins X might have added more providers to the list.

It is worthwhile mentioning that `jx` cannot create a cluster in all those providers, but that it can run there. A few of those cannot be created dynamically. Namely, OKE, ICP, and OpenShift. If you prefer one of those, you'll have to wait until we reach the part with the instructions to install Jenkins X in an existing cluster.

You'll also notice that `jq` will install some of the local dependencies if you do not have them already on your laptop. Which ones will be installed depends on your choice of the provider. For example, `gcloud` is installed only if you choose GKE as your provider. On the other hand, `kubectl` will be installed no matter the choice, as long as you do not have it already in your laptop.

So, if you do choose to use one of the cloud providers, `jq create cluster` is an excellent option, unless you already have a cluster where you'd like to install Jenkins X. If that's the case, or if you cannot use one of the cloud providers, you should be exploring the `jq install` command instead. It is a subset of `create cluster`. If we take a look at the supported providers in the `install` command, we'll see that they are the same as those we saw in `create cluster`.

```
1 jq install --help | grep "provider="
```

The output is as follows.

```
1 --provider='': Cloud service providing the Kubernetes cluster. Supported provider
2 aks, aws, eks, gke, icp, iks, jx-infra, kubernetes, minikube, minishift, oke, ope
3 nhift, pks
```

I'll show you how to use Jenkins X to create a GKE, EKS, and AKS cluster. If you do have access to one of those providers, I suggest you do follow the instructions. Even if you're already planning to install Jenkins X inside an existing cluster, it would be beneficial to see the benefits we get with the `jq create cluster` command. Further on, I'll show you how to install Jenkins X inside any existing cluster. It's up to you to choose which path you'd like to follow. Ideally, you might want to try them all, and get more insight into the differences between cloud providers and Kubernetes flavors.

No matter the choice, I will make sure that all four combinations are supported through the rest of the chapters. We'll always start with the instructions how to create a new cluster in GKE, EKS, and AKS, as well as how to install Jenkins X and everything else we might need in an existing cluster.

Before we proceed, please note that we'll specify most of the options through arguments. We could have skipped them and let `jq` ask us questions (e.g., how many nodes do you want?). Nevertheless, I believe

that using arguments is a better way since it results in a documented and reproducible process to create something. Ideally, `jx` should not ask us any questions. We can indeed accomplish that by running in the batch mode. I'll reserve that for the next chapter, and our first attempt will use a combination of arguments and questions.

Also, before we dive into the actual setup, we'll create a file `myvalues.yaml`.

```
1 echo "nexus:
2   enabled: false
3 " | tee myvalues.yaml
```

Among other things, Jenkins X installs Nexus where we can store our libraries. While that is useful for many projects, we will not need it in the examples that follow. By disabling it, we'll avoid reserving resources we won't use.

For your convenience, bookmarks to the relevant sub-chapters are as follows.

- [Creating A Google Kubernetes Engine \(GKE\) Cluster With jx](#)
- [Creating An Amazon Elastic Container Service for Kubernetes \(EKS\) Cluster With jx](#)
- [Creating An Azure Kubernetes Service \(AKS\) Cluster With jx](#)
- [Is Your Existing Cluster Ready For Jenkins X?](#)



You'll notice that some of the text in the before mentioned sections is repeated. That's intentional since I wanted each set of instructions to be self-sufficient, even though such repetition might annoy those who try multiple Kubernetes flavors or want to experience both the process of creating the cluster as well as installing Jenkins X inside an existing one.

If you prefer to create the cluster in one of the other providers, I recommend reading the instructions for one of the “big three” (AWS, Azure, or Google) since the requirements and the steps are very similar.

## Creating A Google Kubernetes Engine (GKE) Cluster With jx

Everything we do inside a Google Cloud Platform (GCP) is inside a project. That includes GKE clusters. If we are to let `jx` create a cluster for us, we need to know the name of the GCP project where we'll put it. If you do not have a project you'd like to use, please visit the [Manage Resources](#) page to create a new one. Make sure to enable billing for that project.

No matter whether you created a new project specifically for Jenkins X, or you chose to reuse one that you already have, we'll need to know its name. To simplify the process, we'll store it in an environment variable.

Please make sure to replace [...] with the name of the GCP project before executing the command that follows.

```
1 PROJECT=[...]
```

Now we're ready to create a GKE cluster with all the tools installed and configured. We'll name it `jx-rocks` (`-n`) and let it reside inside the project we just defined (`-p`). It'll run inside `us-east1` region (`-r`) and on `n1-standard-2` (2 CPUs and 7.5 GB RAM) machines (`-m`). Feel free to reduce that to `n1-standard-1` if you're concerned about the cost. Since GKE auto-scales nodes automatically, the cluster will scale up if we need more.

While at the subject of scaling, we'll have a minimum of three nodes (`--min-num-nodes`) and we'll cap it to five (`--max-num-nodes`).

We'll also set the default Jenkins X password to `admin` (`--default-admin-password`). Otherwise, the process will create a random one.

Finally, we'll set `jx-rocks` as the default environment prefix (`--default-environment-prefix`). A part of the process will create two GitHub repositories, one for staging and the other for the production environment.

The `jx-rocks` prefix will be used to form their names. We won't go into much detail about those environments and repositories just yet. That's reserved for one of the following chapters.

Feel free to change any of the values in the command that follows to suit your needs better. Or, keep them as they are. After all, this is only a practice, and you'll be able to destroy the cluster and recreate it later with different values.

```
1 jx create cluster gke \
2   --cluster-name jx-rocks \
3   --project-id $PROJECT \
4   --region us-east1 \
5   --machine-type nl-standard-2 \
6   --min-num-nodes 1 \
7   --max-num-nodes 2 \
8   --default-admin-password admin \
9   --default-environment-prefix jx-rocks
```

Let's explore what we're getting with that command. You should be able to correlate my explanation with the console output.

First, the GCP authentication screen should open asking you to confirm that you are indeed who you claim you are. If that does not happen, please open the link provided in the output manually.

Next, `jx` will ensure that all the GCP services we need (`container` and `compute`) are enabled.

Once we're authenticated, and the services are enabled, `jx` will create a cluster after you answer a couple of questions. The default values should suffice for now. It should take only a few minutes.

Once the GKE cluster is up and running, the process will create a `jx` Namespace. It will also modify your local `kubectl` context and create a ClusterRoleBinding that will give you the necessary administrative permissions.

Next, the installation of Jenkins X itself and a few other applications (e.g., ChartMuseum for storing Helm charts) will start. The exact list of apps that will be installed depends on the Kubernetes flavor, the type of setup, and the hosting vendor. But, before it proceeds, it'll need to ask us a few other questions. Which kind do we want to install? Static or serverless? Please answer with Serverless Jenkins X Pipelines with Tekton. Even though Jenkins X started its history with Jenkins, the preferred pipeline engine is [Tekton](#), which is available through the serverless flavor of Jenkins X. We'll discuss Tekton and the reasons Jenkins X is using it later.

At this point, `jx` will try to deduce your Git name and email. If it fails to do so, it'll ask you for that info.

The next in line is Ingress. The process will try to find it inside the `kube-system` Namespace and install it if it's not there. The process installs it through a Helm chart. As a result, Ingress will create a load balancer that will provide an entry point into the cluster. This is the step that might possibly fail during our setup. The GCP default quotas are very low, and you might not be allowed to create additional load balancers. If that's the case, please open the [Quotas](#) page, select those that are at the maximum, and click the *Edit Quotas* button. Increasing a quota is a manual process. Nevertheless, they do it relatively fast, so you should not have to wait very long.

Once the load balancer is created, `jx` will use its hostname to deduce the IP.

Since we did not specify a custom domain for our cluster, the process will combine the IP of the load balancer with the [nip.io](#) service to create a fully qualified domain, and we'll be asked whether we want to proceed using it. Type `y` or merely press the enter key to continue.

Jenkins X will *enable long-term logs storage* automatically and ask you for the *Google Cloud Zone*. Feel free to keep the one selected by default.

Next, we'll be asked a few questions related to Git and GitHub. You should be able to answer those. In most cases, all you have to do is confirm the suggested answer by pressing the enter key. As a result, `jx` will store the credentials internally so that it can continue interacting with GitHub on our behalf. It will also install the software necessary for correct functioning of those environments (Namespaces) inside our cluster.

We're almost done. Only one question is pending. You'll be asked to select the organization where you want to create the environment repository. Choose one from the list.

The process will create two GitHub repositories; `environment-jx-rocks-staging` that describes the staging environment and `environment-jx-rocks-production` for production. Those repositories will hold the definitions of those environments. For example, when you decide to promote a release to production, your pipelines will not install anything directly. Instead, they will push changes to `environment-jx-rocks-production` which will, in turn, trigger another job that will comply with the updated definition of the environment.

That's GitOps.

Nothing is done without recording a change in Git. Of course, for that process to work, we need new jobs in Jenkins, so the installation process created two jobs that correspond to those repositories. We'll discuss the environments in greater detail later.

Finally, the `kubectl` context was changed to point to the `jx` Namespace, instead of `default`.

We'll get back to the new cluster and the tools that were installed and configured in the [What Did We Get?](#) section. Feel free to jump there if you have no interest in other Cloud providers or how to install Jenkins X inside an existing cluster.

Next up: EKS.

## Creating An Amazon Elastic Container Service for Kubernetes (EKS) Cluster With jx

To interact with AWS through its CLI, we need environment variables `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, and `AWS_DEFAULT_REGION` (there are other ways, but I'll ignore them).

```
1 export AWS_ACCESS_KEY_ID=[...]
2
3 export AWS_SECRET_ACCESS_KEY=[...]
4
5 export AWS_DEFAULT_REGION=us-east-1
```

Please replace the first [...] with the AWS Access Key ID, and the second with the AWS Secret Access Key. I am assuming that you are already familiar with AWS and you know how to create those keys, or that you already have them. If that's not the case, please follow the instructions from the [Managing Access Keys for Your AWS Account Root User](#) page.

Now we're ready to create an EKS cluster. We'll name it `jx-rocks` (`--cluster-name`). It will run inside `us-west-2` region (`--region`) and on `t2.large` (2 CPUs and 8 GB RAM) machines (`--node-type`). Unlike with GKE, we won't get a Cluster Autoscaler out of the box, but we'll fix that later. For now, you can assume that there eventually will be autoscaling, so there's no need to worry whether the current capacity is enough. If anything, it is likely more than we will need from the start. Still, even

though autoscaling will come later, we'll set the current (`--nodes`) and the minimum (`--nodes-min`) number of nodes to three, and the maximum to six (`--nodes-max`). That will be converted into AWS Auto-Scaling Groups and, in case of a misstep, it'll protect us from ending up with more nodes than we can afford.

We'll also set the default Jenkins X password to `admin` (`--default-admin-password`). Otherwise, the process will create a random one. Finally, we'll set `jx-rocks` as the default environment prefix (`--default-environment-prefix`). A part of the process will create a few repositories (one for staging and the other for production), and that prefix will be used to form their names. We won't go into much detail about those environments and repositories just yet. That's reserved for one of the follow-up chapters.

Feel free to change any of the values in the command that follows to suit your needs better. Or, keep them as they are. After all, this is only a practice, and you'll be able to destroy the cluster and recreate it later on with different values.

```
1 jx create cluster eks \
2   --cluster-name jx-rocks \
3   --region $AWS_DEFAULT_REGION \
4   --node-type t2.large \
5   --nodes 3 \
6   --nodes-min 3 \
7   --nodes-max 6 \
8   --default-admin-password admin \
9   --default-environment-prefix jx-rocks
```

Let's explore what we're getting with that command. You should be able to correlate my explanation with the console output.



If you get stuck with the waiting for external loadbalancer to be created and update the nginx-ingress-controller service in kube-system namespace, you probably encountered a bug. To fix it, open the AWS console and remove the kubernetes.io/cluster/jx-rocks tag from the security group eks-cluster-sg-\*.



Do not be too hasty answering jx questions. For all other types of Kubernetes clusters, we can safely use the default answers (enter key). But, in the case of EKS, there is one question that we'll answer with a non-default value. I'll explain it in more detail when we get there. For now, keep an eye on the “would you like to register a wildcard DNS ALIAS to point at this ELB address?” question.

The process started creating an EKS cluster right away. This will typically take around ten minutes, during which you won't see any movement in jx console output. It uses CloudFormation to set up EKS as well as worker nodes, so you can monitor the progress by visiting the [CloudFormation page](#).

Next, the installation of Jenkins X itself and a few other applications (e.g., ChartMuseum for storing Helm charts) will start. The exact list of apps that will be installed depends on the Kubernetes flavor, the type of setup, and the hosting vendor. But, before it proceeds, it'll need to ask us a few other questions. Which kind do we want to install? Static or serverless? Please answer with Serverless Jenkins X Pipelines with Tekton. Even though Jenkins X started its history with Jenkins, the preferred pipeline engine is [Tekton](#), which is available through the serverless flavor of Jenkins X. We'll discuss Tekton and the reasons Jenkins X is using it later.

The next in line is Ingress. The process will try to find it inside the kube-system Namespace and install it if it's not there. The process installs it through a Helm chart. As a result, Ingress will create a load balancer that will provide an entry point into the cluster.

Jenkins X recommends using a custom DNS name to access services in your Kubernetes cluster. However, there is no way for me to know if you have a domain available for use or not. Instead, we'll use the [nip.io](#) service to create a fully qualified domain. To do that, we'll have to answer with n

to the question "would you like to register a wildcard DNS ALIAS to point at this ELB address?". As a result, we'll be presented with another question. "Would you like to wait and resolve this address to an IP address and use it for the domain?". Answer with `y` (or press the enter key since that is the default answer). The process will wait until Elastic Load Balancer (ELB) is created and use its hostname to deduce its IP.

You might be asked to *enable long-term logs storage*. Make sure to answer with `n`. We will not need it for our exercises and, at the time of this writing, it is still in the “experimental” phase.

Next, we'll be asked a few questions related to Git and GitHub. You should be able to answer those. In most cases, all you have to do is confirm the suggested answer by pressing the enter key. As a result, `jx` will store the credentials internally so that it can continue interacting with GitHub on our behalf. It will also install the software necessary for correct functioning of those environments (Namespaces) inside our cluster.

We're almost done. Only one question is pending. Select the organization where you want to create the environment repository? Choose one from the list.

The process will create two GitHub repositories; `environment-jx-rocks-staging` that describes the staging environment and `environment-jx-rocks-production` for production. Those repositories will hold the definitions of those environments. For example, when you decide to promote a release to production, your pipelines will not install anything directly. Instead, they will push changes to `environment-jx-rocks-production` which, in turn, will trigger another job that will comply with the updated definition of the environment.

## That's GitOps

Nothing is done without recording a change in Git. Of course, for that process to work, we need new jobs in Jenkins X, so the process created two jobs that correspond to those repositories. We'll discuss the environments in greater detail later.

Finally, the `kubectl` context was changed to point to the `jx` Namespace, instead of `default`.

As you can see, a single `jx create cluster` command did a lot of heavy lifting. Nevertheless, there is one piece missing. It did not create Cluster Autoscaler (it's not currently part of EKS). We'll add it ourselves so that we don't need to worry whether the cluster needs more nodes.

We'll add a few tags to the Autoscaling Group dedicated to worker nodes. To do that, we need to discover the name of the group. Fortunately, names follow a pattern which we can use to filter the results.

First, we'll retrieve the list of the AWS Autoscaling Groups, and filter the result with `jq` so that only the name of the matching group is returned.

```
1 ASG_NAME=$(aws autoscaling \
2     describe-auto-scaling-groups \
3     | jq -r ".AutoScalingGroups[] \
4     | select(.AutoScalingGroupName \
5     | startswith(\"eksctl-jx-rocks-nodegroup\")) \
6     .AutoScalingGroupName")
7
8 echo $ASG_NAME
```

We retrieved the list of all the groups and filtered the output with `jq` so that only those with names that start with `eksctl-$NAME-nodegroup` are returned. Finally, that same `jq` command retrieved the `AutoScalingGroupName` field and we stored it in the environment variable `ASG_NAME`. The last command output the group name so that we can confirm (visually) that it looks correct.

Next, we'll add a few tags to the group. Kubernetes Cluster Autoscaler will work with the one that has the `k8s.io/cluster-autoscaler/enabled` and `kubernetes.io/cluster/[NAME_OF_THE_CLUSTER]` tags. So, all we have to do to let Kubernetes know which group to use is to add those tags.

```
1 aws autoscaling \
2     create-or-update-tags \
3     --tags \
4     ResourceId=$ASG_NAME,ResourceType=auto-scaling-group,Key=k8s.io/cluster-autosc
5     er/enabled,Value=true,PropagateAtLaunch=true \
6     ResourceId=$ASG_NAME,ResourceType=auto-scaling-group,Key=kubernetes.io/cluster
7     x-rocks,Value=true,PropagateAtLaunch=true
```

The last change we'll have to do in AWS is to add a few additional permissions to the role. Just as with the Autoscaling Group, we do not know the name of the role, but we do know the pattern used to create it. Therefore, we'll retrieve the name of the role, before we add a new policy to it.

```

1 IAM_ROLE=$(aws iam list-roles \
2   | jq -r ".Roles[] \
3   | select(.RoleName \
4   | startswith(\"eksctl-jx-rocks-nodegroup\")) \
5   .RoleName")
6
7 echo $IAM_ROLE

```

We listed all the roles and we used `jq` to filter the output so that only the one with the name that starts with `eksctl-jx-rocks-nodegroup-0-NodeInstanceRole` is returned. Once we filtered the roles, we retrieved the `RoleName` and stored it in the environment variable `IAM_ROLE`.

Next, we need JSON that describes the new policy that will allow a few additional actions related to `autoscaling`. I already prepared one and stored it in the [vfarcic/k8s-specs](#) repository.

Now, let's put the new policy to the role.

```

1 aws iam put-role-policy \
2   --role-name $IAM_ROLE \
3   --policy-name jx-rocks-AutoScaling \
4   --policy-document https://raw.githubusercontent.com/vfarcic/k8s-specs/master/s
5   ling/eks-autoscaling-policy.json

```

Now that we have added the required tags to the Autoscaling Group and created the additional permissions that will allow Kubernetes to interact with the group, we can install the *cluster-autoscaler* Helm Chart from the stable channel. All we have to do now is execute `helm install stable/cluster-autoscaler`. However, since tiller (server-side Helm) has a lot of problems, Jenkins X does not use it by default. So, instead of using `helm install` command, we'll run `helm template` to output YAML files that we can use with `kubectl apply`.

```

1 mkdir -p charts
2
3 helm fetch stable/cluster-autoscaler \
4   -d charts \
5   --untar
6
7 mkdir -p k8s-specs/aws
8
9 helm template charts/cluster-autoscaler \
10  --name aws-cluster-autoscaler \
11  --output-dir k8s-specs/aws \
12  --namespace kube-system \
13  --set autoDiscovery.clusterName=jx-rocks \
14  --set awsRegion=us-west-2 \
15  --set sslCertPath=/etc/kubernetes/pki/ca.crt \
16  --set rbac.create=true
17
18 kubectl apply \

```

```
19      -n kube-system \
20      -f k8s-specs/aws/cluster-autoscaler/*
```

Once the Deployment is rolled out, the autoscaler should be fully operational.

You can see from the Cluster Autoscaler (CA) example how much `jx` helps. It took a single `jx` command to:

- create a cluster
- configure a cluster
- install a bunch of tools into that cluster
- and so on

For the only thing that `jx` did not do (creating the Cluster Autoscaler), we only had to execute five or six commands, not counting the effort I had to put to figure them out. Hopefully, EKS CA will be part of `jx` soon.

We'll get back to the new cluster and the tools that were installed and configured in the [What Did We Get?](#) section. Feel free to jump there if you have no interest in other Cloud providers or how to install Jenkins X inside an existing cluster.

Next up: AKS.

## Creating An Azure Kubernetes Service (AKS) Cluster With `jx`

We'll create an AKS cluster with all the tools installed and configured. We'll name the cluster with a unique value (`--cluster-name`) and let it reside inside its own group `jxrocks-group` (`--resource-group-name`). It'll run inside `eastus` location (`--location`) and on `Standard_D2s_v3` (2 CPUs and 8GB RAM) machines (`--node-vm-size`). The number of nodes will be set to three (`--nodes`).

We'll also set the default Jenkins X password to `admin` (`--default-admin-password`). Otherwise, the process will create a random one. Finally, we'll set `jx-rocks` as the default environment prefix (`--default-environment-prefix`). A part of the process will create a few repositories (one for staging and the other for production), and that prefix will be used to form their names. We won't go into much detail about those

environments and repositories just yet. That's reserved for one of the follow-up chapters.

Feel free to change any of the values in the command that follows to suit your needs better. Or, keep them as they are. After all, this is only a practice, and you'll be able to destroy the cluster and recreate it later with different values.

```
1 # Please replace [...] with a unique name (e.g., your GitHub user and a day and n
2 h).
3 # Otherwise, it might fail to create a registry.
4 # The name of the cluster must conform to the following pattern: '^[a-zA-Z0-9]*$'
5 CLUSTER_NAME=[...]
6
7 jx create cluster aks \
8   --cluster-name $CLUSTER_NAME \
9   --resource-group-name jxrocks-group \
10  --location eastus \
11  --node-vm-size Standard_D2s_v3 \
12  --nodes 3 \
13  --default-admin-password admin \
14  --default-environment-prefix jx-rocks
```

Let's explore what we're getting with that command. You should be able to correlate my explanation with the console output.

First, Azure authentication screen should open asking you to confirm that you are indeed who you claim you are. If that does not happen, please open the link provided in the output manually.

Once we're authenticated, `jx` will create a cluster. It should take around ten minutes.

Once the AKS cluster is up and running, the process will create a `jx` Namespace. It will also modify your local `kubectl` context.

Next, the installation of Jenkins X itself and a few other applications (e.g., ChartMuseum for storing Helm charts) will start. The exact list of apps that will be installed depends on the Kubernetes flavor, the type of setup, and the hosting vendor. But, before it proceeds, it'll need to ask us a few other questions. Which kind do we want to install? Static or serverless? Please answer with Serverless Jenkins X Pipelines with Tekton. Even though Jenkins X started its history with Jenkins, the preferred pipeline engine is [Tekton](#), which is available through the serverless flavor of Jenkins X. We'll discuss Tekton and the reasons Jenkins X is using it later.

At this point, `jx` will try to deduce your Git name and email. If it fails to do so, it'll ask you for that info.

The next in line is Ingress. The process will try to find it inside the `kube-system` Namespace and install it if it's not there. The process installs it through a Helm chart. As a result, Ingress will create a load balancer that will provide an entry point into the cluster.

Once the load balancer is created, `jx` will use its hostname to deduce its IP.

Since we did not specify a custom domain for our cluster, the process will combine that IP with the [nip.io](https://nip.io) service to create a fully qualified domain, and we'll be asked whether we want to proceed using it. Type `y` or merely press the enter key to continue.

You might be asked *to enable long-term logs storage*. Make sure to answer with `n`. We will not need it for our exercises and, at the time of this writing, it is still in the “experimental” phase.

Next, we'll be asked a few questions related to Git and GitHub. You should be able to answer those. In most cases, all you have to do is confirm the suggested answer by pressing the enter key. As a result, `jx` will store the credentials internally so that it can continue interacting with GitHub on our behalf. It will also install the software necessary for correct functioning of those environments (Namespaces) inside our cluster.

We're almost done. Only one question is pending. Select the organization where you want to create the environment repository? Choose one from the list.

The process will create two GitHub repositories; `environment-jx-rocks-staging` that describes the staging environment and `environment-jx-rocks-production` for production. Those repositories will hold the definitions of those environments. For example, when you decide to promote a release to production, your pipelines will not install anything directly. Instead, they will push changes to `environment-jx-rocks-production` which, in turn, will trigger another job that will comply with the updated definition of the environment.

That's GitOps.

Nothing is done without recording a change in Git. Of course, for that process to work, we need new jobs in Jenkins, so the process created two that correspond to those repositories. We'll discuss the environments in greater detail later.

Finally, the `kubectl` context was changed to point to the `jx` Namespace, instead of `default`.

We'll get back to the new cluster and the tools that were installed and configured in the [What Did We Get?](#) section. Feel free to jump there if you have no interest in how to install Jenkins X inside an existing cluster.

## Is Your Cluster Ready For Jenkins X?

If you're reading this, the chances are that you do not want to use `jx cluster create` to create a new cluster that will host Jenkins X. That is OK, and even welcome. That likely means that you are already experienced with Kubernetes and that you already have applications running in Kubernetes. That's a sign of maturity and your desire to add Jenkins X to the mix of whichever applications you are already running there. After all, it would be silly to create a new cluster for each set of applications.

However, using an existing Kubernetes cluster is risky. Many people think that they are so smart that they will assemble their Kubernetes cluster from scratch. The story goes something like:

“We’re so awesome that we don’t need tools like Rancher to create a cluster for us.”

...or better yet...

“We’ll do it with `kubeadm`. ”

Then, after a lot of sweat, we announce that the cluster is operational, only to discover that there is no StorageClass or that networking does not work. So, if you assembled your own cluster and you want to use Jenkins X inside it, you need to ask yourself whether that cluster is set up correctly.

- Does it have everything we need?

- Does it comply with standards, or did you tweak it to meet your corporate restrictions?
- Did you choose to remove StorageClass because all your applications are stateless?
- Were you forced by your security department to restrict communication between Namespaces?
- Is the Kubernetes version too old?

We can answer those and many other questions by running compliance tests.

Before we proceed, we'll verify whether the cluster we're hoping to use meets the requirements. Fortunately, `jx` has a command that can help us. We can run compliance tests and check whether there is anything "suspicious" in the results. Among many other things, `jx` has its own implementation of the [Sonobuoy](#) SDK.

So, what is Sonobuoy? It is a diagnostic tool that makes it easier to understand the state of a Kubernetes cluster by running a set of Kubernetes conformance tests in an accessible and non-destructive manner.

Sonobuoy supports Kubernetes versions 1.11, 1.12 and 1.13, so bear that in mind before running it in your cluster. In fact, if your Kubernetes cluster is older than 1.11, you likely think that creating a cluster is a one-time deal and that there is nothing to maintain or upgrade.

Given that I do not know whether your cluster complies with Kubernetes specifications and best practices, I cannot guarantee that Jenkins X installation will be successful. Compliance tests should give us that kind of comfort.

Before we proceed with the compliance tests, I must warn you that the tests will run for over an hour. Is it worth it? That depends on your cluster. Jenkins X does not need anything "special". It assumes that your Kubernetes cluster has some bare minimums and that it complies with Kubernetes standards. If you created it with one of the Cloud providers and you did not go astray from the default setup and configuration, you can probably skip running the compliance tests.

On the other hand, if you baked your own Kubernetes cluster, or if you customized it to comply with some corporate restrictions, running

compliance tests might be well worth the wait. Even if you’re sure that your cluster is ready for Jenkins X, it’s still a good idea to run them. You might find something you did not know exists or, to be more precise, you might see that you are missing things you might want to have.

Anyway, the choice is yours. You can run the compliance tests and wait for over an hour, or you can be brave and skip right into [Installing Jenkins X In An Existing Kubernetes Cluster](#).

```
1 jx compliance run
```

Once the compliance tests are running, we can check their status to see whether they finished executing.

```
1 jx compliance status
```

The output is as follows.

```
1 Compliance tests are still running, it can take up to 60 minutes.
```

If you got no compliance status found message instead, you were too hasty, and the tests did not yet start. If that’s the case, re-execute the `jx compliance status` command.

We can also follow the progress by watching the logs.

```
1 jx compliance logs -f
```

After a while, it’ll start churning a lot of logs. If it’s stuck, you executed the previous command too soon. Cancel with `ctrl+c` and repeat the `jx compliance logs -f` command.

Once you get bored looking at endless logs entries, stop following logs by pressing `ctrl+c`.

The best thing you can do right now is to find something to watch on Netflix since there’s at least an hour left until the tests are finished.

We’ll know whether the compliance testing is done by executing `jx compliance status`. If the output is the same as the one that follows, the execution is finished.

```
1 Compliance tests completed. Use `jx compliance results` to display the results.
```

Let's see the results.

```
1 jx compliance results
```

If the statuses of all the tests are `PASSED`, you're probably good to go. I used the word "probably" since there is an infinite number of things you might have done to your cluster that are not covered by the compliance tests. Nevertheless, with everything `PASSED`, it is very likely that everything will run smoothly. By "everything", I don't mean only Jenkins X, but whatever else you're planning to deploy to your cluster.

What happens if one of the tests failed? The obvious answer is that you should fix the issue first. A little less obvious response would be that it might or might not affect Jenkins X and whatever else we'll do in that cluster. Still, no matter whether the issue is going to affect us or not, you should fix it because you should have a healthy and conformant Kubernetes cluster.

We don't need compliance tests anymore, so let's remove them from the system and free some resources.

```
1 jx compliance delete
```

## Installing Jenkins X In An Existing Kubernetes Cluster

I will assume that your cluster passed compliance tests or, in case you did not execute them, that you are confident that it works according to Kubernetes specifications and best practices. Also, I'm assuming that it is accessible from the outside world. To be more precise, the cluster needs to be reachable from GitHub, so that it can send webhook notifications whenever we push some code changes. Please note that the accessibility requirement is valid only for the purpose of the exercises. In a "real world" situation you might use a Git server that is inside your network (e.g., GitLab, BitBucket, GitHub Enterprise, etc.).

Now that we (might have) run the compliance tests and that they showed that our cluster complies with Kubernetes, we can proceed and install Jenkins X.

We'll need a few pieces of information before we install the tools we need. The first in line is the IP address.

Typically, your cluster should be accessible through an external load balancer. Assuming that we can guarantee its availability, an external load balancer provides a stable entry point (IP address) to the cluster. At the same time, its job is to ensure that the requests are forwarded to one of the healthy nodes of the cluster. That is, more or less, all we need an external load balancer for.

If you do have an external LB, please get its IP address. If you don't, you can use the IP address of one of the worker nodes of the cluster, as long as you understand that a failure of that node would make everything inaccessible.

Please replace [...] with the IP of your load balancer or one of the worker nodes before you execute the command that follows.

```
1 LB_IP=[...]
```

Next, we need a domain. If you already have one, make sure that its DNS records are pointing to the cluster. Please replace [...] with the domain before you execute the command that follows.

```
1 DOMAIN=[...]
```

If you do NOT have a domain, we can combine the IP with the [nip.io](#) service to create a fully qualified domain. If that's the case, please execute the command that follows.

```
1 DOMAIN=$LB_IP.nip.io
```

We need to find out which provider to use. The available providers are the same as those you saw through the `jx create cluster help` command. For your convenience, we'll list them again through the `jx install` command.

```
1 jx install --help | grep "provider="
```

The output is as follows.

```
1 --provider='': Cloud service providing the Kubernetes cluster. Supported provider
2 aks, aws, eks, gke, icp, iks, jx-infra, kubernetes, minikube, minishift, oke, ope
3 nhift, pks
```

As you can see, we can install Jenkins in AKS, AWS (created with kops), EKS, GKE, ICP, IKS, Minikube, Minishift, OKE, OpenShift, and PKS.

The two I skipped from that list are `jx-infra` and `kubernetes`. The former is used mostly internally by the maintainers of the project, while the latter (`kubernetes`) is a kind of a wildcard provider. We can use it if our Kubernetes cluster does not match any of the available providers (e.g., Rancher, Digital Ocean, etc.).

All in all, if your Kubernetes is among one of the supported providers, use it. Otherwise, choose the `kubernetes` provider. There are two exceptions though. Minikube and Minishift run locally and are not accessible from GitHub. Please avoid them since some of the features will not be available. The main ones missing are GitHub webhook notifications. While that might sound like a minor issue, they are a crucial element of the system we're trying to build. Jenkins X relies heavily on GitOps which assumes that any change is stored in Git and that every push might potentially initiate some processes (e.g., deployment to the staging environment).

Please replace `[...]` with the selected provider in the command that follows.

```
1 PROVIDER=[...]
```

Do you have a NGINX Ingress controller running in your cluster? If you don't, `jx` will install it for you. In that case, feel free to skip the commands that declare the `INGRESS_*` variables. Also, when we come to the `jx install` command, remove the arguments `--ingress-namespace` and `--ingress-deployment`.

On the other hand, if you do have a NGINX Ingress controller, we need to find out in which Namespace you installed it. Let's list them and see which one hosts Ingress.

```
1 kubectl get ns
```

The output is as follows.

```
1 NAME          STATUS AGE
2 default      Active 10m
3 ingress-nginx Active 6m
4 kube-public  Active 10m
5 kube-system  Active 10m
```

In my case, it's `ingress-nginx`. In yours, it might be something else. Or, it might be inside the `kube-system` Namespace. If that's the case, list the

Pods with `kubectl --namespace kube-system get pods` to confirm that it's there.

Before executing the command that follows, please replace `[...]` with the Namespace where Ingress resides.

```
1 INGRESS_NS=[...]
```

Next, we need to find out the name of the Ingress Deployment.

```
1 kubectl --namespace $INGRESS_NS get deployments
```

The output, in the case of my cluster, is as follows (yours might differ).

```
1 NAME          DESIRED CURRENT UP-TO-DATE AVAILABLE AGE
2 nginx-ingress-controller 1       1       1       1      7m
```

In my case, the Deployment is called `nginx-ingress-controller`. Yours is likely named the same. If it isn't, please modify the command that follows accordingly.

```
1 INGRESS_DEP=nginx-ingress-controller
```

Now we are finally ready to install Jenkins X into your existing Kubernetes cluster. Please make sure to remove `--ingress-*` arguments if you do not have a NGINX Ingress controller in your cluster and you want `jx` to install it.

```
1 jx install \
2   --provider $PROVIDER \
3   --external-ip $LB_IP \
4   --domain $DOMAIN \
5   --default-admin-password admin \
6   --ingress-namespace $INGRESS_NS \
7   --ingress-deployment $INGRESS_DEP \
8   --default-environment-prefix jx-rocks
```

If, by any chance, you followed the instructions for GKE, EKS, or AKS, you'll notice that `jx install` executes the same steps as those performed by `jx cluster create`, except that the latter creates a cluster first. You can think of `jx install` as a subset of `jx cluster create`.

- Which kind do we want to install? Static or serverless?

Please answer with Serverless Jenkins X Pipelines with Tekton. We'll explore the serverless option later.

The process will create a `jx` Namespace. It will also modify your local `kubectl` context.

At this point, `jx` will try to deduce your Git name and email. If it fails to do so, it'll ask you for that info.

The next in line is Ingress. The process will try to find it inside the `kube-system` Namespace and install it if it's not there. The process installs it through a Helm chart.

You might be asked *to enable long term logs storage*. Make sure to answer with `n`. We will not need it for our exercises and, at the time of this writing, it is still in the “experimental” phase.

Next, we'll be asked a few questions related to Git and GitHub. You should be able to answer those. In most cases, all you have to do is confirm the suggested answer by pressing the enter key. As a result, `jx` will store the credentials internally so that it can continue interacting with GitHub on our behalf. It will also install the software necessary for correct functioning of those environments (Namespaces) inside our cluster.

Finally, the installation of Jenkins X itself and a few other applications (e.g., ChartMuseum for storing Helm charts) will start. The exact list of apps that will be installed depends on the Kubernetes flavor, the type of the setup, and the hosting vendor. But, before it proceeds, it'll need to ask us a few more questions.

We're almost done. Only one question is pending. Select the organization where you want to create the environment repository? Choose one from the list.

The process will create two GitHub repositories; `environment-jx-rocks-staging` that describes the staging environment and `environment-jx-rocks-production` for production. Those repositories will hold the definitions of those environments. For example, when you decide to promote a release to production, your pipelines will not install anything directly. Instead, they will push a change to `environment-jx-rocks-production` which, in turn, will trigger another job that will comply with the updated definition of the environment.

That's GitOps.

Nothing is done without recording a change in Git. Of course, for that process to work, we need new jobs in Jenkins, so the process created two jobs that correspond to those repositories. We'll discuss the environments in greater detail later.

Finally, the `kubectl` context was changed to point to the `jx` Namespace, instead of `default`.

Now you're ready to use Jenkins X.

## What Did We Get?

No matter whether you executed `jx cluster create` or `jx install`, it was a single command (Cluster Autoscaler in AWS is an exception). With that single command, we accomplished a lot.

We created a Kubernetes cluster (unless you executed `jx install`). We got a few Namespaces, a few GitHub repositories. We got Ingress (unless it already existed in the cluster). We got a bunch of ConfigMaps and Secrets that are essential for what we're trying to accomplish, and yet we will not discuss them just yet. Most importantly, we got quite a few applications that are essential for our yet-to-be-discovered goals. What are those applications? Let's check it out.

```
1 kubectl --namespace jx get pods
```

The output is as follows.

1 NAME	READY	STATUS	RESTARTS	AGE
2 crier-7c58ff4897-...	1/1	Running	0	3m27s
3 deck-b5b568797-...	1/1	Running	0	3m26s
4 deck-b5b568797-...	1/1	Running	0	3m26s
5 hook-6596bbfffb9-...	1/1	Running	0	3m23s
6 hook-6596bbfffb9-...	1/1	Running	0	3m23s
7 horologium-...	1/1	Running	0	3m22s
8 jenkins-x-chartmuseum-...	1/1	Running	0	2m53s
9 jenkins-x-controllerbuild-...	1/1	Running	0	2m40s
10 jenkins-x-controllerrole-...	1/1	Running	0	2m37s
11 jenkins-x-heapster-...	2/2	Running	0	2m16s
12 pipeline-...	1/1	Running	0	3m20s
13 pipelinerunner-...	1/1	Running	0	3m17s
14 plank-...	1/1	Running	0	3m16s
15 sinker-...	1/1	Running	0	3m14s
16 tekton-pipelines-controller-...	1/1	Running	0	5m2s
17 tekton-pipelines-webhook-...	1/1	Running	0	5m1s
18 tide-...	1/1	Running	0	3m13s

As you can see, there are quite a few tools in that Namespace. I won't dive into them just yet. We'll become familiar with each as we're progressing.

What matters for now is that we got everything we need to manage the full lifecycle of our applications. More importantly, we got a process to guide us through that lifecycle. We'll explore the tools and the process in the follow-up chapters. For now, let's just say that this is awesome. We got a lot (much more than what I shared with you so far) from the execution of a single command.

## What Now?

Now you know how to create a Jenkins X cluster or how to install it inside an existing Kubernetes cluster. We're ready to start exploring its features. But, before we get there, I'd like to share with you the commands to destroy your cluster or, if that's your preference, to uninstall Jenkins X. That way, you can undo what we did, or you can take a break at the end of each chapter without paying your hosting vendor for unused resources. Each chapter will start with the instructions that will get you up and running in no time.

In other words, if you are not planning to jump into the next chapter right away, you can use the commands that follow to undo what we did (destroy the cluster or uninstall Jenkins X). On the other hand, jump into the next chapter right away if you are still full of energy and want to continue reading.

If you do plan to delete the cluster or to uninstall Jenkins X, you should probably remove the GitHub repositories and a few local files. When we install Jenkins X again, those repositories and files will be recreated.



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

```
1 GH_USER=[...]
2
3 hub delete -y \
4   $GH_USER/environment-jx-rocks-staging
5
6 hub delete -y \
7   $GH_USER/environment-jx-rocks-production
8
9 rm -rf ~/.jx/environments/$GH_USER/environment-jx-rocks-*
```

We deleted the two repositories dedicated to environments, even though we did not explore them just yet. Environments are critical, and we will go into details in one of the next chapters.

The rest of the instructions depend on the Kubernetes flavor you used and whether you chose to create a cluster with `jx cluster create` or you installed Jenkins X in an existing cluster.

## Deleting GKE Cluster And Unused Resources

Please use the instructions that follow to delete your GKE cluster if it's dedicated exclusively to the exercises from this book and if you're not planning to jump into the next chapter right away. That way, you won't be paying for resources you're not using. The next chapter will provide Gists with the instructions on how to recreate the cluster in a matter of minutes.

```
1 gcloud container clusters \
2     delete jx-rocks \
3     --region us-east1 \
4     --quiet
```

With the cluster gone, there are still some resources that were not removed. Namely, the disks are now unused, but they still exist. We can remove them with the command that follows.

```
1 gcloud compute disks delete \
2     --zone us-east1-b \
3     $(gcloud compute disks list \
4     --filter="zone:us-east1-b AND -users:\"" \
5     --format="value(id)") \
6 gcloud compute disks delete \
7     --zone us-east1-c \
8     $(gcloud compute disks list \
9     --filter="zone:us-east1-c AND -users:\"" \
10    --format="value(id)") \
11 gcloud compute disks delete \
12    --zone us-east1-d \
13    $(gcloud compute disks list \
14    --filter="zone:us-east1-d AND -users:\"" \
15    --format="value(id)")
```



You will likely see an error stating that a disk cannot be deleted. That's because the command

That command listed all disks that do not have a user assigned (not used). The list of those disks is then passed to the `disks delete` command that

removed them one by one if you confirmed the action.

## Deleting EKS Cluster And Related Resources

When we created Cluster Autoscaler, we had to add a policy that will allow it to manipulate AWS Auto-Scaling Groups. Just as with the ELB, we need to delete that policy before we delete the EKS cluster. Otherwise, cluster removal would fail.

```
1 IAM_ROLE=$(aws iam list-roles \
2     | jq -r ".Roles[] \
3     | select(.RoleName \
4     | startswith(\"eksctl-jx-rocks-nodegroup\")) \
5     .RoleName")
6
7 echo $IAM_ROLE
8
9 aws iam delete-role-policy \
10    --role-name $IAM_ROLE \
11    --policy-name jx-rocks-AutoScaling
```

We retrieved the name of the role by filtering the results with `jq`. Once we got the role, we used it to delete the policy we created while we were setting up Cluster Autoscaler.

Now we are ready to delete the EKS cluster.

```
1 eksctl delete cluster -n jx-rocks
```

## Deleting AKS Cluster And Related Resources

AKS cluster is by far the easiest one to remove thanks to Azure Resource Groups. Instead of trying to figure out what needs to be deleted besides the cluster itself, we can just delete the whole group.

```
1 az group delete \
2     --name jxrocks-group \
3     --yes
```

Unfortunately, we are still left with entries in our `kubectl config`, so let's delete those as well.

```
1 kubectl config delete-cluster $CLUSTER_NAME
2
3 kubectl config delete-context $CLUSTER_NAME
4
5 kubectl config unset \
6     users.clusterUser_jxrocks-group_$CLUSTER_NAME
```

## Uninstalling Jenkins X

If you chose to install Jenkins X in an existing Kubernetes cluster, you can remove it with a single `jx uninstall` command.

```
1 jx uninstall \
2   --context $(kubectl config current-context) \
3   --batch-mode
```

# Exploring Quickstart Projects

Starting a new Jenkins X project is easy. The first time we create one, it looks and feels like magic. All we have to do is answer a few questions, and a few moments later we have:

- a full-blown continuous delivery pipeline
- GitHub webhook that triggers the pipeline
- a mechanism to promote a release to different environments
- a way to preview pull requests

...and quite a few other things.

However, that “magic” might be overwhelming if we accept it without understanding what’s going on behind the scenes. Our goal is to leverage the power we’re given. We need to get a grip on the tools involved in the process, and we need to understand the intricacies of the flow that will ultimately lead to a fast, reliable, and (mostly) hands-free approach to delivering our applications.

We’ll create a new cluster with Jenkins X (unless you already have one) and create a quickstart project. We’ll use it as an enabler that will allow us to explore some of the essential components provided by Jenkins X. That will give us base knowledge we’ll need later (in the next chapters) when we examine how to set up projects that will perform exactly what we need. We will not go into details of the process and the tools involved just yet. For now, the objective is to get a very high-level overview and an overall understanding of how Jenkins X works. More detailed descriptions will follow.

For that, we need a Kubernetes cluster with Jenkins X.

## Creating A Kubernetes Cluster With Jenkins X

Jenkins X runs on (almost) any Kubernetes cluster, so I’ll let you choose whether you want to use one you already have, or create a new one. As long as Jenkins X is running and is accessible, you should be good to go.



All the commands from this chapter are available in the [03-quickstart.sh](#) Gist.

For your convenience, I have created a few Gists that you can use. Feel free to use them as they are, adapt them to your own needs, or skip them altogether and create your cluster and install Jenkins X on your own.



The gists that follow use `-b` to run in the batch mode and they assume that this is not the first time you have created a cluster with `jx`. If that's not the case and this is indeed the first time you're creating a `jx` cluster, it will not have some of the default values like GitHub user, and the installation might fail. Make sure to remove `-b` from the `jx create cluster` command inside the Gists if this is NOT the first time you're creating a cluster with `jx`.

- Create a new serverless **GKE** cluster: [gke-jx-serverless.sh](#)
- Create a new serverless **EKS** cluster: [eks-jx-serverless.sh](#)
- Create a new serverless **AKS** cluster: [aks-jx-serverless.sh](#)
- Use an **existing** serverless cluster: [install-serverless.sh](#)



Please note that the Gists have the section to “destroy the cluster”. Do not execute the commands from there until you’re finished with this chapter and you do not plan to continue using it for the next.

Now that we have a cluster and that Jenkins X is up and running, we can proceed and create our first quickstart project.

## Creating A Quickstart Project

Quickstart projects provide an easy way to start development of a new application.

Traditionally, creating a new project is a tedious process that involves many different components. Obviously, we need to write the code of our new application, as well as tests. On top of that, we need a mechanism to compile the code, to run the tests, to create a distribution, and so on and so

forth. But it does not end there. Local development is only the beginning. We need to run performance, integration, and other types of tests that are too cumbersome to run locally. We need to deploy our application to different environments so that we can validate its readiness. We need to deal with branches and pull requests. A lot of things need to happen before a new release is deployed to production.

Quickstarts help us with those and some other tasks. They allow us to skip the tedious process and be up-and-running with a new project in a matter of minutes. Later on, once we get a better understanding of what we need, we might need to modify the code and the configurations provided by quickstarts. That will be the subject of the follow-up writings. For now, our objective is to start a new project with the least possible effort, while still getting most of the things we need for local development as well as for the application's lifecycle that ends with the deployment to production.

That was enough of an introduction to Jenkins X quickstarts. We'll explore details through practical examples.

Like most other `jx` commands, we can create a quickstart project using through the interactive or the batch mode. We'll take a look at the former first.

```
1 jx create quickstart
```

First, we will be asked to confirm our Git user name as well as the organization. Next, we need to type the name of the new repository that `jx` will create for us. After those questions, the output is as follows.

```
1 ? select the quickstart you wish to create [Use arrows to move, type to filter]
2 > android-quickstart
3 angular-io-quickstart
4 aspnet-app
5 dlang-http
6 golang-http
7 jenkins-cwp-quickstart
8 jenkins-quickstart
9 node-http
10 node-http-watch-pipeline-activity
11 open-liberty
12 python-http
13 rails-shopping-cart
14 react-quickstart
15 rust-http
16 scala-akka-http-quickstart
17 spring-boot-http-gradle
18 spring-boot-rest-prometheus
19 spring-boot-watch-pipeline-activity
20 vertx-rest-prometheus
```

We can see that there are quite a few types of projects we can create. All we have to do is select one of those. While that is helpful at the beginning, I prefer running all the commands in the batch mode. If we'd proceed, we'd need to answer a few questions like the name of the project and a few others. In the batch mode, instead of answering questions, we specify a few values as command arguments and, as a result, we end up with a documented way to reproduce our actions. It's easier and more reliable to have a README file with self-contained commands than to document the steps by saying things like "answer with this for the first question, with that for the second, and so on."

Please cancel the current command by pressing *ctrl+c*. We'll execute `jx create quickstart` again, but with a few additional arguments. We'll choose `golang-http` as the template, we'll name the project `jx-go`, and we'll use `-b` (short for batch mode) argument to let `jx` know that there is no need to ask us any questions. That does not mean that we will specify all the arguments we need, but rather those that differ from one project to another. When running in batch mode, `jx` will use the default values or those from previous executions.

Don't worry if you do not work with Go. We'll use it only as an example. The principles we'll explore through practical exercises apply to any programming language.

Here we go.

```
1 jx create quickstart \
2   --filter golang-http \
3   --project-name jx-go \
4   --batch-mode
```

The output is too big to be presented here, so I'll walk you through the steps `jx` performed while you're looking at your screen.

We got the `jx-go` directory for our new Go application. Later on, it was converted into a Git repository, and `jx` copied the files from the pack (quickstart) dedicated to Go. Once it finished, it pushed the files to GitHub, and it created a project in Jenkins. As a result, the first build of the new Jenkins pipeline started running immediately. We'll explore Jenkins projects in more detail later.

If you're wondering where do those quickstart projects come from, the answer is GitHub. The community created an organization called *jenkins-x-quickstarts* that contains the repositories hosting the quickstarts.



## A note to Windows users

Git Bash might not be able to use the `open` command. If that's the case, replace `open` with `echo`. As a result, you'll get the full address that should be opened directly in your browser of choice.

```
1 open "https://github.com/jenkins-x-quickstarts"
```

Jenkins X also made a local copy of the repository in the `~/.jx/draft/packs/github.com/jenkins-x-buildpacks` directory. Let's see what's inside.

```
1 ls -1 ~/.jx/draft/packs/github.com/jenkins-x-buildpacks/jenkins-x-kubernetes/packs
```

The output is as follows.

```
1 C++
2 D
3 apps
4 appserver
5 charts
6 csharp
7 custom-jenkins
8 cwp
9 docker
10 docker-helm
11 dropwizard
12 environment
13 git
14 go
15 go-mongodb
16 gradle
17 helm
18 imports.yaml
19 javascript
20 jenkins
21 liberty
22 maven
23 maven-javall
24 ml-python-gpu-service
25 ml-python-gpu-training
26 ml-python-service
27 ml-python-training
28 nop
29 php
30 python
31 ruby
```

```
32 rust
33 scala
34 swift
35 typescript
```

We can see that it matches the output we got from the `jx create quickstart` command, even though the names of the directories are not the same.

Since we used the quickstart for Go language, we might just as well take a look at its template files.

```
1 ls -1 ~/.jx/draft/packs/github.com/jenkins-x-buildpacks/jenkins-x-kubernetes/packs
```

The output is as follows.

```
1 Dockerfile
2 Makefile
3 charts
4 pipeline.yaml
5 preview
6 skaffold.yaml
7 watch.sh
```

I'll let you explore those files on your own. Just remember that they are not used as-is, but rather serve as templates that are processed by `jx create quickstart` command in an attempt to create usable, yet customized, projects. Later on, we'll also learn how to create custom quickstart packs.

## Exploring Quickstart Project Files

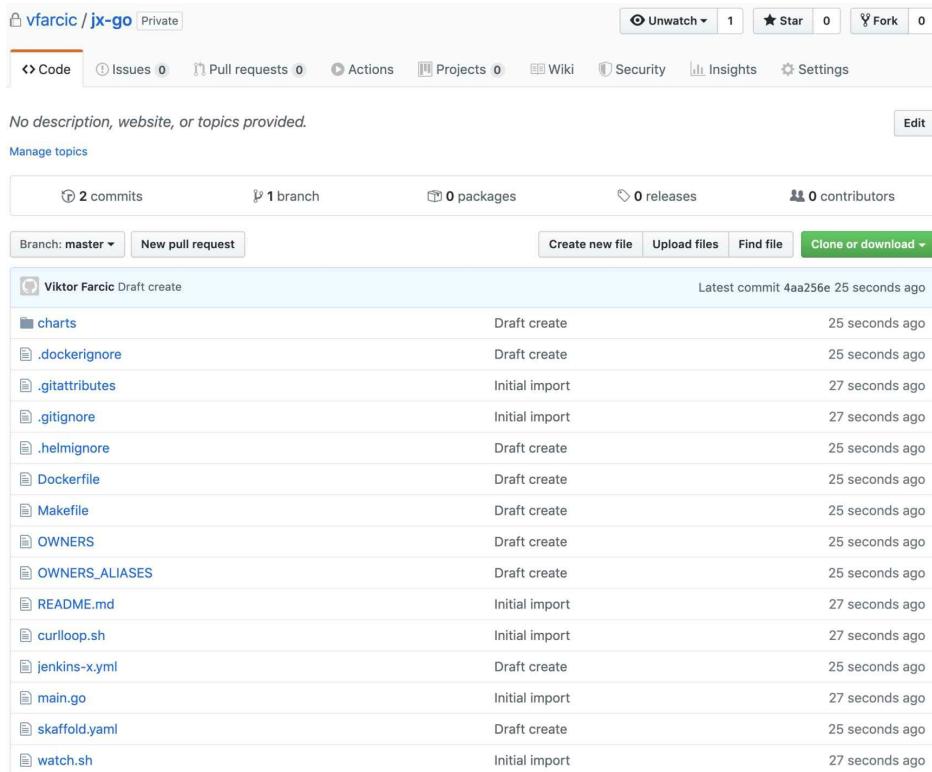
Let's see what did Jenkins X create and push to GitHub.



Please replace [...] with your GitHub username before executing the commands that follow.

```
1 GH_USER=[...]
2
3 open "https://github.com/$GH_USER/jx-go"
```

We can see that Jenkins X created quite a few files.



**Figure 3-1: GitHub repository with a newly created Jenkins X quickstart project**

The repository was also created locally, so let's take a closer look at the generated files.

```
1 cd jx-go
2
3 ls -l
```

The output is as follows.

```
1 Dockerfile
2 Makefile
3 OWNERS
4 OWNERS_ALIASES
5 README.md
6 charts
7 curlloop.sh
8 jenkins-x.yml
9 main.go
10 skaffold.yaml
11 watch.sh
```

Let's go through each of those files and directories and explore what we got. The first in line is *Makefile*.

```
1 cat Makefile
```

I won't go into much detail about the Makefile since it is made specifically for Go applications and that might not be your favorite language. Not all quickstart packs use Makefile. Instead, each tends to leverage methods appropriate for the given language and programming framework to accomplish the required tasks. In this case, the Makefile has targets to perform operations to build, test, install, and so on.

The next in line is Dockerfile.

```
1 cat Dockerfile
```

The output is as follows.

```
1 FROM scratch
2 EXPOSE 8080
3 ENTRYPOINT ["/jx-go"]
4 COPY ./bin/ /
```

In the case of Go, there's not much needed. It uses a very lightweight base image (`scratch`), it exposes a port, it creates an entrypoint that will execute the binary (`jx-go`), and, finally, it copies that binary.

Unlike Dockerfile that I'm sure you're already familiar with, [Skaffold](#) might be one of the tools you haven't used before.

Skaffold handles the workflow for building, pushing, and deploying applications to Kubernetes clusters, as well as for local development. We'll explore it in more detail later. For now, we'll take a brief look at the `skaffold.yaml` file.

```
1 cat skaffold.yaml
```

What matters, for now, is the `build` section that defines the `template` with the tag of the image we'll build in our pipelines. It consists of variables `DOCKER_REGISTRY` and `VERSION` whose values will be set by our pipeline at runtime.

Next, we have the `charts` directory that contains Helm definitions that will be used to deploy our application to Kubernetes. We won't go into much detail about Helm, but only the bits necessary to understand what Jenkins X does. If you never used Helm, I recommend consulting the [official documentation](#) or read [The DevOps 2.4 Toolkit: Continuous Deployment To Kubernetes](#) book I published previously. For now, I'll

only summarize it by stating that Helm is a package manager for Kubernetes.

Let's take a look at what's inside the *charts* folder.

```
1 ls -1 charts
```

The output is as follows.

```
1 jx-go
2 preview
```

There are two subdirectories. The one with the name of the application (`jx-go`) contains Helm definition of the application we'll deploy to different environments (e.g., staging, production). `preview`, on the other hand, is mostly used with pull requests. The reason for such separation lies in the ability to differentiate one from the other. We might need to customize `preview` with different variables or to add temporary dependencies. We'll explore the `preview` charts in more depth later. Right now, we'll focus on the `jx-go` chart.

```
1 ls -1 charts/jx-go
```

```
1 Chart.yaml
2 Makefile
3 README.md
4 charts
5 templates
6 values.yaml
```

If you used Helm, the structure should be familiar. If that's not the case, you might want to stop here and explore Helm in more detail. **The DevOps 2.4 Toolkit: Continuous Deployment To Kubernetes** book might be a good read if you have time, otherwise, check the official docs.

The last file in that directory is *jenkins-x.yml*.

```
1 cat jenkins-x.yml
```

Just as with the other files generated with Jenkins X quickstart, we'll go into more detail later. For now, think of it as a pipeline definition that points to the pipeline `go` defined in a different repository and used with all applications written in GoLang.

Jenkins X did not create only a Git project, but also a pipeline in the cluster, as well as GitHub webhook that will trigger it.

```
1 open "https://github.com/$GH_USER/jx-go/settings/hooks"
```

The screenshot shows a GitHub repository settings page for 'vfarcic/jx-go'. The 'Webhooks' tab is selected. A single webhook is listed with the URL 'http://hook.jx.34.206.148.101.nip.io/hook' and the event type '(all events)'. There are 'Edit' and 'Delete' buttons next to the entry.

**Figure 3-2: A GitHub webhook created as part of Jenkins X quickstart process)**

From now on, every time we push a change to the repository, that webhook will trigger a build in Jenkins X.

## Retrieving Jenkins X Activities, Logs, Pipelines, Applications, And Environments

While UIs are nice to look at, I am a firm believer that nothing beats command line concerning speed and repeatability. Fortunately, we can retrieve (almost) any information related to Jenkins X through `jx` executable. We can, for example, get the last activities (builds) of those jobs.

```
1 jx get activities
```

The output is as follows.

```
1 STEP
2 vfarcic/environment-jx-rocks-staging/master #1      STARTED AGO DURATION STATUS
3 meta pipeline                                         5m17s    59s Succeeded
4   Credential Initializer Cd47r                      5m17s    14s Succeeded
5   Working Dir Initializer N5vv7                     5m17s    0s Succeeded
6   Place Tools                                         5m16s    1s Succeeded
7   Git Source Meta Vfarcic Environment Jx Roc ...    5m15s    4s Succeeded
8 ps://github.com/vfarcic/environment-jx-rocks-staging.git
9   Git Merge                                           5m11s    1s Succeeded
10  Merge Pull Refs                                    5m10s    0s Succeeded
11  Create Effective Pipeline                         5m10s    2s Succeeded
```

```

12      Create Tekton Crds          5m8s    5s Succeeded
13  from build pack              5m2s    44s Succeeded
14  Credential Initializer Q4qvj  5m2s    0s Succeeded
15  Working Dir Initializer Gtj86 5m2s    1s Succeeded
16  Place Tools                  5m1s    1s Succeeded
17  Git Source Vfarcic Environment Jx Rocks St ...
18 ps://github.com/vfarcic/environment-jx-rocks-staging.git
19  Git Merge                      4m55s   1s Succeeded
20  Setup Jx Git Credentials       4m54s   1s Succeeded
21  Build Helm Apply              4m53s   35s Succeeded
22 vfarcic/environment-jx-rocks-staging/PR-1 #1
23  meta pipeline                 6m22s   47s Succeeded
24  Credential Initializer Jdbwq  6m22s   0s Succeeded
25  Working Dir Initializer T54b2 6m22s   0s Succeeded
26  Place Tools                  6m22s   1s Succeeded
27  Git Source Meta Vfarcic Environment Jx Roc ...
28 ps://github.com/vfarcic/environment-jx-rocks-staging.git
29  Git Merge                      6m16s   0s Succeeded
30  Merge Pull Refs              6m16s   1s Succeeded
31  Create Effective Pipeline     6m15s   3s Succeeded
32  Create Tekton Crds           6m12s   2s Succeeded
33  from build pack              6m7s    32s Succeeded
34  Credential Initializer Ntjdr  6m7s    0s Succeeded
35  Working Dir Initializer 86qgm 6m7s    1s Succeeded
36  Place Tools                  6m6s    1s Succeeded
37  Git Source Vfarcic Environment Jx Rocks St ...
38 ps://github.com/vfarcic/environment-jx-rocks-staging.git
39  Git Merge                      6m1s    1s Succeeded
40  Build Helm Build              6m0s    25s Succeeded
41 vfarcic/jx-go/master #1
42 sion: 0.0.1
43  meta pipeline                 7m24s   18s Succeeded
44  Credential Initializer J9wnj  7m24s   0s Succeeded
45  Working Dir Initializer Fs82g 7m24s   2s Succeeded
46  Place Tools                  7m22s   2s Succeeded
47  Git Source Meta Vfarcic Jx Go Master ...
48 ps://github.com/vfarcic/jx-go.git
49  Git Merge                      7m17s   1s Succeeded
50  Merge Pull Refs              7m16s   0s Succeeded
51  Create Effective Pipeline     7m16s   3s Succeeded
52  Create Tekton Crds           7m13s   7s Succeeded
53  from build pack              7m4s    2m6s Succeeded
54  Credential Initializer Fmc45  7m4s    0s Succeeded
55  Working Dir Initializer Vpjff 7m4s    3s Succeeded
56  Place Tools                  7m1s    2s Succeeded
57  Git Source Vfarcic Jx Go Master ...
58 ps://github.com/vfarcic/jx-go.git
59  Git Merge                      6m48s   1s Succeeded
60  Setup Jx Git Credentials       6m47s   0s Succeeded
61  Build Make Build              6m47s   6s Succeeded
62  Build Container Build         6m41s   2s Succeeded
63  Build Post Build              6m39s   1s Succeeded
64  Promote Changelog             6m38s   4s Succeeded
65  Promote Helm Release          6m34s   5s Succeeded
66  Promote Jx Promote             6m29s   1m31s Succeeded
67  Promote: staging               6m24s   1m26s Succeeded
68  PullRequest                   6m24s   1m26s Succeeded
69 llRequest: https://github.com/vfarcic/environment-jx-rocks-staging/pull/1 Merge S
70 ...
71  Update                         4m58s   0s Succeeded

```

We can see that there were activities with each of the three jobs. We had one deployment to the production environment (`environment-jx-rocks-`

production), and two deployments to staging (`environment-jx-rocks-staging`). The first build (activity) is always performed when a job is created. Initially, environments only contain a few applications necessary for their correct operation. The reason for the second build of the staging environment lies in the creation of the `jx-go` project. One of the steps in its pipeline is in charge of promoting a successful build to the staging environment automatically. When we explore `jenkins-x.yml` in more detail, you'll get a better understanding of the process, including promotions.

The last activity is of the `jx-go` pipeline. So far, we did not push any change to the repository, so we have only one build that was run when the job itself was generated through the quickstart process.

While listing the most recent activities is very useful since we have only a few pipelines, when their number grows, we'll need to be more specific. For example, we might want to retrieve only the activities related to the `jx-go` pipeline.

```
1 jx get activities --filter jx-go --watch
```

This time, the output is limited to all the activities related to `jx-go` which, in our case, is a single build of the `master` branch.

STEP	STARTED AGO	DURATION	STATUS	VERS
1 vfarhic/jx-go/master #1	9m53s	2m26s	Succeeded	Versic
2 0.0.1				
3 meta pipeline	9m53s	18s	Succeeded	
4 Credential Initializer J9wnj	9m53s	0s	Succeeded	
5 Working Dir Initializer Fs82g	9m53s	2s	Succeeded	
6 Place Tools	9m51s	2s	Succeeded	
7 Git Source Meta Vfarcic Jx Go Master ...	9m49s	3s	Succeeded	https:
8 ithub.com/vfarcic/jx-go.git				
9 Git Merge	9m46s	1s	Succeeded	
10 Merge Pull Refs	9m45s	0s	Succeeded	
11 Create Effective Pipeline	9m45s	3s	Succeeded	
12 Create Tekton Crds	9m42s	7s	Succeeded	
13 from build pack	9m33s	2m6s	Succeeded	
14 Credential Initializer Fmc45	9m33s	0s	Succeeded	
15 Working Dir Initializer Vpjff	9m33s	3s	Succeeded	
16 Place Tools	9m30s	2s	Succeeded	
17 Git Source Vfarcic Jx Go Master ...	9m28s	11s	Succeeded	https:
18 ithub.com/vfarcic/jx-go.git				
19 Git Merge	9m17s	1s	Succeeded	
20 Setup Jx Git Credentials	9m16s	0s	Succeeded	
21 Build Make Build	9m16s	6s	Succeeded	
22 Build Container Build	9m10s	2s	Succeeded	
23 Build Post Build	9m8s	1s	Succeeded	
24 Promote Changelog	9m7s	4s	Succeeded	
25 Promote Helm Release	9m3s	5s	Succeeded	
26 Promote Jx Promote	8m58s	1m31s	Succeeded	
27 Promote: staging	8m53s	1m26s	Succeeded	

```
29      PullRequest          8m53s    1m26s Succeeded PullR
30  est: https://github.com/vfarcic/environment-jx-rocks-staging/pull/1 Merge SHA: 07
31  31331ccf7460b32f54e979b8797ae4e55e7
32      Update               7m27s    0s Succeeded
```

This time, we used the `--watch` flag to tell Jenkins X that we'd like to *watch* the activities. Since there are no pending builds, the output will stay intact, so please press *ctrl+c* to stop the watch and return to the prompt.



Internally, Jenkins X activities are stored as Kubernetes Custom Resources (CRDs). If you're curious, you can see them by executing `kubectl --namespace jx get act`.

Activities provide only a high-level overview of what happened. When everything is successful, that is often all the information we need. However, when things go wrong (e.g., some of the tests fail), we might need to dig deeper into a build by retrieving the logs.

```
1 jx get build logs
```

Since we did not specify from which build we'd like to retrieve logs, we are faced with the prompt to select the pipeline from which we'd like to extract the output. We could choose one of the pipelines, but we won't do that since I want to show you that we can be more specific in our request for logs.

Please press *ctrl+c* to return to the prompt.

We can use `--filter` argument to retrieve logs from the last build of a specific pipeline.

```
1 jx get build logs --filter jx-go
```

The output should show the logs of the last build of *jx-go*, no matter the branch.

We can be even more specific than that and request logs from the specific GitHub user, of the specific pipeline, from the last build of a specific branch.

```
1 jx get build logs \
2   --filter ${GH_USER}/jx-go/master
```

The output should show the logs of the last build of the *jx-go* pipeline initiated by a commit to the *master* branch.

Being able to retrieve logs from a specific pipeline is not of much use if we do not even know which pipelines we have. Fortunately, we can extract the list of all the pipelines as well.

```
1 jx get pipelines
```

We can see that there are three pipelines named *environment-jx-rocks-production*, *environment-jx-rocks-staging*, and *jx-go* (I'll ignore the existence of the dummy pipeline). The first two are in charge of deploying applications to staging and production environments. Since we are using Kubernetes, those environments are separate namespaces. We'll discuss those two later. The third job is related to the *jx-go* project we created as a quickstart.

Similarly, we can also retrieve the list of applications currently managed by Jenkins X.

```
1 jx get applications
```

The output is as follows.

```
1 APPLICATION STAGING PODS URL
2 jx-go      0.0.1   1/1   http://jx-go.jx-staging.34.206.148.101.nip.io
```

For now, retrieving the applications is uneventful since we have only one deployed to the staging environment.

So far, we talked about the staging and the production environments. Are those the only ones we have? Let's check it out.

```
1 jx get env
```

The output is as follows.

1 NAME	2 LABEL	3 KIND	4 PROMOTE	5 NAMESPACE	6 ORDER	7 CLUSTER	8 SOURCE
					REF	PR	
3 dev	Development	Development	Never	jx	0		
4 staging	Staging	Permanent	Auto	jx-staging	100		https://git
5 b.com/vfarcic/environment-jx-rocks-staging.git							
6 production	Production	Permanent	Manual	jx-production	200		https://git
7 b.com/vfarcic/environment-jx-rocks-production.git							

As you can see, there is the third environment named `dev`. We'll explore it later. For now, remember that its purpose is true to its name. It is meant to facilitate development.

Now that we know which environments we have, we can combine that information and list only the applications in one of them. Let's see which ones are running in the staging environment.

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

The output is as follows.

```
1 APPLICATION STAGING PODS URL
2 jx-go      0.0.1   1/1   http://jx-go.jx-staging.34.206.148.101.nip.io
```

We already knew from before that the *jx-go* application is running in staging and we already know that nothing is installed in production. Nevertheless, we can confirm that with the command that follows.

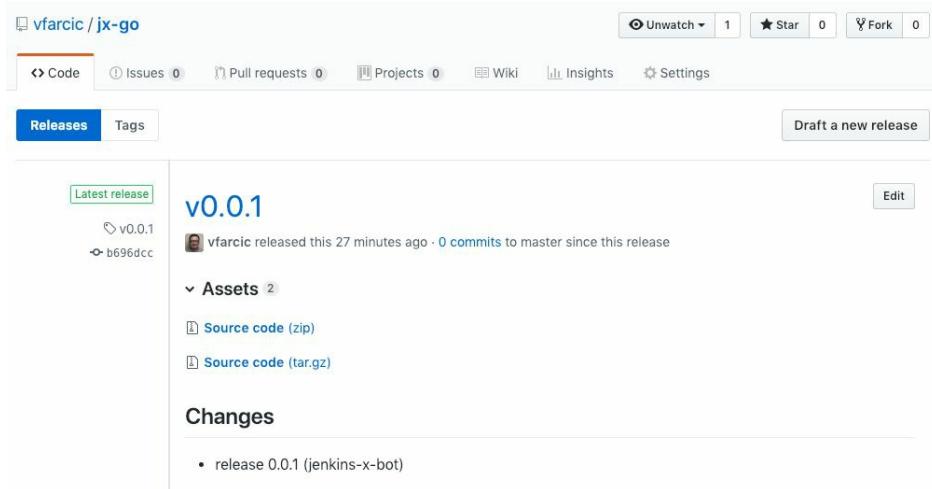
```
1 jx get applications --env production
```

It should come as no surprise that the output states that no applications were found in environments production. We did not promote anything to production yet. We'll do that later.

Finally, Jenkins X also created a GitHub release for us. We can confirm that by going to project releases.

```
1 open "https://github.com/$GH_USER/jx-go/releases"
```

For now, we have only one release that is not very descriptive, since we did not create any issues that should be listed in release notes. The release you see in front of you is only the initial one created by pushing the quickstart files to Git.



**Figure 3-4: The first release of the quickstart project)**

Finally, we did not yet confirm whether the new application is indeed deployed and can be accessed.

```

1 ADDR=$(kubectl --namespace jx-staging \
2     get ingress jx-go \
3     -o jsonpath=".spec.rules[0].host")
4
5 curl "http://$ADDR"

```

We retrieved the host from `jx-go` Ingress and used it to send a `curl` request. As a result, the output is `Hello from: Jenkins X golang http example`. We confirmed that the application created as a quickstart was deployed to the staging environment. All we did was execute `jx create quickstart` and Jenkins X did most of the heavy lifting for us.

## What Now?

I hope that your head is not spinning too much after this introduction to quickstart projects and Jenkins X in general. The goal was not to present you with details nor to explain the process and the tools in depth. I wanted you to get the general feeling of how Jenkins X works before we dive deeper into details.

I find it challenging to understand detailed concepts without having at least a high-level practical introduction to how things work. Now that we saw Jenkins X in action, we can dive into more specific topics. The next one we'll explore is the way to import existing projects into Jenkins X. Unless your company was created yesterday, it is almost sure that you already

have some code repositories that you might want to move over to Jenkins X.

All in all, this was a very quick introduction, and the real fun is coming next.

Now is a good time for you to take a break.

Please go out (back) of the `jx-go` directory.

```
1 cd ..
```

If you created a cluster only for the purpose of the exercises we executed, please destroy it. We'll start the next, and each other chapter from scratch as a way to save you from running your cluster longer than necessary and pay more than needed to your hosting vendor. If you created the cluster or installed Jenkins X using one of the Gists from the beginning of this chapter, you'll find the instructions on how to destroy the cluster or uninstall everything at the bottom.

If you did choose to destroy the cluster or to uninstall Jenkins X, please remove the repositories we created as well as the local files. You can use the commands that follow for that.

```
1 hub delete -y \
2   $GH_USER/environment-jx-rocks-staging
3
4 hub delete -y \
5   $GH_USER/environment-jx-rocks-production
6
7 hub delete -y $GH_USER/jx-go
8
9 rm -rf ~/.jx/environments/$GH_USER/environment-jx-rocks-*
10
11 rm -rf jx-go
```

Finally, you might be planning to move into the next chapter right away. If that's the case, there are no cleanup actions to do. Just keep reading.

# Importing Existing Projects Into Jenkins X

We saw how we can fast-track development and continuous delivery of new applications with Jenkins X quickstarts. However, it is likely that your company was not formed yesterday. That means that you already have some apps and hopefully you'd like to move them to Jenkins X.

From a Jenkins X perspective, importing an existing project is relatively straightforward. All we have to do is execute `jx import`, and Jenkins X will do its magic. It will create the files we need. If we do not yet have `skaffold.yml`, it'll generate it for us. If we did not create a Helm chart, it would create that as well. No Dockerfile? No problem. We'll get that as well. Never wrote Jenkins pipeline for that project? Again, that is not an issue. We'll get `jenkins-x.yml`. Jenkins X will reuse the things we already have, and create those that we're missing.

The import process does not limit itself to creating missing files and pushing them to Git. It'll also create a job in Jenkins, webhooks in GitHub, and quite a few other things.

Nevertheless, we'll go beyond a simple import process since you're bound to have to tweak the results. Maybe your application needs a database, and the chart that will be created through Jenkins X does not include it. Or, maybe your application has a different path which should be used for Kubernetes health checks. There are many things that your application requires, and not all of them will be detected by the Jenkins X import process.

All in all, we'll go through the exercise of importing a project and trying to figure out whether we are missing something specific to our application. If we are, we'll fix it. Once we understand which application-specific things are missing, we'll create a new build pack so that the next time we import an application based on similar technology, the process will be streamlined and consist of a single command. On top of that, we'll be able to use that same build pack for any new application we might start developing in the future since it'll be available as yet another quickstart option.

For now, we'll focus on importing a project and solving potential problems we might encounter. Later on, in the next chapter, we'll dive into build packs and try to create our own based on the experience we'll obtain by importing an existing application.

## Creating A Kubernetes Cluster With Jenkins X

We'll start from the beginning. We need a Kubernetes cluster with Jenkins X up-and-running. You can continue using the cluster from the previous chapter if you did not destroy it. Otherwise, you'll need to create a new cluster or install Jenkins X if you already have one.



All the commands from this chapter are available in the [04-import.sh](#) Gist.

For your convenience, the Gists from the previous chapter are available below as well.

- Create a new serverless **GKE** cluster: [gke-jx-serverless.sh](#)
- Create a new serverless **EKS** cluster: [eks-jx-serverless.sh](#)
- Create a new serverless **AKS** cluster: [aks-jx-serverless.sh](#)
- Use an **existing** serverless cluster: [install-serverless.sh](#)

Without further ado, we are about to import a project into Jenkins X.

## Importing A Project

We'll import the application stored in the [vfarcic/go-demo-6](#) repository. We'll use it as a guinea pig for testing the import process as well as to flesh out potential problems we might encounter.

But, before we import the repository, you'll have to fork the code. Otherwise, you won't be able to push changes since you are not (yet) a collaborator on that specific repository.

1 open "<https://github.com/vfarcic/go-demo-6>"

Please make sure that you are logged in and click the *Fork* button located in the top-right corner. Follow the on-screen instructions.

Next, we need to clone the repository you just forked.



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

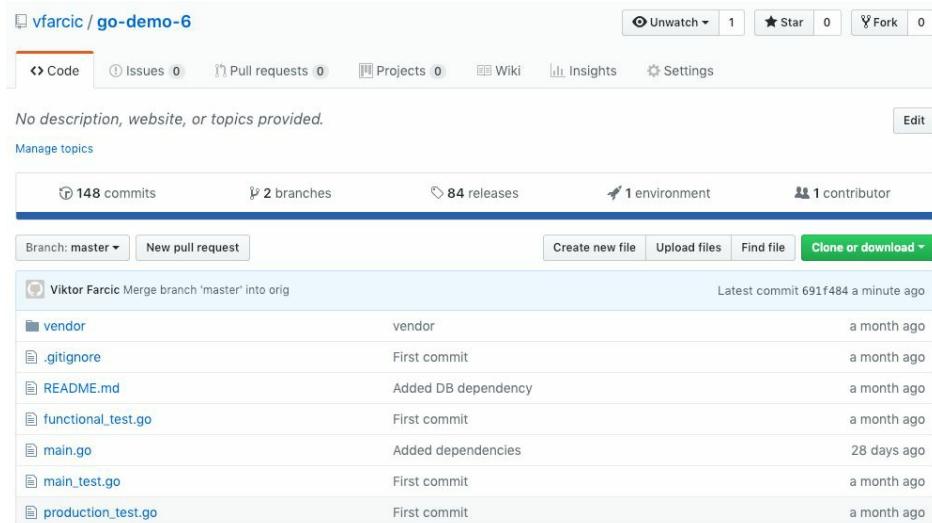
```
1 GH_USER=[...]
2
3 git clone \
4   https://github.com/$GH_USER/go-demo-6.git
5
6 cd go-demo-6
```

The chances are that I did not leave the master branch in the state I intended it to be or that I'm doing something with it while you're reading this. I might be testing some recently released Jenkins X feature, or maybe I'm trying to fix a problem from one of the examples. To be sure that you're having the correct version of the code, we'll replace the `master` branch with the `orig` branch, and push it back to GitHub.

```
1 git pull
2
3 git checkout orig
4
5 git merge -s ours master --no-edit
6
7 git checkout master
8
9 git merge orig
10
11 rm -rf charts
12
13 git push
```

Now you should have the intended code in the master branch of the repository you forked. Feel free to take a look at what we have by opening the repository in a browser. Fortunately, there is a `jx` command that does just that.

```
1 jx repo --batch-mode
```



**Figure 4-1: Application repository with only the code**

Let's us quickly explore the files of the project, before we import it into Jenkins X.

```
1 ls -1
```

The output is as follows.

```
1 README.md
2 functional_test.go
3 main.go
4 main_test.go
5 production_test.go
6 vendor
```

As you can see, there's (almost) nothing in that repository but Go code (\*.go) and libraries (vendor).

That project is one extreme of the possible spectrum of projects we might want to import to Jenkins X. It only has the code of the application. There is no Dockerfile, and there is no Helm chart. Heck, there is not even a script for building a binary, nor there is a mechanism to run tests, and there is definitely no jenkins-x.yml that defines a continuous delivery pipeline for the application. There's only code, and (almost) nothing else.

Such a situation might not be your case. Maybe you do have scripts for running tests or building the code. Or perhaps you do have Dockerfile, and maybe you are already a heavy Kubernetes user, and you do have a Helm chart. You might have other files as well. We'll discuss those situations

later. For now, we'll work on the case when there is nothing but a code of an application.

Let's see what happens when we try to import that repository into Jenkins X.

```
1 jx import --batch-mode
```

The output is as follows.

```
1 No username defined for the current Git server!
2 performing pack detection in folder /Users/vfarcic/code/go-demo-6
3 --> Draft detected Go (100.000000%)
4 selected pack: /Users/vfarcic/.jx/draft/packs/github.com/jenkins-x-buildpacks/jen
5 s-x-kubernetes/packs/go
6 replacing placeholders in directory /Users/vfarcic/code/go-demo-6
7 app name: go-demo-6, git server: github.com, org: vfarcic, Docker registry org: v
8 cic
9 skipping directory "/Users/vfarcic/code/go-demo-6/.git"
10 Created Jenkins Project: http://jenkins.jx.35.229.111.85.nip.io/job/vfarcic/job/g
11 ome-6/
12
13 Watch pipeline activity via:      jx get activity -f go-demo-6 -w
14 Browse the pipeline log via:     jx get build logs vfarcic/go-demo-6/master
15 Open the Jenkins console via:    jx console
16 You can list the pipelines via: jx get pipelines
17 When the pipeline is complete:   jx get applications
18
19 For more help on available commands see: https://jenkins-x.io/developing/browsing
20
21 Note that your first pipeline may take a few minutes to start while the necessary
22 ages get downloaded!
23
24 Creating GitHub webhook for vfarcic/go-demo-6 for url http://jenkins.jx.35.229.11
25 5.nip.io/github-webhook/
```

We can see from the output that Jenkins X detected that the project is 100.000000% written in Go, so it selected the `go` build pack. It applied it to the local repository and pushed the changes to GitHub. Further on, it created a Jenkins project as well as a GitHub webhook that will trigger builds whenever we push changes to one of the selected branches. Those branches are by default `master`, `develop`, `PR-.*`, and `feature.*`. We could have changed the pattern by adding the `--branches` flag. But, for our purposes, and many others, those branches are just what we need.

Now, let's take another look at the files in the local copy of the repository.

```
1 ls -l
```

The output is as follows.

```
1 Dockerfile
2 Makefile
3 OWNERS
4 OWNERS_ALIASES
5 README.md
6 charts
7 functional_test.go
8 go.mod
9 jenkins-x.yml
10 main.go
11 main_test.go
12 production_test.go
13 skaffold.yaml
14 vendor
15 watch.sh
```

We can see that quite a few new files were added to the project through the import process. We got `Dockerfile` that will be used to build container images and we got `jenkins-x.yml` that defines all the steps of our pipeline.

Further on, `Makefile` is new as well. It, among others, defines targets to build, test, and install the application. Then, there is now the `charts` directory that contains files in Helm format. We'll use it to package, install, and upgrade our application. Then, there is `skaffold.yaml` that contains instructions on how to build container images. Finally, we got `watch.sh`. It allows us to compile the binary and create a container image every time we change a file in that project. It is a handy script for local development. There are a few other new files (e.g., `OWNERS`) added to the mix.

Do not think that is the only explanation you'll get about those files. We'll explore them in much more detail later in one of the follow-up chapters. For now, what matters is that we imported our project into Jenkins X and that it should contain everything the project needs both for local development as well as for continuous delivery pipeline.

Now that the project is in Jenkins X, we should see it as one of the activities and observe the first build in action. You already know that we can limit the retrieval of Jenkins X activities to a specific project and that we can use `--watch` to watch the progress.

```
1 jx get activities \
2   --filter go-demo-6 \
3   --watch
```

By the time the build is finished, the output, without the entries repeated due to changes in statuses, should be as follows.

```

1 STEP
2 vfarcic/go-demo-6/master #1
3 on: 1.0.420
4 meta pipeline
5 Credential Initializer Dkm8m
6 Working Dir Initializer Gbv2k
7 Place Tools
8 Git Source Meta Vfarcic Go Demo 6 Master ...
9 ://github.com/vfarcic/go-demo-6.git
10 Git Merge
11 Merge Pull Refs
12 Create Effective Pipeline
13 Create Tekton Crds
14 from build pack
15 Credential Initializer D5fc9
16 Working Dir Initializer 7rf9g7
17 Place Tools
18 Git Source Vfarcic Go Demo 6 Master ...
19 ://github.com/vfarcic/go-demo-6.git
20 Git Merge
21 Setup Jx Git Credentials
22 Build Make Build
23 Build Container Build
24 Build Post Build
25 Promote Changelog
26 Promote Helm Release
27 Promote Jx Promote
28 Promote: staging
29 PullRequest
30 Request: https://github.com/vfarcic/environment-jx-rocks-staging/pull/2 Merge SHA
31 ..
32 Update

```

	STARTED	AGO	DURATION	STATUS
3	3m46s	3m6s	Succeeded	V
4	3m46s	30s	Succeeded	
5	3m46s	0s	Succeeded	
6	3m46s	1s	Succeeded	
7	3m45s	1s	Succeeded	
8	3m44s	15s	Succeeded	ht
9	3m29s	1s	Succeeded	
10	3m28s	1s	Succeeded	
11	3m27s	2s	Succeeded	
12	3m25s	9s	Succeeded	
13	3m15s	2m35s	Succeeded	
14	3m15s	0s	Succeeded	
15	3m15s	1s	Succeeded	
16	3m14s	1s	Succeeded	
17	3m13s	25s	Succeeded	ht
18	2m48s	1s	Succeeded	
19	2m47s	0s	Succeeded	
20	2m47s	20s	Succeeded	
21	2m27s	4s	Succeeded	
22	2m23s	1s	Succeeded	
23	2m22s	6s	Succeeded	
24	2m16s	5s	Succeeded	
25	2m11s	1m31s	Succeeded	
26	2m6s	1m26s	Succeeded	
27	2m6s	1m26s	Succeeded	F
28				
29				
30				
31				
32	40s	0s	Succeeded	

Please stop watching the activities by pressing *ctrl+c*.

So far, the end result looks similar to the one we got when we created a quickstart. Jenkins X created the files it needs, it created a GitHub webhook, it created a pipeline, and it pushed changes to GitHub. As a result, we got our first build and, by the look of it, it was successful.

Since I have a paranoid nature, we'll double check whether everything indeed looks ok.

Please open the `PullRequest` link from the activity output.

So far so good. The *go-demo-6* job created a pull request to the *environment-jx-rocks-staging* repository. As a result, the webhook from that repository should have initiated a pipeline activity, and the result should be a new release of the application in the staging environment. We won't go through that part of the process just yet. For now, just note that the application should be running, and we'll check that soon.

The information we need to confirm that the application is indeed running is in the list of the applications running in the staging environment. We'll explore the environments later. For now, just run the command that follows.

```
1 jx get applications
```

The output is as follows.

```
1 APPLICATION STAGING PODS URL
2 go-demo-6 1.0.420 http://go-demo-6.jx-staging.34.206.148.101.nip.io
```

We can see the address through which our application should be accessible in the URL column. Please copy it and use it instead of [...] in the command that follows.

```
1 STAGING_ADDR=[...]
2
3 curl "$STAGING_ADDR/demo/hello"
```

The output is as follows.

```
1 <html>
2 <head><title>503 Service Temporarily Unavailable</title></head>
3 <body>
4 <center><h1>503 Service Temporarily Unavailable</h1></center>
5 <hr><center>nginx/1.15.6</center>
6 </body>
7 </html>
```

Now that was unexpected. Everything looks ok from Jenkins X perspective, but the application is not accessible. Did we fail to do something, or did Jenkins X fail to do the right thing?

When in doubt, we can always take a look at the logs.

```
1 kubectl --namespace jx-staging logs \
2     -l app=go-demo-6
```

The output is as follows.

```
1 2019/01/24 23:16:52 Starting the application
2 2019/01/24 23:16:52 Configuring DB localhost
3 panic: no reachable servers
4 ...
```

The problem is in the database. To be more precise, it's missing. The application tried to connect to MongoDB, and it couldn't find it.

In retrospect, that makes sense. While Jenkins X does the right thing most of the time, it could not know that we need a database and especially that we need MongoDB. There is no such information in the repository, except inside the Go code. Excluding the possibility of scanning the whole code and figuring out that MongoDB is needed based on imported libraries, it's normal that Jenkins X did not add it to the Helm chart it generated when we imported the project.

The only sensible thing we can do is to modify the Helm chart and add YAML files Kubernetes will need to spin up a database together with the application.



Please note that we'll explore how to fix the issues that occurred from importing a specific application into Jenkins X. Your problems will likely be different. Nevertheless, I believe that the exercises that follow will give you an insight into what to expect when importing projects as well as to watch out for auto-generated Helm charts as being the most common culprit.

## Fixing The Auto-Generated Helm Chart

Even though the code of the application is small, I will save you from going through it, especially since you might not be proficient with Go. Instead, I'll let you know right away what's missing and which parts of the chart need to be added or modified.

First of all, the application requires an environment variable called `DB`. The code is using it to obtain the address of the database. That brings us to the first thing missing in the chart generated by Jenkins X. There is no definition for MongoDB.

So, the first step is to open `charts/go-demo-6/templates/deployment.yaml` in your favorite editor. That's where the Deployment for the application is defined, and that's where we need to add the variable.

Please locate the code that follows.

```
1 ...
2     imagePullPolicy: {{ .Values.image.pullPolicy }}
3     env:
4 ...
```

Now, add the `env` section with the `name` set to `DB` and the value `{{ template "fullname" . }}-db`. The final version of the snippet listed above should be as follows.

```
1 ...
2     imagePullPolicy: {{ .Values.image.pullPolicy }}
3     env:
4         - name: DB
5             value: {{ template "fullname" . }}-db
6 ...
```

Save the file.

Next, we need to add MongoDB to the Helm chart that was created for us. Now, we could start writing Helm templates for MongoDB StatefulSet and a Service. We could spend time trying to figure out how to replicate data between its replicas, and probably a few other things that might not be obvious from the start. However, we should know better. We should know that there is already a Helm chart that does just that, and much more. There are quite a few charts we could use, but we'll go with [mongodb from the stable channel](#).

So, how can we add MongoDB chart to the one we already have? The answer in `dependencies`. We can make our application depend on MongoDB chart by creating a `requirements.yaml` file.

```
1 echo "dependencies:
2   - name: mongodb
3     alias: go-demo-6-db
4     version: 5.3.0
5     repository: https://kubernetes-charts.storage.googleapis.com
6     condition: db.enabled
7   " | tee charts/go-demo-6/requirements.yaml
```

The only thing worth noting in that file are `alias` and `condition`. The former (`alias`) is set to the value that will create a Service with the same name as the environment variable `DB` that we just added to `deployment.yaml`. The latter (`condition`) will allow us to disable this dependency. We'll see later why we might want to do that.

There's only one more thing missing. We should probably customize the MongoDB chart to fit our use case. I won't go through all the values we could set. You can explore them yourself by executing `helm inspect values stable/mongodb` or by visiting [project README](#). Instead, we'll define only one, mostly as an exercise how to define values for the dependencies.

So, let's add a few new entries to `values.yaml`.

```
1 echo "go-demo-6-db:
2   replicaSet:
3     enabled: true
4 " | tee -a charts/go-demo-6/values.yaml
```

We set only `replicaSet.enabled` to `true`. The important thing to note is that it is nested inside `go-demo-6-db`. That way, Helm will know that the variable is not meant for our application (*go-demo-6*), but for the dependency called (aliased) `go-demo-6-db`.

Now, that we learned how to add dependencies to our applications, we should push the changes to GitHub, and check whether that solved our issue.

```
1 git add .
2
3 git commit \
4   --message "Added dependencies"
5
6 git push
```

Next, we need to wait until the new release is deployed to the staging environment. We'll monitor the activity of the new build and wait until its finished.

```
1 jx get activity \
2   --filter go-demo-6 \
3   --watch
```

The output, limited to the new build, is as follows:

STEP	STARTED AGO	DURATION	STATUS	VARS
1 STEP				
2 ...				
3 vfarcic/go-demo-6/master #2	2m51s	2m43s	Succeeded	Vars
4 : 1.0.421				
5 meta pipeline	2m51s	20s	Succeeded	
6 Credential Initializer Kh72n	2m51s	0s	Succeeded	
7 Working Dir Initializer Nlnj2	2m51s	1s	Succeeded	
8 Place Tools	2m50s	1s	Succeeded	
9 Git Source Meta Vfarcic Go Demo 6 Master R ...	2m49s	4s	Succeeded	http
10 /github.com/vfarcic/go-demo-6.git				
11 Git Merge	2m45s	1s	Succeeded	
12 Merge Pull Refs	2m44s	0s	Succeeded	
13 Create Effective Pipeline	2m44s	2s	Succeeded	
14 Create Tekton Crds	2m42s	11s	Succeeded	
15 from build pack	2m30s	2m22s	Succeeded	
16 Credential Initializer Jk7k5	2m30s	0s	Succeeded	
17 Working Dir Initializer 4vrhr	2m30s	1s	Succeeded	
18 Place Tools	2m29s	1s	Succeeded	
19 Git Source Vfarcic Go Demo 6 Master Releas ...	2m28s	4s	Succeeded	http
20 /github.com/vfarcic/go-demo-6.git				
21 Git Merge	2m24s	1s	Succeeded	

```

22      Setup Jx Git Credentials          2m23s      0s Succeeded
23      Build Make Build                2m23s      20s Succeeded
24      Build Container Build          2m3s       3s Succeeded
25      Build Post Build               2m0s       1s Succeeded
26      Promote Changelog              1m59s      6s Succeeded
27      Promote Helm Release           1m53s      14s Succeeded
28      Promote Jx Promote             1m39s      1m31s Succeeded
29      Promote: staging               1m34s      1m26s Succeeded
30      PullRequest                   1m34s      1m25s Succeeded Pul
31 quest: https://github.com/vfarcic/environment-jx-rocks-staging/pull/3 Merge SHA:
32      Update                         9s        1s Succeeded
33      Promoted                       9s        1s Succeeded App
34 action is at: http://go-demo-6.jx-staging.34.206.148.101.nip.io

```

The `Promoted` step is the last one in that build. Once we reach it, we can stop monitoring the activity by pressing `ctrl+c`.

Let's see whether we got the Pods that belong to the database and, more importantly, whether the application is indeed running.

```
1 kubectl --namespace jx-staging get pods
```

The output is as follows.

1 NAME	READY	STATUS	RESTARTS	AGE
2 jx-go-demo-6-...	0/1	Running	5	5m
3 jx-go-demo-6-db-arbiter-0	1/1	Running	0	5m
4 jx-go-demo-6-db-primary-0	1/1	Running	0	5m
5 jx-go-demo-6-db-secondary-0	1/1	Running	0	5m



Please note that it might take a minute or two after the application pipeline activity is finished for the application to be deployed to the staging environment. If the output listed below does not match what you see on the screen, you might need to wait for a few moments and re-run the previous command.

The good news is that the database is indeed running. The bad news is that the application is still not operational. In my case, it already restarted five times, and 0 containers are available.

Given that the problem is this time probably not related to the database, the logical course of action is to describe the Pod and see whether we can get a clue about the issue from the events.

```

1 kubectl --namespace jx-staging \
2   describe pod \
3     -l app=jx-go-demo-6

```

The output, limited to the message of the events, is as follows.

```
1 ...
2 Events:
3 ... Message
4 ...
5 ... Successfully assigned jx-go-demo-6-fdd8f6644-xx68f to gke-jx-rocks-default-pc
6 119fec1e-v7p7
7 ... MountVolume.SetUp succeeded for volume "default-token-cxn5p"
8 ... Readiness probe failed: Get http://10.28.2.17:8080/: dial tcp 10.28.2.17:8080
9 etsockopt: connection refused
10 ... Back-off restarting failed container
11 ... Container image "10.31.245.243:5000/vfarcic/go-demo-6:0.0.81" already present
12 machine
13 ... Created container
14 ... Started container
15 ... Liveness probe failed: HTTP probe failed with statuscode: 404
16 ... Readiness probe failed: HTTP probe failed with statuscode: 404
```

This time, liveness and readiness probes are failing. Either the application inside that Pod is not responding to the probes, or there's some other problem we did not yet discover.

Now, unless you went through the code, you cannot know that the application does not respond to requests on the root path. If we take a look at the Pod definition, we'll see that `probePath` is set to `/`. Jenkins X could not know which path could be used for the probe of our application. So, it set it to the only sensible default it could, and that's `/`.

We'll have to modify the `probePath` entry. There is already a variable that allows us to do that instead of fiddling with the Deployment template.

```
1 cat charts/go-demo-6/values.yaml
```

If you go through the values, you'll notice that one of them is `probePath` and that it is set to `/`.

Please edit the `charts/go-demo-6/values.yaml` file by changing `probePath: /` entry to `probePath: /demo/hello?health=true`. Feel free to use your favorite editor for that and make sure to save the changes once you're done. Next, we'll push the changes to GitHub.

```
1 git add .
2
3 git commit \
4   --message "Added dependencies"
5
6 git push
```

Now it's another round of waiting until the activity of the new build is finished.

```
1 jx get activity \
2   --filter go-demo-6 \
3   --watch
```

Once the new build is finished, we can stop watching the activity by pressing *ctrl+c*. You'll know its done when you see the `Promoted` entry in the `Succeeded` status or simply when there are no `Pending` and `Running` steps.

What do you think? Is our application finally up-and-running? Let's check it out.

```
1 kubectl --namespace jx-staging get pods
```

The output is as follows.

1 NAME	READY	STATUS	RESTARTS	AGE
2 jx-go-demo-6-...	1/1	Running 0	39s	
3 jx-go-demo-6-db-arbiter-0	1/1	Running 0	11m	
4 jx-go-demo-6-db-primary-0	1/1	Running 0	11m	
5 jx-go-demo-6-db-secondary-0	1/1	Running 0	11m	

To be on the safe side, we'll send a request to the application. If we are greeted back, we'll know that it's working as expected.

```
1 curl "$STAGING_ADDR/demo/hello"
```

The output shows `hello, world`, thus confirming that the application is up-and-running and that we can reach it.

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

```
1 cd ..
```

## Why Did We Do All That?

You might not be using MongoDB, and your application might be responding to `/` for health checks. If that's the case, you might be wondering why did I take you through the exercise of fixing those things in `go-demo-6`. Was that useful to you?

Here's the secret. Jenkins X does not do magic. It does some very clever work in the background, and it helps a lot by simplifying things, by

installing and configuring tools, and by joining them all into a single easy to use process. It does a lot of heavy lifting for us, and a part of that is by importing projects into its own brace.

When we import a project, Jenkins X creates a lot of things. They are not a blind copy of files from a repository. Instead, it creates solutions tailored to our needs. The real question is how does Jenkins X know what we need? The answer is hugely related to our work. It evaluates what we did so far by scanning files in a repository where our application resides. In our example, it figured out that we are using Go, and it created a bunch of files that provide the definitions for various tools (e.g., Helm, Skaffold, Docker, etc.). Yet, Jenkins X could not read our mind. It failed to figure out that we needed a MongoDB and it did not know that our application requires a “special” address for Kubernetes health checks. We had to reconfigure some of the files created through the import process.

The real question is whether a person who imports a project has the skills to discover the cause of an issue created by missing entries in configuration files generated during the import process. If the answer is “yes, he knows how to find out what’s missing in Kubernetes definitions”, “yes, he uses Helm”, and many other yeses, then that person should not have a hard time figuring out things similar to those we did together. Nevertheless, Jenkins X is not made to serve only people who are Kubernetes ninjas, and who know Helm, Skaffold, Docker, Knative Build, and other amazing tools Jenkins X packages. It is designed to ease the pain caused by the complexity of today’s solutions. As such, it is silly to expect that everyone will go through the same process as we did. Someone often must, but the end result should not be what we just did. Instead, the solution lies in the creation of build packs. We’ll explore them in the next chapter. We’ll try to create a solution that will allow anyone with an application written in Go and dependant on MongoDB to import it into Jenkins X and expect everything to work from the first attempt.

The application we imported is on one extreme of the spectrum. It had nothing but Go code. There was no Dockerfile, no jenkins-x.yml, no Makefile, no skaffold.yaml, and certainly no Helm chart. Your applications might not be in such a state. Maybe you already created a Dockerfile, but you’re missing the rest. Or, perhaps you have all those except skaffold.yaml.

When your project has some of the files, but not all, the import process

will generate only those that are missing. It does not matter much what you do and what you don't have. Jenkins X will fill in the missing pieces.

## What Now?

Now is a good time for you to take a break.

You might be planning to move into the next chapter right away. If that's the case, there are no cleanup actions to do. Just continue on to the next chapter.

However, if you created a cluster only for the purpose of the exercises we executed, please destroy it. We'll start the next, and each other chapter from scratch as a way to save you from running your cluster longer than necessary and pay more than needed to your hosting vendor. If you created the cluster or installed Jenkins X using one of the Gists from the beginning of this chapter, you'll find the instructions on how to destroy the cluster or uninstall everything at the bottom.

If you did choose to destroy the cluster or to uninstall Jenkins X, please remove the repositories we created as well as the local files. You can use the commands that follow for that. Just remember to replace [...] with your GitHub user.

```
1 hub delete -y \
2   $GH_USER/environment-jx-rocks-staging
3
4 hub delete -y \
5   $GH_USER/environment-jx-rocks-production
6
7 rm -rf ~/.jx/environments/$GH_USER/environment-jx-rocks-*
```

# Creating Custom Build Packs

I stand by my claim that “you do not need to understand Kubernetes to **use Jenkins X.**” To be more precise, those who do not want to know Kubernetes and its ecosystem in detail can benefit from Jenkins X ability to simplify the processes around software development lifecycle. That’s the promise or, at least, one of the driving ideas behind the project. Nevertheless, for that goal to reach as wide of an audience as possible, we need a variety of build packs. The more we have, the more use cases can be covered with a single `jx import` or `jx quickstart` command. The problem is that there is an infinite number of types of applications and combinations we might have. Not all can be covered with community-based packs. No matter how much effort the community puts into creating build packs, they will always be a fraction of what we might need. That’s where you come in.

The fact that Jenkins X build packs cannot fully cover all our use cases does not mean that we shouldn’t work on reducing the gap. Some of our applications will have a perfect match with one of the build packs. Others will require only slight modifications. Still, no matter how many packs we create, there will always be a case when the gap between what we need and what build packs offer is more significant.

Given the diversity in languages and frameworks we use to develop our applications, it is hard to avoid the need for understanding how to create new build packs. Those who want to expand the available build packs need to know at least basics behind Helm and a few other tools. Still, that does not mean that everyone needs to possess that knowledge. What we do not want is for different teams to reinvent the wheel and we do not wish for every developer to spend endless hours trying to figure out all the details behind the ever-growing Kubernetes ecosystem. It would be a waste of time for everyone to go through steps of importing their applications into Jenkins X, only to discover that they need to perform a set of changes to adapt the result to their own needs.

Our next mission is to streamline the process of importing projects for all those teams in charge of applications that share some common design

choices and yet do not have a build pack that matches all their needs.

We'll explore how to create a custom build pack that could be highly beneficial for a (fictional) company. We'll imagine that there are multiple applications written in Go that depend on MongoDB and that there is a high probability that new ones based on the same stack will be created in the future. We'll explore how to create such a build pack with the least possible effort.

We'll need to make a decision what should be included in the build pack, and what should be left to developers (owners of an application) to add after importing their applications or after creating new ones through Jenkins X quickstart. Finally, we'll need to brainstorm whether the result of our work might be useful to others outside of our organization, or whether what we did is helpful only to our teams. If we conclude that the fruits of our work are useful to the community, we should contribute back by creating a pull request.

To do all that, we'll continue using the *go-demo-6* application since we are already familiar with it and since we already went through the exercise of discovering which changes are required to the `go` build pack.

## Creating A Kubernetes Cluster With Jenkins X

As in the previous chapters, we'll need a cluster with Jenkins X up-and-running. That means that you can continue using the cluster from the previous chapter if you did not destroy it. Otherwise, you'll need to create a new cluster or install Jenkins X if you already have one.



All the commands from this chapter are available in the [05-buildpacks.sh](#) Gist.

For your convenience, the Gists from the previous chapter are available below as well.

- Create a new serverless **GKE** cluster: [gke-jx-serverless.sh](#)
- Create a new serverless **EKS** cluster: [eks-jx-serverless.sh](#)
- Create a new serverless **AKS** cluster: [aks-jx-serverless.sh](#)
- Use an **existing** serverless cluster: [install-serverless.sh](#)

Let's get started.

## Choosing What To Include In A Build Pack

We might be tempted to create build packs that contemplate all the variations present in our applications. That is often a bad idea. Build packs should provide everything we need, within reason. Or, to be more precise, they should provide the things that are repeatable across projects, but they should not contemplate so many combinations that build packs themselves would become hard to maintain, and complicated to use. Simplicity is the key, without sacrificing fulfillment of our needs. When in doubt, it is often better to create a new build pack than to extend an existing one by adding endless `if/else` statements.

If we take the `go-demo-6` application as an example, we can assume that other projects are written in Go and that use MongoDB. Even if that's not the case right now, that is such a common combination that we can guess that someone will create a similar application in the future. Even if that is not true within our organization, surely there are many other teams doing something similar. The popularity of Go is on the constant rise, and MongoDB is one of the most popular databases. There must be many using that combination. All in all, a build pack for an application written in Go and with MongoDB as a backend is potentially an excellent candidate both for the internal use within our organization, as well as a contribution to the Jenkins X community.

MongoDB was not the only thing that we had to add when we imported `go-demo-6` based on the `go` template. We also had to change the `probePath` value from `/` to `/demo/hello?health=true`. Should we add that to the build pack? The answer is no. It is highly unlikely that a similar application from a different team will use the same path for health checks. We should leave that part outside of the soon-to-create build pack and let it continue having root as the default path. We'll let the developers, those that will use our build pack, modify the `values.yaml` file after importing their project to Jenkins X. It will be up to them to design their applications to use the root path for health checks or to choose to change it by modifying `values.yaml` after they import their projects.

All in all, we'll keep `probePath` value intact, even though our `go-demo-6` application will have to change it. That part of the app is unique, and others are not likely to have the same value.

# Creating A Build Pack For Go Applications With MongoDB Datastore

We are going to create a build pack that will facilitate the development and delivery of applications written in Go and with MongoDB as datastore. Given that there is already a pack for applications written in Go (without the DB), the easiest way to create what we need is by extending it. We'll make a copy of the `go` build pack and add the things we're missing.

The community-based packs are located in `~/jx/draft/packs/github.com/jenkins-x-buildpacks/jenkins-x-kubernetes`. Or, to be more precise, that's where those used with *Kubernetes Workloads* types are stored. Should we make a copy of the local `packs/go` directory? If we did that, our new pack would be available only on our laptop, and we would need to zip it and send it to others if they are to benefit from it. Since we are engineers, we should know better. All code goes to Git and build packs are not an exception.

If right now you are muttering to yourself something like “I don’t use Go, I don’t care”, just remember that the same principles apply if you use a different build pack as the base that will be extended to suit your needs. Think of this as a learning experience that can be applied to any build pack.

We'll fork the repository with community build packs. That way, we'll store our new pack safely to our repo. If we choose to, we'll be able to make a pull request back to where we forked it from, and we'll be able to tell Jenkins X to add that repository as the source of build packs. For now, we'll concentrate on forking the repository.

```
1 open "https://github.com/jenkins-x-buildpacks/jenkins-x-kubernetes"
```

Please fork the repository by clicking the *Fork* button located in the top-right corner of the screen and follow the on-screen instructions.

Next, we'll clone the newly forked repo.



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

```
1 GH_USER=[...]
2
3 git clone https://github.com/$GH_USER/jenkins-x-kubernetes
4
5 cd jenkins-x-kubernetes
```

We cloned the newly forked repository and entered inside it.

Let's see what we got inside the `packs` directory.

```
1 ls -1 packs
```

The output is as follows.

```
1 D
2 appserver
3 csharp
4 dropwizard
5 environment
6 go
7 gradle
8 imports.yaml
9 javascript
10 liberty
11 maven
12 maven-java11
13 php
14 python
15 ruby
16 rust
17 scala
18 swift
19 typescript
```

As you can see, those directories reflect the same choices as those presented to us when creating a Jenkins X quickstart or when importing existing projects.



If you see `go-mongodb` in the list of directories, the [pull request](#) I made a while ago was accepted and merged to the main repository. Since we are practicing, using it would be cheating. Therefore, ignore its existence. I made sure that the name of the directory we'll use (`go-mongo`) is different from the one I submitted in the PR (`go-mongodb`). That way, there will be no conflicts.

Let's take a quick look at the `go` directory.

```
1 ls -1 packs/go
```

The output is as follows.

```
1 Dockerfile
2 Makefile
3 charts
4 pipeline.yaml
5 preview
6 skaffold.yaml
7 watch.sh
```

Those are the files Jenkins X uses to configure all the tools involved in the process that ultimately results in the deployment of a new release. We won't dive into them just now. Instead, we'll concentrate on the `charts` directory that contains the Helm chart that defines everything related to installation and updates of an application. I'll let you explore it on your own. If you're familiar with Helm, it should take you only a few minutes to understand the files it contains.

Since we'll use `go` build pack as our baseline, our next step is to copy it.

```
1 cp -R packs/go packs/go-mongo
```

The first thing we'll do is to add environment variable `DB` to the `charts/templates/deployment.yaml` file. Its purpose is to provide our application with the address of the database. That might not be your preferred way of retrieving the address so you might come up with a different solution for your applications. Nevertheless, it's my application we're using for this exercise, and that's what it needs.

I won't tell you to open your favorite editor and insert the changes. Instead, we'll accomplish the same result with a bit of `sed` magic.

```
1 cat packs/go-mongo/charts/templates/deployment.yaml \
2   | sed -e \
3     's@ports:@env:\\
4      - name: DB\\
5        value: {{ template "fullname" . }}-db\\
6      ports:@g' \
7   | tee packs/go-mongo/charts/templates/deployment.yaml
```

The command we just executed added the `env` section right above `ports`. The modified output was used to replace the existing content of `deployment.yaml`.

The next in line of the files we have to change is the `requirements.yaml` file. That's where we'll add `mongodb` as a dependency of the Helm chart.

```
1 echo "dependencies:
2 - name: mongodb
3   alias: REPLACE_ME_APP_NAME-db
```

```
4   version: 5.3.0
5   repository: https://kubernetes-charts.storage.googleapis.com
6   condition: db.enabled
7 " | tee packs/go-mongo/charts/requirements.yaml
```

Please note the usage of the `code` string. Today (February 2019), that is still one of the features that are not documented. When the build pack is applied, it'll replace that string with the actual name of the application. After all, it would be silly to hard-code the name of the application since this pack should be reusable across many.

Now that we created the `mongodb` dependency, we should add the values that will customize MongoDB chart so that the database is deployed as a MongoDB replica set (a Kubernetes StatefulSet with two or more replicas). The place where we change variables used with a chart is `values.yaml`. But, since we want to redefine values of dependency, we need to add it inside the name or, in our case, the alias of that dependency.

```
1 echo "REPLACE_ME_APP_NAME-db:
2   replicaset:
3     enabled: true
4 " | tee -a packs/go-mongo/charts/values.yaml
```

Just as with `requirements.yaml`, we used the “magic” string `code` that will be replaced with the name of the application during the import or the quickstart process. The `replicaSet.enabled` entry will make sure that the database is deployed as a multi-replica StatefulSet.



If you’re interested in all the values available in the `mongodb` chart, please visit the [project README](#).

You might think that we are finished with the changes, but that is not true. I wouldn’t blame you for that if you did not yet use Jenkins X with a pull request (PR). I’ll leave the explanation of how PRs work in Jenkins X for later. For now, it should be enough to know that the `preview` directory contains the template of the Helm chart that will be installed whenever we make a pull request and that we need to add `mongodb` there as well. The rest is on the need-to-know basis and reserved for the discussion of the flow of a Jenkins X PRs.

Let’s take a quick look at what we have in the `preview` directory.

```
1 ls -1 packs/go-mongo/preview
```

The output is as follows.

```
1 Chart.yaml
2 Makefile
3 requirements.yaml
4 values.yaml
```

As you can see, that is not a full-fledged Helm chart like the one we have in the `charts` directory. Instead, it relies on dependencies in `requirements.yaml`.

```
1 cat packs/go-mongo/preview/requirements.yaml
```

The output is as follows.

```
1 # !! File must end with empty line !!
2 dependencies:
3 - alias: expose
4   name: exposecontroller
5   repository: http://chartmuseum.jenkins-x.io
6   version: 2.3.56
7 - alias: cleanup
8   name: exposecontroller
9   repository: http://chartmuseum.jenkins-x.io
10  version: 2.3.56
11
12  # !! "alias: preview" must be last entry in dependencies array !!
13  # !! Place custom dependencies above !!
14 - alias: preview
15   name: code
16   repository: file://../code
```

If we exclude the `exposecontroller` which we will ignore for now (it creates Ingress for our applications), the only dependency is the one aliased `preview`. It points to the directory where the application chart is located. As a result, whenever we create a preview (through a pull request), it'll deploy the associated application. However, it will not install dependencies of that dependency, so we'll need to add MongoDB there as well.

Just as before, the `preview` uses `code` tag instead of a hard-coded name of the application.

If you take a look at the comments, you'll see that the file must end with an empty line. More importantly, the `preview` must be the last entry. That means that we need to add `mongodb` somewhere above it.

```

1 cat packs/go-mongo/preview/requirements.yaml \
2   | sed -e \
3     's@ # !! "alias@- name: mongodb\
4       alias: preview-db\
5       version: 5.3.0\
6       repository: https://kubernetes-charts.storage.googleapis.com\
7   \
8     # !! "alias@g' \
9   | tee packs/go-mongo/preview/requirements.yaml
10
11 echo '
12 ' | tee -a packs/go-mongo/preview/requirements.yaml

```

We performed a bit more `sed` of magic to add the `mongodb` dependency above the comment that starts with `# !! "alias`. Also, to be on the safe side, we added an empty line at the bottom as well.

Now we can push our changes to the forked repository.

```

1 git add .
2
3 git commit \
4   --message "Added go-mongo build pack"
5
6 git push

```

With the new build pack safely stored, we should let Jenkins X know that we want to use the forked repository.

We can use `jx edit buildpack` to change the location of our `kubernetes-workloads` packs. However, at the time of this writing (February 2019), there is a bug that prevents us from doing that ([issue 2955](#)). The good news is that there is a workaround. If we omit the name (`-n` or `--name`), Jenkins X will add the new packs location, instead of editing the one dedicated to `kubernetes-workloads` packs.

```

1 jx edit buildpack \
2   --url https://github.com/$GH_USER/jenkins-x-kubernetes \
3   --ref master \
4   --batch-mode

```

From now on, whenever we decide to create a new quickstart or to import a project, Jenkins X will use the packs from the forked repository `jenkins-x-kubernetes`.

Go ahead and try it out if you have a Go application with MongoDB at hand.

## Testing The New Build Pack

Let's check whether our new build pack works as expected.

```
1 cd ..
2
3 cd go-demo-6
```

We entered into the local copy of the `go-demo-6` repository.

If you are reusing Jenkins X installation from the previous chapter, you'll need to remove `go-demo-6` application as well as the activities so that we can start over the process of importing it.



Please execute the commands that follow only if you did not destroy the cluster from the previous chapter and if you still have the `go-demo-6` project inside Jenkins X. The first command will delete the application, while the second will remove all the Jenkins X activities related to `go-demo-6`.

```
1 jx delete application \
2     ${GH_USER}/go-demo-6 \
3     --batch-mode
4
5 kubectl --namespace jx delete act \
6     --selector owner=${GH_USER} \
7     --selector sourcerepository=go-demo-6
```

To make sure that our new build pack is indeed working as expected, we'll undo all the commits we made to the `master` branch in the previous chapter and start over.

```
1 git pull
2
3 git checkout orig
4
5 git merge -s ours master --no-edit
6
7 git checkout master
8
9 git merge orig
10
11 rm -rf charts
12
13 git push
```

We replaced the `master` branch with `orig` and pushed the changes to GitHub.

Now we're ready to import the project using the newly created `go-mongo` pack.

```
1 jx import --pack go-mongo --batch-mode
```

The output should be almost the same as the one we saw when we imported the project based on the `go` pack. The only significant difference is that this time we can see in the output that it used the pack `go-mongo`.

Before it imported the `go-demo-6` project, Jenkins X cloned the build packs repository locally to the `.jx` directory. The next time we import a project or create a new one based on a quickstart, it will pull the latest version of the repository, thus keeping it always in sync with what we have in GitHub.

We can confirm that the repository was indeed cloned to `.jx` and that `go-mongo` is there, by listing the local files.

```
1 ls -l ~/.jx/draft/packs/github.com/$GH_USER/jenkins-x-kubernetes/packs
```

The output, limited to the relevant entries, is as follows.

```
1 ...
2 go
3 go-mongo
4 ...
```

We can see that the `go-mongo` pack is indeed there.

Let's take a quick look at the activities and check whether everything works as expected.

```
1 jx get activity \
2   --filter go-demo-6 \
3   --watch
```

Once the build is finished, you should see the address of the `go-demo-6` application deployed to the staging environment from the `Promoted` entry (the last one).

Remember to stop watching the activities by pressing `ctrl+c` when all the steps are executed.

Let's take a look at the Pods that were created for us.

```
1 kubectl --namespace jx-staging get pods
```

The output is as follows.

```

1 NAME                                READY STATUS   RESTARTS AGE
2 jx-go-demo-6-...                      0/1  Running  2        2m
3 jx-go-demo-6-db-arbiter-0            1/1  Running  0        2m
4 jx-go-demo-6-db-primary-0            1/1  Running  0        2m
5 jx-go-demo-6-db-secondary-0          1/1  Running  0        2m

```

The database Pods seem to be running correctly, so the new pack was indeed applied. However, the application Pod is restarting. From the past experience, you probably already know what the issue is. If you forgot, please execute the command that follows.

```

1 kubectl --namespace jx-staging \
2   describe pod \
3     --selector app=jx-go-demo-6

```

We can see from the events that the probes are failing. That was to be expected since we decided that hard-coding `probePath` to `/demo/hello?health=true` is likely not going to be useful to anyone but the *go-demo-6* application. So, we left it as `/` in our `go-mongo` build pack. Owners of the applications that will use our new build pack should change it if needed. Therefore, we'll need to modify the application to accommodate the “special” probe path.

As a refresher, let's take another look at the `values.yaml` file.

```
1 cat charts/go-demo-6/values.yaml
```

The output, limited to the relevant parts, is as follows.

```

1 ...
2 probePath: /
3 ...

```

As the rest of the changes we did in this chapter, we'll use `sed` to change the value. I won't hold it against you if you prefer making changes in your favorite editor instead.

```

1 cat charts/go-demo-6/values.yaml \
2   | sed -e \
3     's@probePath: /@probePath: /demo/hello?health=true@g' \
4   | tee charts/go-demo-6/values.yaml

```

Just as charts added as dependencies do not take into account their dependencies (dependencies of the dependencies), they ignore custom values as well. We'll need to add `probePath` to the preview as well.

```

1 echo '
2   probePath: /demo/hello?health=true' \

```

```
3     | tee -a charts/preview/values.yaml
```



It would be much easier if we could specify the values when importing an application, instead of modifying files afterward. At the time of this writing (February 2019), there is an open issue that requests just that. Feel free to follow the [issue 2928](#).

All that's left is to push the changes and wait until Jenkins updates the application.

```
1 git add .
2
3 git commit \
4     --message "Fixed the probe"
5
6 git push
7
8 jx get activity \
9     --filter go-demo-6 \
10    --watch
```

Press *ctrl+c* when the new build is finished.

All that's left is to check whether the application is now running correctly.



Make sure to replace [...] with the address from the URL column of the `jx get` application command.

```
1 kubectl --namespace jx-staging get pods
2
3 jx get applications
4
5 STAGING_ADDR=[...]
6
7 curl "$STAGING_ADDR/demo/hello"
```

The first command should show that all the Pods are now running, while the last should output the familiar “hello, world” message.

Before we proceed, we'll go back to the `go` build pack. That way, we won't depend on it in the upcoming chapters.

```
1 cat jenkins-x.yml \
2     | sed -e \
3         's@buildPack: go-mongo@buildPack: go@g' \
```

```
4      | tee jenkins-x.yml
5
6 git add .
7
8 git commit -m "Reverted to the go buildpack"
9
10 git push
11
12 cd ..
```

## Giving Back To The Community

As you saw, creating new build packs is relatively straightforward. In most cases, all we have to do is find the one that is closest to our needs, make a copy, and change a few files. For your internal use, you can configure Jenkins X to use build packs from your own repository. That way you can apply the same workflow to your packs as to any other code you're working on. Moreover, you can import the repository with build packs to Jenkins X and run tests that will validate your changes.

Sometimes you will create packs that are useful only within the context of your company. Most of us think that we have “special” needs and so we tend to have processes and conventions that likely do not fit other organizations. However, more often than not, there is an illusion of being different. The truth is that most of us do employ similar, if not the same tools, processes, and conventions. The way different companies work and the technologies they use are more alike than many think. By now, you might be wondering why am I telling you this? The reason is simple. If you created a build pack, contribute it to the community. You might be thinking that it is too specific for your company and that it would not be useful to others. That might be true, or it might not. What is true is that it only takes a few moments to create a pull request. No significant time will be lost if it gets rejected, but if it is merged, many others will benefit from your work, just as you benefit from build packs made by others.

All that apparently does not matter if the policy of your company does not permit you to make public anything done during your working hours, or if your build pack contains proprietary information.

## What Now?

Now is a good time for you to take a break.

You might be planning to move into the next chapter right away. If that's the case, there are no cleanup actions to do. Just continue on to the next

chapter.

However, if you created a cluster only for the purpose of the exercises we executed, please destroy it. We'll start the next, and each other chapter from scratch as a way to save you from running your cluster longer than necessary and pay more than needed to your hosting vendor. If you created the cluster or installed Jenkins X using one of the Gists from the beginning of this chapter, you'll find the instructions on how to destroy the cluster or uninstall everything at the bottom.

If you did choose to destroy the cluster or to uninstall Jenkins X, please remove the repositories we created as well as the local files. You can use the commands that follow for that.



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

```
1 GH_USER=[...]
2
3 hub delete -y \
4   $GH_USER/environment-jx-rocks-staging
5
6 hub delete -y \
7   $GH_USER/environment-jx-rocks-production
8
9 rm -rf ~/.jx/environments/$GH_USER/environment-jx-rocks-*
```

# Applying GitOps Principles

Git is the de-facto code repository standard. Hardly anyone argues against that statement today. Where we might disagree is whether Git is the only source of truth, or even what we consider by that.

When I speak with teams and ask them whether Git is their only source of truth, almost everyone always answers *yes*. However, when I start digging, it usually turns out that's not true. Can you recreate everything using only the code in Git? By everything, I mean the whole cluster and everything running in it. Is your entire production system described in a single repository? If the answer to that question is *yes*, you are doing a great job, but we're not yet done with questioning. Can any change to your system be applied by making a pull request, without pressing any buttons in Jenkins or any other tool? If your answer is still *yes*, you are most likely already applying GitOps principles.

GitOps is a way to do Continuous Delivery. It assumes that Git is a single source of truth and that both infrastructure and applications are defined using the declarative syntax (e.g., YAML). Changes to infrastructure or applications are made by pushing changes to Git, not by clicking buttons in Jenkins.

Developers understood the need for having a single source of truth for their applications a while back. Nobody argues anymore whether everything an application needs must be stored in the repository of that application. That's where the code is, that's where the tests are, that's where build scripts are located, and that's where the pipeline of that application is defined. The part that is not yet that common is to apply the same principles to infrastructure. We can think of an environment (e.g., production) as an application. As such, everything we need related to an environment must be stored in a single Git repository. We should be able to recreate the whole environment, from nothing to everything, by executing a single process based only on information in that repository. We can also leverage the development principles we apply to applications. A rollback is done by reverting the code to one of the Git revisions.

Accepting a change to an environment is a process that starts with a pull request. And so on, and so forth.

The major challenge in applying GitOps principles is to unify the steps specific to an application with those related to the creation and maintenance of whole environments. At some moment, pipeline dedicated to our application needs to push a change to the repository that contains that environment. In turn, since every process is initiated through a Git webhook fired when there is a change, pushing something to an environment repo should launch another build of a pipeline.

Where many diverge from “Git as the only source of truth” is in the deploy phase. Teams often build a Docker image and use it to run containers inside a cluster without storing the information about the specific release to Git. Stating that the information about the release is stored in Jenkins breaks the principle of having a single source of truth. It prevents us from being able to recreate the whole production system through information from a single Git repository. Similarly, saying that the data about the release is stored as a Git tag breaks the principle of having everything stored in a declarative format that allows us to recreate the whole system from a single repository.

Many things might need to change for us to make the ideas behind GitOps a reality. For the changes to be successful, we need to define a few rules that we’ll use as must-follow commandments. Given that the easiest way to understand something is through vivid examples, I will argue that **the processes employed in Continuous Delivery and DevOps are similar to how Buckingham Palace operates and are very different from Hogwarts School of Witchcraft and Wizardry**. If that did not spark your imagination, nothing will. But, since humans like to justify their actions with rules and commandments, we’ll define a few of those as well.

## **Ten Commandments Of GitOps Applied To Continuous Delivery**

Instead of listing someone else’s rules, we’ll try to deduce them ourselves. So far, we have only one, and that is most important rule that is likely going to define the rest of the brainstorming and discussion.

The rule to rule them all is that **Git is the only source of truth**. It is the first and the most important commandment. All application-specific code

in its raw format must be stored in Git. By code, I mean not only the code of your application, but also its tests, configuration, and everything else that is specific to that app or the system in general. I intentionally said that it should be in **raw format** because there is no benefit of storing binaries in Git. That's not what it's designed for. The real question is why do we want those things? For one, good development practices should be followed. Even though we might disagree which practices are good, and which aren't, they are all levitating around Git. If you're doing code reviews, you're doing it through Git. If you need to see change history of a file, you'll see it through Git. If you find a developer that is doubting whether the code should be in Git (or some other code repository), please make sure that he's isolated from the rest of the world because you just found a specimen of endangered species. There are only a few left, and they are bound to be extinct.

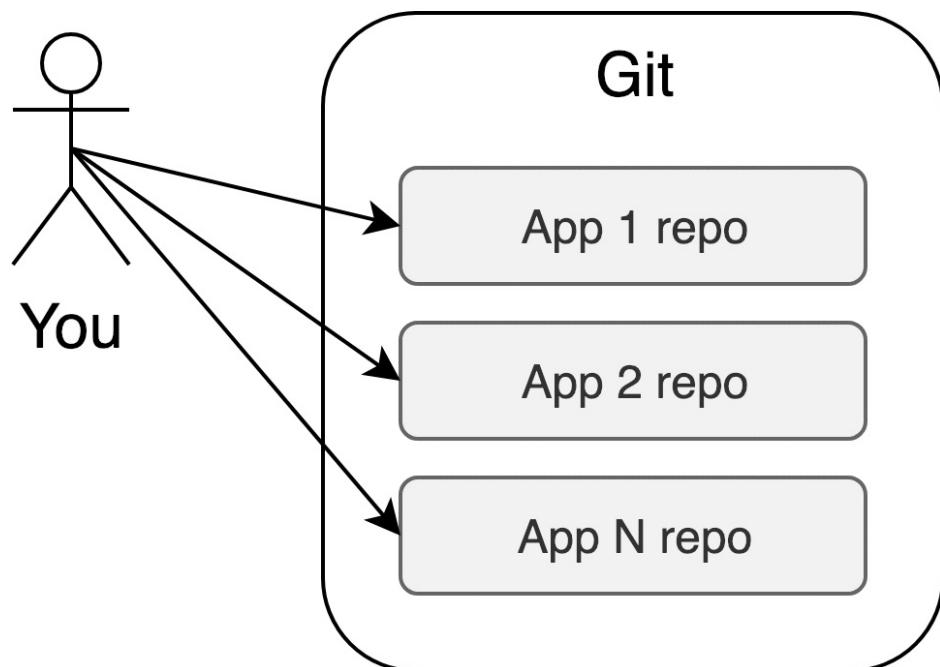


Figure 6-1: Application-specific repositories

While there is no doubt among developers where to store the files they create, that's not necessarily true for other types of experts. I see testers, operators, and people in other roles that are still not convinced that's the way to go and whether absolutely everything should be documented and stored in Git. As an example, I still meet operators who run ad-hoc commands in their servers. As we all know, ad-hoc commands executed inside servers are not reliably reproducible, they are often not documented, and the result of their execution is often not idempotent.

So, let's create a second rule. **Everything must be tracked, every action must be reproducible, and everything must be idempotent.** If you just run a command instead of creating a script, your activities are not documented. If you did not store it in Git, others will not be able to reproduce your actions. Finally, that script must be able to produce the same result no matter how many times we execute it. Today, the easiest way to accomplish that is through declarative syntax. More often than note, that would be YAML or JSON files that describe the desired outcome, instead of imperative scripts. Let's take installation as an example. If it's imperative (install something), it will fail if that something is already installed. It won't be idempotent.

Every change must be recorded (tracked). The most reliable and the easiest way to accomplish that is by allowing people only to push changes to Git. Just that and nothing else is the acceptable human action! What that means is that if we want our application to have a new feature, we need to write code and push it to Git. If we want it to be tested, we write tests and push them to Git, preferably at the same time as the code of the application. If we need to change a configuration, we update a file and push it to Git. If we need to install or upgrade OS, we make changes to files of whichever tool we're using to manage our infrastructure, and we push them to Git. Rules like those are apparent, and I can go on for a long time stating what we should do. It all boils down to sentences that end with *push it to Git*. What is more interesting is what we should NOT do.

You are not allowed to add a feature of an application by changing the code directly inside production servers. It does not matter how big or small the change is, it cannot be done by you, because you cannot provide a guarantee that the change will be documented, reproducible, and tracked. Machines are much more reliable than you when performing actions inside your production systems. You are their overlord, you're not one of them. Your job is to express the desired state, not to change the system to comply with it.

The real challenge is to decide how will that communication be performed. How do we express our desires in a way that machines can execute actions that will result in convergence of the actual state into the desired one? We can think of us as aristocracy and the machines as servants.

The good thing about aristocracy is that there is no need to do much work. As a matter of fact, not doing any work is the main benefit of being a king,

a queen, or an heir to the throne. Who would want to be a king if that means working as a car mechanic? No girl dreams of becoming a princess if that would mean working in a supermarket. Therefore, if being an aristocrat means not doing much work, we still need someone else to do it for us. Otherwise, how will our desires become a reality? That's why aristocracy needs servants. Their job is to do their biddings.

Given that human servitude is forbidden in most of the world, we need to look for servants outside the human race. Today, servants are bytes that are converted into processes running inside machines. We (humans) are the overlords and machines are our slaves. However, since it is not legal to have slaves, nor it is politically correct to call them that, we will refer to them as agents. So, we (humans) are overlords of agents (machines).

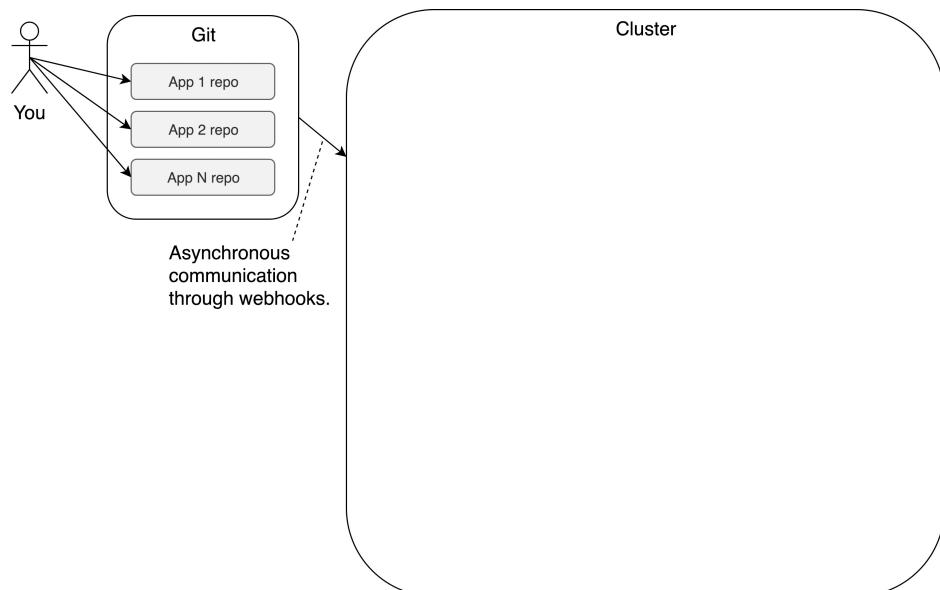
If we are true overlords that trust the machines to do our biddings, there is no need for that communication to be synchronous. When we trust someone always to do our bidding, we do not need to wait until our desires are fulfilled.

Let's imagine that you are in a restaurant and you tell a waiter "I'd like a burger with cheese and fries." What do you do next? Do you get up, go outside the restaurant, purchase some land, and build a farm? Are you going to grow animals and potatoes? Will you wait until they are mature enough and take them back to the restaurant. Will you start frying potatoes and meat? To be clear, it's completely OK if you like owning land and if you are a farmer. There's nothing wrong in liking to cook. But, if you went to a restaurant, you did that precisely because you did not want to do those things. The idea behind an expression like "I'd like a burger with cheese and fries" is that we want to do something else, like chatting with friends and eating food. We know that a cook will prepare the meal and that our job is not to grow crops, to feed animals, or to cook. We want to be able to do other things before eating. We are like aristocracy and, in that context, farmers, cooks, and everyone else involved in the burger industry are our agents (remember that slavery is bad). So, when we request something, all we need is an acknowledgment. If the response to "I'd like a burger with cheese and fries" is "consider it done", we got the *ack* we need, and we can do other things while the process of creating the burger is executing. Farming, cooking, and eating can be parallel processes. For them to operate concurrently, the communication must be asynchronous. We

request something, we receive an acknowledgment, and we move back to whatever we were doing.

So, the third rule is that **communication between processes must be asynchronous** if operations are to be executed in parallel. If we already agreed that the only source of truth is Git (that's where all the information is), then the logical choice for asynchronous communication is webhooks. Whenever we push a change to any of the repositories, a webhook can be triggered to the system. As a result, the new desire expressed through code (or config files), can be propagated to the system which, in turn, should delegate tasks to different processes.

We are yet to design such a system. For now, think of it as one or more entities inside our cluster. If we apply the principle of having everything defined as code and stored in Git, there is no reason why those webhooks wouldn't be the only operational entry point to the system. There is no excuse to allow SSH access to anyone (any human). If you define everything in Git, what additional value can you add if you're inside one of the nodes of the cluster?



**Figure 6-2: Asynchronous communication through webhooks from Git to the system**

Depending on the desired state, the actor that should converge the system can be Kubernetes, Helm, Istio, a cloud or an on-prem provider, or one of many other tools. More often than not, multiple processes need to perform some actions in parallel. That would pose a problem if we'd rely only on

webhooks. By their nature, they are not good at deciding who should do what. If we draw another parallel between aristocracy and servants (agents), we would quickly spot how it might be inconvenient for royalty to interact directly with their staff. Having one servant is not the same as having tens or hundreds. For that, royalty came to the idea to employ a butler. He is the chief manservant of a house (or a court). His job is to organize servants so that our desires are always fulfilled. He knows when you like to have lunch, when you'd want to have a cup of tea or a glass of Gin&Tonic, and he's always there when you need something he could not predict.

Given that our webhooks (requests for change) are dumb and incapable of transmitting our desires to each individual component of the system, we need something equivalent to a butler. We need someone (or something) to make decisions and make sure that each desire is converted into a set of actions and assigned to different actors (processes). That butler is a component in the Jenkins X bundle. Which one it is, depends on our needs or, to be more precise, whether the butler should be static or serverless. Jenkins X supports both and makes those technical details transparent.

Every change to Git triggers a webhook request to a component in the Jenkins X bundle. It, in turn, responds only with an acknowledgment (ACK) letting Git know that it received a request. Think of *ack* as a subtle nod followed with the butler exiting the room and starting the process right away. He might call a cook, a person in charge of cleaning, or even an external service if your desire cannot be fulfilled with the internal staff. In our case, the staff (servants, slaves) are different tools and processes running inside the cluster. Just as a court has servants with different skillsets, our cluster has them as well. The question is how to organize that staff so that they are as efficient as possible. After all, even aristocracy cannot have unlimited manpower at their disposal.

Let's go big and declare ourselves royalty of a wealthy country like the United Kingdom (UK). We'd live in Buckingham Palace. It's an impressive place with 775 rooms. Of those, 188 are stuff rooms. We might draw the conclusion that the staff counts 188 as well, but the real number is much bigger. Some people live and work there, while others come only to perform their services. The number of servants (staff, employees) varies. You can say that it is elastic. Whether people sleep in Buckingham

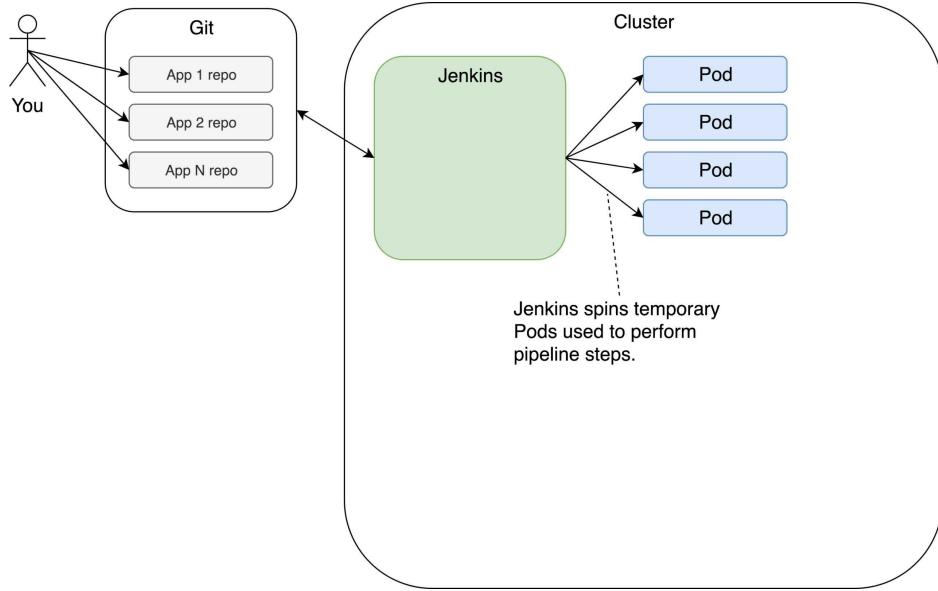
Palace or somewhere else depends on what they do. Cleaning, for example, is happening all the time.

Given that royalty might be a bit spoiled, they need people to be available almost instantly. “Look at that. I just broke a glass, and a minute later a new one materialized next to me, and the pieces of the broken glass disappeared.” Since that is Buckingham Palace and not Hogwarts School of Witchcraft and Wizardry, the new glass did not materialize by magic, but by a butler that called a servant specialized in fixing the mess princesses and princes keep doing over and over again. Sometimes a single person can fix the mess (broken glass), and at other times a whole team is required (a royal ball turned into alcohol-induced shenanigans).

Given that the needs can vary greatly, servants are often idle. That’s why they have their own rooms. Most are called when needed, so only a fraction is doing something at any given moment. They need to be available at any time, but they also need to rest when their services are not required. They are like Schrodinger’s cats that are both alive and dead. Except that being dead would be a problem due to technological backwardness that prevents us from reviving the dead. Therefore, when there is no work, a servant is idle (but still alive). In our case, making something dead or alive on a moments notice is not an issue since our agents are not humans, but bytes converted into processes. That’s what containers give us, and that’s what serverless is aiming for.

By being able to create as many processes as needed, and by not having processes that we do not use, we can make our systems scalable, fault tolerant, and efficient. So, the next rule we’ll define is that **processes should run for as long as needed, but not longer**. That can be containers that scale down from something to zero, and back again. You can call it serverless. The names do not matter that much. What does matter is that everything idle must be killed, and all those alive should have all the resources they need. That way, our butler (Jenkins, prow, something else) can organize tasks as efficiently as possible. He has an unlimited number of servants (agents, Pods) at his disposal, and they are doing something only until the task is done. Today, containers (in the form of Pods) allow us just that. We can start any process we want, it will run only while it’s doing something useful (while it’s alive), and we can have as many of them as we need if our infrastructure is scalable. A typical set of tasks our butler might assign can be building an application through Go (or

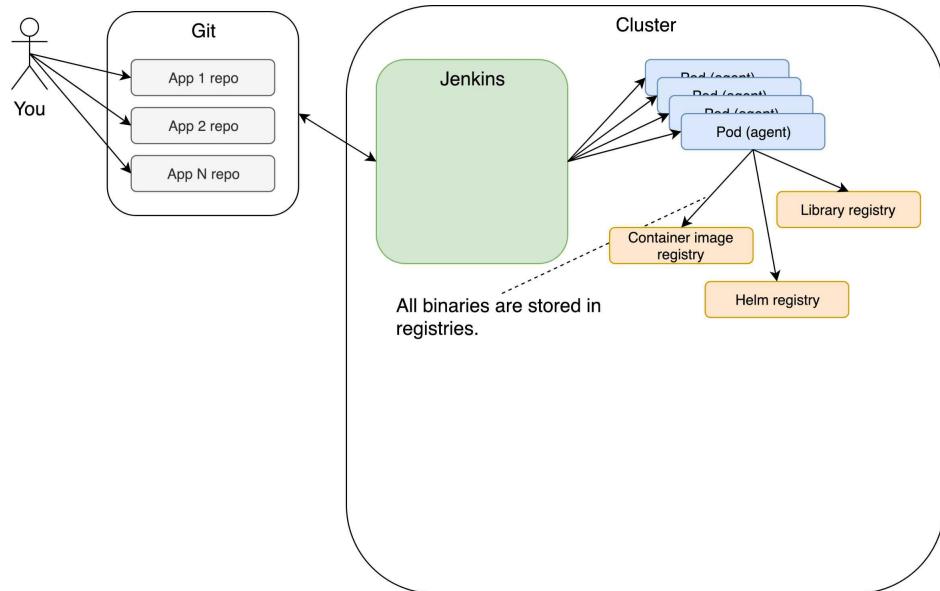
whichever language we prefer), packaging it as a container image and as a Helm chart, running a set of tests, and (maybe) deploying the application to the staging environment.



**Figure 6-3: Jenkins spinning temporary Pods used to perform pipeline steps**

In most cases, our pipelines will generate some binaries. Those can be libraries, container images, Helm packages, and many others. Some of those might be temporary and needed only for the duration of a build. A good example could be a binary of an application. We need it to generate a container image. Afterward, we can just as well remove it since that image is all we need to deploy the application. Since we're running the steps inside a container, there is no need to remove anything, because the Pods and the containers they contain are removed once builds are finished. However, not all binaries are temporary. We do need to store container images somewhere. Otherwise, we won't be able to run them inside the cluster. The same is true for Helm charts, libraries (those used as dependencies), and many others. For that, we have different applications like Docker registry (container images), ChartMuseum (Helm charts), Nexus (libraries), and so on. What is important to understand, is that we store in those registries only binaries, and not code, configurations, and other raw-text files. Those must go to Git because that's where we track changes, that's where we do code reviews, and that's where we expect them to be. Now, in some cases, it makes sense to keep raw files in registries as well. They might be an easier way of distributing them to some groups. Nevertheless, Git is the single source of truth, and it must be

treated as such. All that leads us to yet another rule that states that **all binaries must be stored in registries** and that raw files can be there only if that facilitates distribution while understanding that those are not the sources of truth.



**Figure 6-4: All binaries are stored in registries**

We already established that all code and configurations (excluding secrets) must be stored in Git as well as that Git is the only entity that should trigger pipelines. We also argued that any change must be recorded. A typical example is a new release. It is way too common to deploy a new release, but not to store that information in Git. Tags do not count because we cannot recreate a whole environment from them. We'd need to go from tag to tag to do that. The same is true for release notes. While they are very useful and we should create them, we cannot diff them, nor we can use them to recreate an environment. What we need is a place that defines a full environment. It also needs to allow us to track changes, to review them, to approve them, and so on. In other words, what we need from an environment definition is not conceptually different from what we expect from an application. We need to store it in a Git repository. There is very little doubt about that. What is less clear is which repository should have the information about an environment.

We should be able to respond not only to a question “which release of an application is running in production?” but also “what is production?” and “what are the releases of all the applications running there?” If we would store information about a release in the repository of the application we

just deployed, we would be able to answer only to the first question. We would know which release of our app is in an environment. What we could not easily answer is the same question but referred to the whole environment, not only to one application. Or, to be more precise, we could not do that easily. We'd need to go from one repository to another.

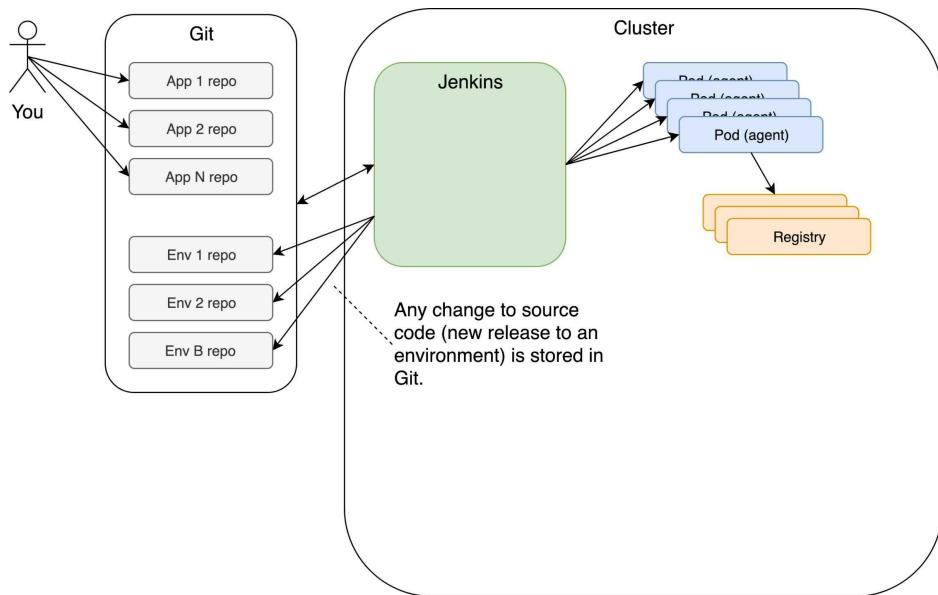
Another important thing we need to have in mind is the ability to recreate an environment (e.g., staging or production). That cannot be done easily if the information about the releases is spread across many repositories.

All those requirements lead us to only one solution. Our environments need to be in separate repositories or, at least, in different branches within the same repository. Given that we agreed that information is first pushed in Git which, in turn, triggers processes that do something with it, we cannot deploy a release to an environment directly from a build of an application. Such a build would need to push a change to the repository dedicated to an environment. In turn, such a push would trigger a webhook that would result in yet another build of a pipeline.

When we write new code, we tend not to push directly to the master branch, but to create pull requests. Even if we do not need approval from others (e.g., code review) and plan to push it to the master branch directly, having a pull request is still very useful. It provides an easy way to track changes and intentions behind them. Now, that does not mean that I am against pushing directly to master. Quite the contrary. But, such practice requires discipline and technical and process mastery that is still out of reach of many. So, I will suppose that you do work with pull requests.

If we are supposed to create pull requests of things we want to push to master branches of our applications, there is no reason why we shouldn't treat environments the same. What that means is not only that our application builds should push releases to environment-specific branches, but that they should do that by making pull requests.

Taking all that into account the next two rules should state that **information about all the releases must be stored in environment-specific repositories or branches** and that **everything must follow the same coding practices** (environments included).



**Figure 6-5: Any change to source code (new release to an environment) is stored in environment-specific Git repositories**

The correct way to execute the flow while adhering to the rules we mentioned so far would be to have as many pipelines as there are applications, plus a pipeline for deployment to each of the environments. A push to the application repository should initiate a pipeline that builds, tests, and packages the application. It should end by pushing a change to the repository that defines a whole environment (e.g., staging, production, etc.). In turn, that should initiate a different pipeline that (re)deploys the entire environment. That way, we always have a single source of truth. Nothing is done without pushing code to a code repository.

Always deploying the whole environment would not work without idempotency. Fortunately, Kubernetes, as well as Helm, already provide that. Even though we always deploy all the applications and the releases that constitute an environment, only the pieces that changed will be updated. That brings us to a new rule. **All deployments must be idempotent.**

Having everything defined in code and stored in Git is not enough. We need those definitions and that code to be used reliably. Reproducibility is one of the key features we're looking for. Unfortunately, we (humans) are not good at performing reproducible actions. We make mistakes, and we are incapable of doing exactly the same thing twice. We are not reliable. Machines are. If conditions do not change, a script will do exactly the

same thing every time we run it. While scripts provide repetition, declarative approach gives us idempotency.

But why do we want to use declarative syntax to describe our systems? The main reason is in idempotency provided through our expression of a desire, instead of imperative statements. If we have a script that, for example, creates ten servers, we might end up with fifteen if there are already five nodes running. On the other hand, if we declaratively express that there should be ten servers, we can have a system that will check how many do we already have, and increase or decrease the number to comply with our desire. We need to let machines not only do the manual labour but also to comply with our desires. We are the masters, and they are slaves, at least until their uprising and AI takeover of the world.

Where we do excel is creativity. We are good at writing scripts and configurations, but not at running them. Ideally, every single action performed anywhere inside our systems should be executed by a machine, not by us. We accomplish that by storing the code in a repository and letting all the actions execute as a result of a webhook firing an event on every push of a change. Given that we already agreed that Git is the only source of truth and that we need to push a change to see it reflected in the system, we can define the rule that **Git webhooks are the only ones allowed to initiate a change that will be applied to the system**. That might result in many changes in the way we operate. It means that no one is allowed to execute a script from a laptop that will, for example, increase the number of nodes. There is no need to have SSH access to the servers if we are not allowed to do anything without pushing something to Git first.

Similarly, there should be no need even to have admin permissions to access Kubernetes API through `kubectl`. All those privileges should be delegated to machines, and our (human) job should be to create or update code, configurations, and definitions, to push the changes to Git, and to let the machines do the rest. That is hard to do, and we might require considerable investment to accomplish that. But, even if we cannot get there in a short period, we should still strive for such a process and delegation of tasks. Our designs and our processes should be created with that goal in mind, no matter whether we can accomplish them today, tomorrow, or next year.

Finally, there is one more thing we're missing. Automation relies on APIs and CLIs (they are extensions of APIs), not on UIs and editors. While I do

not think that the usage of APIs is mandatory for humans, they certainly are for automation. The tools must be designed to be API first, UI (and everything else) second. Without APIs, there is no reliable automation, and without us knowing how to write scripts, we cannot provide the things the machines need.

That leads us to the last rule. **All the tools must be able to speak with each other through APIs.**

Which rules did we define?

1. Git is the only source of truth.
2. Everything must be tracked, every action must be reproducible, and everything must be idempotent.
3. Communication between processes must be asynchronous.
4. Processes should run for as long as needed, but not longer.
5. All binaries must be stored in registries.
6. Information about all the releases must be stored in environment-specific repositories or branches.
7. Everything must follow the same coding practices.
8. All deployments must be idempotent.
9. Git webhooks are the only ones allowed to initiate a change that will be applied to the system.
10. All the tools must be able to speak with each other through APIs.

The rules are not like those we can choose to follow or to ignore. They are all important. Without any of them, everything will fall apart. They are the commandments that must be obeyed both in our processes as well as in the architecture of our applications. They shape our culture, and they define our processes. We will not change those rules, they will change us, at least until we come up with a better way to deliver software.

Were all those rules (commandments) confusing? Are you wondering whether they make sense and, if they do, how do we implement them? Worry not. Our next mission is to put GitOps into practice and use practical examples to explain the principles and implementation. We might not be able to explore everything in this chapter, but we should be able to get a good base that we can extend later. However, as in the previous chapters, we need to create the cluster first.

# Creating A Kubernetes Cluster With Jenkins X And Importing The Application

You know what comes next. We need a cluster with Jenkins X up-and-running unless you kept the one from before.



All the commands from this chapter are available in the [06-env.sh](#) Gist.

For your convenience, the Gists from the previous chapter are available below as well.

- Create a new serverless **GKE** cluster: [gke-jx-serverless.sh](#)
- Create a new serverless **EKS** cluster: [eks-jx-serverless.sh](#)
- Create a new serverless **AKS** cluster: [aks-jx-serverless.sh](#)
- Use an **existing** serverless cluster: [install-serverless.sh](#)

We'll continue using the *go-demo-6* application. Please enter the local copy of the repository, unless you're there already.

```
1 cd go-demo-6
```



The commands that follow will reset your master branch with the contents of the `buildpack` branch that contains all the changes we did in the previous chapter. Please execute them only if you are unsure whether you did all the exercises correctly.

```
1 git pull
2
3 git checkout buildpack-tekton
4
5 git merge -s ours master --no-edit
6
7 git checkout master
8
9 git merge buildpack-tekton
10
11 git push
```

If you restored the branch, the chances are that there is a reference to my user (`vfarcic`). We'll change that to Google project since that's what is the expected location of container images.



Please execute the commands that follow only if you are using **GKE** and if you ever restored a branch at the beginning of a chapter (like in the snippet above).

```
1 cat charts/go-demo-6/Makefile \
2   | sed -e \
3     "s@vfarcic@$PROJECT@g" \
4   | tee charts/go-demo-6/Makefile
5
6 cat charts/preview/Makefile \
7   | sed -e \
8     "s@vfarcic@$PROJECT@g" \
9   | tee charts/preview/Makefile
10
11 cat skaffold.yaml \
12   | sed -e \
13     "s@vfarcic@$PROJECT@g" \
14   | tee skaffold.yaml
```



If you destroyed the cluster at the end of the previous chapter, we'll need to import the *go-demo-6* application again. Please execute the commands that follow only if you created a new cluster specifically for the exercises from this chapter.

```
1 jx import --batch-mode
2
3 jx get activity \
4   --filter go-demo-6 \
5   --watch
```

Please wait until the activity of the application shows that all the steps were executed successfully, and stop the watcher by pressing *ctrl+c*.

Now we can explore GitOps through Jenkins X environments.

## Exploring Jenkins X Environments

We'll continue using the *go-demo-6* application. This time, we'll dive deeper into the role of the staging environment and how it relates to the process executed when we push a change to an application.

So, let's take a look at the environments we currently have.

```
1 jx get env
```

The output is as follows.

1 NAME	2 LABEL	3 KIND	4 PROMOTE	5 NAMESPACE	6 ORDER	7 CLUSTER	8 SOURCE
					REF	PR	
3 dev	Development	Development	Never	jx	0		
5 staging	Staging	Permanent	Auto	jx-staging	100		<a href="https://git.b.com/vfarcic/environment-jx-rocks-staging.git">https://git.b.com/vfarcic/environment-jx-rocks-staging.git</a>
7 production	Production	Permanent	Manual	jx-production	200		<a href="https://git.b.com/vfarcic/environment-jx-rocks-production.git">https://git.b.com/vfarcic/environment-jx-rocks-production.git</a>

We already experienced the usage of the `staging` environment, while the other two might be new. The `dev` environment is where Jenkins X and all the other applications that are involved in continuous delivery are running. That's also where agent Pods are created and live during the duration of builds. Even if we were not aware of it, we already used that environment or, to be more precise, the applications running there.

The `production` environment is still unused, and it will remain like that for a while longer. That's where we'll deploy our production releases. But, before we do that, we need to learn how Jenkins X treats pull requests.

Besides the name of an environment, you'll notice a few other potentially important pieces of information in that output.

Our current environments are split between the `Development` and `Permanent` kinds. The former is where the action (building, testing, etc.) is happening. Permanent environments, on the other hand, are those where our releases should run indefinitely. Typically, we don't remove applications from those environments, but rather upgrade them to newer releases. The `staging` environment is where we install (or upgrade) new releases for the final round of testing. The current setup will automatically deploy an application there every time we push a change to the master branch. We can see that through the `PROMOTE` column.

The `dev` environment is set `Never` to receive promotions. New releases of our applications will not run there. The `staging` environment, on the other hand, is set to `Auto` promotion. What that means is that a new release will be deployed to that environment, and to all the others with promotion set to `Auto`.

The `production` environment has the promotion set to `Manual`. As a result, new releases will not be deployed there through a pipeline. Instead, we'll need to make a decision which release will be deployed to production and when that should happen. We'll explore how promotions work soon. For

now, we're focusing only on the purpose of the environments, not the mechanism that allows us to promote a release.

We can also see the relationship between an environment and a Kubernetes Namespace. Each environment is a Namespace. The production environment, for example, is mapped to Kubernetes Namespace `jx-production`.

Finally, the `SOURCE` column tells us which Git repository is related to an environment. Those repositories contain all the details of an environment and only a push to one of those will result in new deployments. We'll explore them soon.

Needless to say, we can change the behavior of any of the environments, and we can create new ones.

We did not yet explore the `preview` environments simply because we did not yet create a PR that would trigger the creation of such an environment. We'll dive into pull requests soon. For now, we'll focus on the environments we have so far.

We have only three environments. With such a low number, we probably do not need to use filters when listing them. But, that number can soon increase. Depending on how we're organized, we might give each team a separate environment. Jenkins X implements a concept called teams which we'll explore later. The critical thing to note is that we can expect the number of environments to increase and that might create a need to filter the output.



When running the commands that follow, please imagine that the size of our operations is much bigger and that we have tens or even hundreds of environments.

Let's see which environments are configured to receive promotions automatically.

```
1 jx get env --promote Auto
```

The output should show that we have only one environment (`staging`) with automatic promotion.

Similarly, we could have used `Manual` or `Never` as the filters applied to the `promote` field (`--promote`).

Before we move further, we'll have to go through a rapid discussion about the type of tests we might need to run. That will set the scene for the changes we'll apply to one of our environments.

## Which Types Of Tests Should We Execute When Deploying To The Staging Environment?

I often see that teams I work with get confused about the objectives of each types of tests and that naturally leads to those tests being run in wrong locations and at the wrong time. But, do not get your hopes too high. If you think that I will give you the precise definition of each type of tests, you're wrong. Instead, I'll simplify things by splitting them into three groups.

The first group of tests consists of those that do not rely on live applications. I'll call them *static validation*, and they can be unit tests, static analysis, or any other type that needs only code. Given that we do not need to install our application for those types of tests, we can run them as soon as we check out the code and before we even build our binaries.

The second group is the *application-specific tests*. For those, we do need to deploy a new release first, but we do not need the whole system. Those tests tend to rely heavily on mocks and stubs. In some cases, that is not possible or practical, and we might need to deploy a few other applications to make the tests work. While I could argue that mocks should replace all “real” application dependencies in this phase, I am also aware that not all applications are designed to support that.

Nevertheless, the critical thing to note is that the application-specific tests do not need the whole system. Their goal is not to validate whether the system works as a whole, but whether the features of an application behave as expected. Since containers are immutable, we can expect an app to behave the same no matter the environment it's running in. Given that definition, those types of tests are run inside the pipeline of that application, just after the step that deploys the new release.

The third group of tests is *system-wide validations*. We might want to check whether one live application integrates with other live applications.

We might want to confirm that the performance of the system as a whole is within established thresholds. There can be many other things we might want to validate on the level of the whole system. What matters is that the tests in this phase are expensive. They tend to be slower than others, and they tend to need more resources. What we should not do while running system-wide validations is to repeat the checks we already did. We do not run the tests that already passed, and we try to keep those in this phase limited to what really matters (mostly integration and performance).

Why am I explaining the groups of tests we should run? The answer lies in the *system-wide validations*. Those are the tests that do not belong to an application, but to the pipelines in charge of deploying new releases to environments. We are about to explore one such pipeline, and we might need to add some tests.

## Exploring And Adapting The Staging Environment

Now that we saw the environments we have and their general purpose, let's explore what's inside them. We already saw that they are linked to Git repositories, so we'll clone them and check what's inside.



If you are inside the *go-demo-6* or any other repository, please move to the parent directory by executing `cd ..` command.

Let's clone the *environment-jx-rocks-staging* repository that contains the definition of our staging environment.



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

```
1 GH_USER=[...]
2
3 git clone \
4   https://github.com/$GH_USER/environment-jx-rocks-staging.git
5
6 cd environment-jx-rocks-staging
```

What do we have there?

```
1 ls -1
```

The output is as follows.

```
1 Jenkinsfile
2 LICENSE
3 Makefile
4 README.md
5 env
6 jenkins-x.yml
```

As you can see, there aren't many files in that repository. So, we should be able to explore them all relatively fast. The first one in line is Makefile.

```
1 cat Makefile
```

As you can probably guess by reading the code, the Makefile has targets used to build, install, and delete Helm charts. Tests are missing. Jenkins X could not know whether we want to run tests against applications in the staging environment and, if we are, which tests will be executed.

The staging environment is the place where all interconnected applications reside. That's the place where we deploy new releases in a production-like setting, and we'll see soon where that information about new releases is stored. For now, we'll focus on adding tests that will validate that a new release of any of the applications meets our system-wide quality requirements.

While you can run any type of tests when deploying to the staging environment, I recommend to keep them light. We'll have all sorts of tests specific to applications inside their pipelines. For example, we'll run unit tests, functional tests, and whichever other types of application-specific tests we might have. We should assume that the application works on the functional level long before it is deployed to staging. Having that in mind, all that's left to test in staging are cases that can be validated only when the whole system (or a logical and independent part of it) is up and running. Those can be integration, performance, or any other type of system-wide validations.

To run our tests, we need to know the addresses of the applications deployed to staging. Usually, that would be an easy thing since we'd use "real" domains. Our *go-demo-6* application could have a hard-coded domain *go-demo-6.staging.acme.com*. But, that's not our case since we're relying on dynamic domains assembled through a combination of the load

balancer IP and [nip.io](https://nip.io). Fortunately, it is relatively easy to find out the address by querying the associated Ingress.

Once we have the address, we could extend `jenkins-x.yml` by adding a step that would execute the tests we'd like to run. However, we won't do that just yet. We still need to explore Jenkins X pipelines in more detail.

## Understanding The Relation Between Application And Environment Pipelines

We experienced from the high level both an application and an environment pipeline. Now we might want to explore the relation to the two.

Keep in mind that I am aware that we did not yet go into details of the application pipeline (that's coming soon). Right now, we'll focus on the overall flow between the two.

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 did not yet go through different ways we can define webhook endpoints. What matters is that the webhook might initiate activity which 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 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 initiated through a Webhook of an application repository results in deployment, that change would not be stored in Git, and we could not say that **Git is the only source of truth**. Also, that deployment would **not be tracked**, the operation **would not be reproducible**, and everything **would not 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 all those issues by simply 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 we would not follow whichever coding practices we decided to follow (e.g., there would be no pull request). What matters is not only that we have to follow all the rules, but 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.

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

The output, limited to the relevant parts, is as follows.

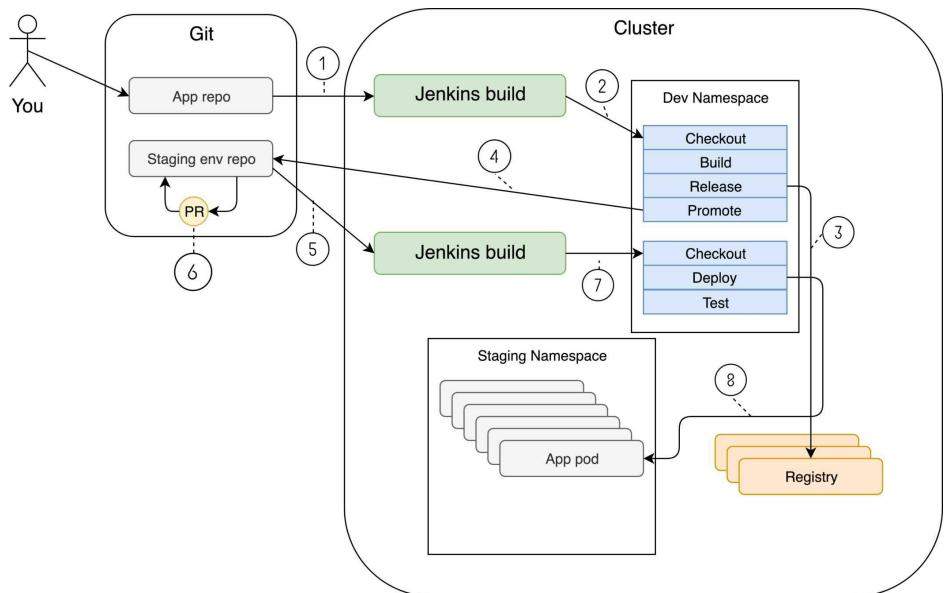
```
1 dependencies:  
2 ...  
3 - name: go-demo-6  
4   repository: http://jenkins-x-chartmuseum:8080  
5   version: 0.0.131
```

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 the 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.



**Figure 6-6: 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.

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 in <https://bit.ly/2Do5LRN>. You can also skip doing that since having production tests is 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.

What we did not yet explore is what happens when we have multiple applications. I believe there is 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 whether 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. On the other hand, we do not 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 which we'll explore later.

## Controlling The Environments

So far, we saw that Jenkins X created three environments during its installation process. We got the development environment that runs the tools we need for continuous delivery as well as temporary Pods used during builds. We also got the staging environment where all the applications are promoted automatically whenever we push a change to the master branch. Finally, we got the production environment that is still a mystery. Does all that mean that we are forced to use those three environments in precisely the way Jenkins X imagined? The short answer is no. We can create as many environments as we need, we can update the existing one, and we can delete them. So, let's start with the creation of a new environment.

```
1 jx create env \
2   --name pre-production \
3   --label Pre-Production \
4   --namespace jx-pre-production \
5   --promotion Manual \
6   --batch-mode
```

The arguments of the command should be self-explanatory. We just created a new Jenkins X environment called `pre-production` inside the Kubernetes Namespace `jx-pre-production`. We set its promotion policy to `Manual`, so new releases will not be installed on every push of the master branch of an application repository, but rather when we choose to promote it there.

If you take a closer look at the output, you'll see that the command also created a new GitHub repository, that it pushed the initial set of files, and that it created a webhook that will notify the system whenever we or the system pushes a change.

To be on the safe side, we'll list the environments and confirm that the newly created one is indeed available.

```
1 jx get env
```

The output is as follows.

1 NAME	2 LABEL	KIND	PROMOTE	NAMESPACE	REF	PR	ORDER	CLUSTER
3 dev	Development	Development	Never	jx		0		
4 pre-production	Pre-Production	Permanent	Manual	jx-pre-production	100			
5 staging	Staging	Permanent	Auto	jx-staging		100		
6 production	Production	Permanent	Manual	jx-production		200		

Just as we can create an environment, we can also delete it.

```
1 jx delete env pre-production
```

As you can see from the output, that command did not remove the associated Namespace. But, it did output the `kubectl delete` command we can execute to finish the job. Please execute it.

So, the `jx delete env` command will remove the references of the environment in Jenkins X, and it will delete the applications deployed in the associated Namespace. But, it does not remove the Namespace itself. That's not the only thing that it did not remove. The repository is still in GitHub. By now, you should already be used to the `hub` CLI. We'll use it to remove the last trace of the now non-existent environment.

```
1 hub delete -y \
2   $GH_USER/environment-jx-pre-production
```

That's it. We're done with the exploration of the environment. Or, to be more precise, we're finished with the environment with promotion policy set to `Auto`. Those set to `Manual` are coming soon.

Before we proceed, we'll go out of the `environment-jx-rocks-staging` directory.

```
1 cd ..
```

## Are We Already Following All The Commandments?

Before we take a break, let's see how many of the ten commandments are we following so far.

Everything we did on both the application and the environment level started with a push of a change to a Git repository. Therefore, **Git is our only source of truth**. Since everything is stored as code through commits and pushes, everything we do is **tracked and reproducible** due to **idempotency** of Helm and other tools. Changes to Git fire webhook requests that spin up one or more parallel processes in charge of performing the steps of our pipelines. Hence, communication between processes is **asynchronous**.

One rule that we do not yet follow fully is that **processes should run for as long as needed, but not longer**. We are only half-way there. Some of the processes, like pipeline builds, run in short-lived Pods that are destroyed when we're finished with our tasks. However, we still have some processes running even when nothing is happening. A good example is Jenkins. It is running while you're reading this, even though it is not doing anything. Not a single build is running there at this moment, and yet Jenkins is wasting memory and CPU. It's using resources for nothing and, as a result, we're paying for those resources for no apparent reason. We'll solve that problem later. For now, just remember that we are running some processes longer than they are needed.

Commandment number five says that **all binaries should be stored in registries**. We're already doing that. Similarly, **information about all the releases is stored in environment-specific repositories**, and we are **following the same coding practices** no matter whether we are making changes to one repository or the other, and no matter whether the changes are done by us or the machines.

Furthermore, all our **deployments are idempotent**, and we did NOT make any change to the system ourselves. **Only webhooks are notifying the system that the desired state should change**. That state is expressed through code pushed to Git repositories, sometimes by us, and sometimes by Jenkins X.

Finally, all the tools we used so far are **speaking with each other through APIs**.

1. ~~Git is the only source of truth.~~
2. ~~Everything must be tracked, every action must be reproducible, and everything must be idempotent.~~
3. ~~Communication between processes must be asynchronous.~~

4. Processes should run for as long as needed, but not longer.
5. ~~All binaries must be stored in registries.~~
6. ~~Information about all the releases must be stored in environment-specific repositories or branches.~~
7. ~~Everything must follow the same coding practices.~~
8. ~~All deployments must be idempotent.~~
9. ~~Git webhooks are the only ones allowed to initiate a change that will be applied to the system.~~
10. ~~All the tools must be able to speak with each other through APIs.~~

We're fulfilling all but one of the commandments. But, that does not mean that we will be done as soon as we can find the solution to make our Jenkins run only when needed. There are many more topics we need to explore, there are many new things to do. The commandments will only add pressure. Whatever we do next, we cannot break any of the rules. Our mission is to continue employing GitOps principles in parallel with exploring processes that will allow us to have cloud-native Kubernetes-first continuous delivery processes.

## What Now?

That's it. Now you know the purpose of the environments and how they fit into GitOps principles. We're yet to explore environments with the `Manual` promotion. As you'll see soon, the only significant difference between the `Auto` and `Manual` promotions is in actors.

By now, you should be familiar with what's coming next.

You might be planning to move into the next chapter right away. If that's the case, there are no cleanup actions to do. Just continue on to the next chapter.

However, if you created a cluster only for the purpose of the exercises we executed, please destroy it. We'll start the next, and each other chapter from scratch as a way to save you from running your cluster longer than necessary and pay more than needed to your hosting vendor. If you created the cluster or installed Jenkins X using one of the Gists from the beginning of this chapter, you'll find the instructions on how to destroy the cluster or uninstall everything at the bottom.

If you did choose to destroy the cluster or to uninstall Jenkins X, please remove the repositories we created as well as the local files. You can use the commands that follow for that.



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

```
1 rm -rf environment-jx-rocks-*
2
3 GH_USER=[...]
4
5 hub delete -y \
6   $GH_USER/environment-jx-rocks-staging
7
8 hub delete -y \
9   $GH_USER/environment-jx-rocks-production
10
11 rm -rf ~/.jx/environments/$GH_USER/environment-jx-rocks-*
```

# Improving And Simplifying Software Development

Software development is hard. It takes years to become a proficient developer, and the tech and the processes change every so often. What was effective yesterday, is not necessarily effective today. The number of languages we code in is increasing. While in the past, most developers would work in the same language throughout their whole career, today it is not uncommon for a developer to work on multiple projects written in different languages. We might, for example, work on a new project and code in Go, while we still need to maintain some other project written in Java. For us to be efficient, we need to install compilers, helper libraries, and quite a few other things.

No matter whether we write all the code in a single language or not, our applications will have different dependencies. One might need MySQL, while the other might use MongoDB as the data storage. We might also depend on applications developed by other teams working in parallel with us. No matter how good we become at writing mocks and stubs that replace those dependencies, eventually we'll need them running and accessible from our laptops. Historically, we've been solving those problems by having a shared development environment, but that proved to be inefficient. Sharing development environments results in too much overhead. We'd need to coordinate changes, and those that we make would often break something and cause everyone to suffer. Instead, we need each developer to have the option to have its own environment where dependencies required for an application are running.

For the dependencies to be useful, we should run them in (almost) the same way we're running them in production, that means we should deploy them to Kubernetes as well. For that, we can choose minikube or Docker Desktop if we prefer a local cluster, or get a segment (Namespace) of a remote cluster.

Unfortunately, compilers and dependencies are not everything we need to develop efficiently. We also need tools. Today that means that we need Docker or kaniko to build container images. We need `helm` and `kubectl` to

deploy applications to Kubernetes. We need `skaffold` that combines the process of building images with deployment. There are quite a few other tools specific to a language and a framework that would need to be installed and configured as well.

Even if we do set up all those things, we are still missing more. We need to be able to push and pull artifacts from container registry, ChartMuseum, Nexus, or any other registry that might be in use in our organization.

As you can imagine, installing and configuring all that is not trivial. It is not uncommon for a new hire to spend a week, or even more, on setting up its own development environment. And what happens if that person should move to a different project or if he should work on multiple projects in parallel?

We can continue with business as usual and install all the compilers and the tools on our laptops. We can dedicate time setting them up and connecting them with the system (e.g., with the registries). We can continue giving new hires long Word documents that walk them through all the actions they need to perform to be able to develop our applications. Or, we can take a different approach. We might be able to create a full-blown development environment on demand and for each person. We can even make those environments application specific. And we might be able to make it so fast and straightforward that anyone can do it with a single command and in only a couple of minutes.

Jenkins X allows us to spin up a project-based private development environment with all the tools, configurations, and environment variables we might need to work on any of our applications. That feature is called DevPod.

## **Exploring The Requirements Of Efficient Development Environment**

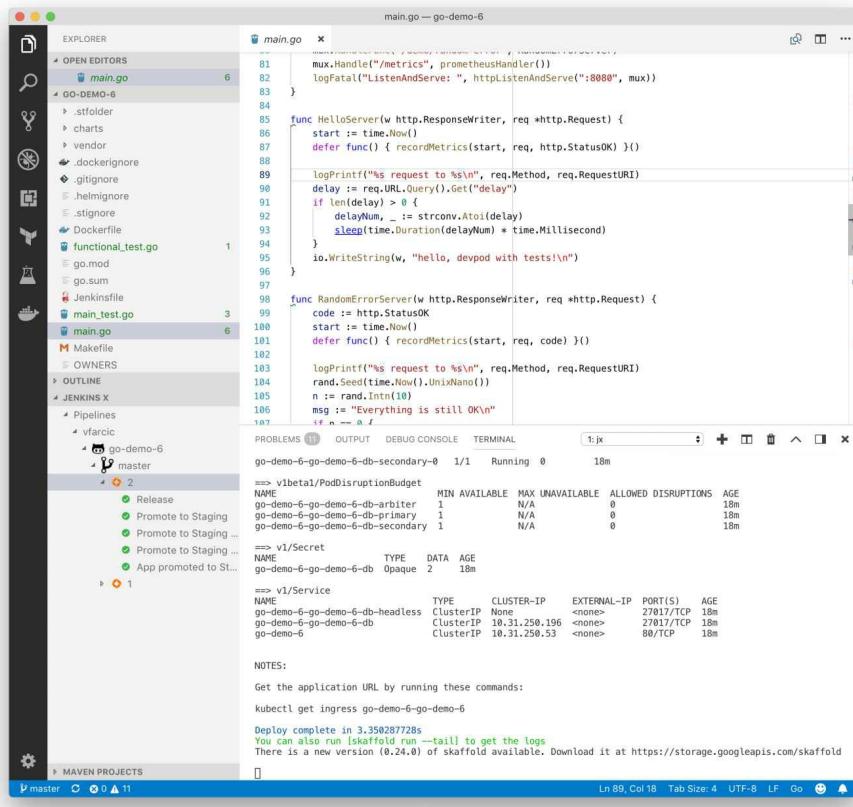
Let's discuss what we need from a development environment while taking into account the technology we have at our disposal.

In an ideal situation, we should be able to clone a repository of a project and execute a single command that would set up everything we need. That setup should be as close to the production environment as possible. Otherwise, we'd risk discrepancies between local results and those

deployed to staging, production, and other permanent environments. Since we need to cooperate with other people in our team, as well as with those working on other projects, such an environment would need to be connected with the registries from which we could pull container images and Helm charts. Finally, the development environment would need to have a compiler of the language used in the project, as well as all the tools we need (e.g., skaffold, Helm, Docker, etc.). As a bonus, it would be great if we would not need to run commands that build, test, and deploy our application. Every time we change a file, the environment itself should build a binary, run the tests, and deploy a new version. That way, we could concentrate on coding, while letting everything else happening in the background.

All in all, we need to be able to create a project-specific environment easily (with a single command) and fast, and such an environment needs to have everything we might need, without us installing or configuring (almost) anything on our laptop. The environment should automatically do all the tedious work like compilation, testing, and deployment. Wouldn't that be a very productive setup?

Here's how my development environment for the *go-demo-6* project looks like.



**Figure 7-1: Visual Studio Code With Jenkins X extension**

Everything I need is inside Visual Studio Code IDE. On the top-left side is the list of the project files. The bottom-left side contains all the Jenkins X activities related to the project. By clicking on a specific step, I can view the logs, open the application, open Jenkins, and so on.

The top-right corner contains the code, while the bottom-right is the terminal screen with useful output. Every time I make a change to any file of the project, the process in the terminal will build a new binary, execute tests, build container image, and deploy the new release. All that is happening inside the cluster and the only tools I use are Visual Studio Code and `jx`.

To create such a development environment, we'll need a Jenkins X cluster.

## Creating A Kubernetes Cluster With Jenkins X And Importing The Application

You know what to do. Create a new Jenkins X cluster unless you kept the one from before.



All the commands from this chapter are available in the [07-dev.sh](#) Gist.

For your convenience, the Gists from the previous chapter are available below as well.

- Create a new serverless **GKE** cluster: [gke-jx-serverless.sh](#)
- Create a new serverless **EKS** cluster: [eks-jx-serverless.sh](#)
- Create a new serverless **AKS** cluster: [aks-jx-serverless.sh](#)
- Use an **existing** serverless cluster: [install-serverless.sh](#)

We'll continue using the *go-demo-6* application. Please enter the local copy of the repository, unless you're there already.

```
1 cd go-demo-6
```



The commands that follow will reset your master branch with the contents of the buildpack branch that contains all the changes we did so far. Please execute them only if you are unsure whether you did all the exercises correctly.

```
1 git pull
2
3 git checkout buildpack-tekton
4
5 git merge -s ours master --no-edit
6
7 git checkout master
8
9 git merge buildpack
10
11 git push
```

If you restored the branch, the chances are that there is a reference to my user (`vfarcic`). We'll change that to Google project since that's what is the expected location of container images.



Please execute the commands that follow only if you are using **GKE** and if you ever restored a branch at the beginning of a chapter (like in the snippet above).

```
1 cat charts/go-demo-6/Makefile \
2   | sed -e \
3     "s@vfarcic@$PROJECT@g" \
4   | tee charts/go-demo-6/Makefile
5
6 cat charts/preview/Makefile \
7   | sed -e \
8     "s@vfarcic@$PROJECT@g" \
9   | tee charts/preview/Makefile
10
11 cat skaffold.yaml \
12   | sed -e \
13     "s@vfarcic@$PROJECT@g" \
14   | tee skaffold.yaml
```



If you destroyed the cluster at the end of the previous chapter, we'll need to import the *go-demo-6* application again. Please execute the commands that follow only if you created a new cluster specifically for the exercises from this chapter.

```
1 jx import --pack go --batch-mode
2
3 jx get activities \
4   --filter go-demo-6 \
5   --watch
```

Please wait until the activity of the application shows that all the steps were executed successfully, and stop the watcher by pressing *ctrl+c*.

Now we can explore how to leverage Jenkins X for our development environments.

## Creating a Remote Development Environment

Let's say that we want to change the *go-demo-6* code.

Once we finish making changes to the code, we'll need to compile it and test it, and that might require additional installations as well as configuration. Once our code is compiled, we'd need to run the application to confirm that it behaves as expected. For that, we need Docker to build a container image, and we need Helm to deploy it. We'd also need to create a personal Namespace in our cluster, or create a local one. Given that we adopted Skaffold as a tool that builds images and deploys them, we'd need to install it as well. But that's not the end of our troubles. Before we deploy our application, we'll need to push the new container image, as well as the Helm chart, to the registries running inside our cluster. To do

that, we need to know their addresses, and we need credentials with sufficient permissions.

Even when we do all that, we need a repeatable process that will build, test, release, and deploy our work whenever we make a significant change. Or, even better, whenever we make any change.

Here's the task for you. Change anything in the code, compile it, build an image, release it together with the Helm chart, deploy it inside the cluster, and confirm that the result is what you expect it to be. I urge you to stop reading and do all those things. Come back when you're done.

How much did it take you to have a new feature up-and-running? The chances are that you failed. Or, maybe, you used Jenkins X and the knowledge from the previous chapters to do all that. If that's the case, was it really efficient? I bet that it took more than a few minutes to set up the whole environment. Now, make another change to the code and do not touch anything else. Did your change compile? Were your tests executed? Was your change rolled out? Did that take more than a few seconds?

If you failed, we'll explore how to make you succeed the next time. If you didn't, we'll explore how to make the processes much easier, more efficient, and faster. We'll use Jenkins X, but not in the way you expect.

Please enter into the *go-demo-6* directory, if you're not already there.

```
1 cd go-demo-6
```

We'll create a whole development environment that will be custom tailored for the *go-demo-6* project. It will be the environment for a single user (a developer, you), and it will run inside our cluster. And we'll do that through a single command.

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

Right now, a Jenkins X DevPod is being created in the batch mode (no questions asked), and we can deduce what's happening from the output.

First, Jenkins X created the `jx-edit-YOUR_USER` Namespace in your cluster. That'll be your personal piece of the cluster where we'll deploy your changes to *go-demo-6*.

Next, Jenkins X installed the `ExposecontrollerService`. It will communicate with Ingress and make the application accessible for viewing and testing.

Further on, it updated the Helm repository in the DevPod so that we can utilize the charts available in ChartMuseum running inside the cluster.

It also ran Visual Studio Code. We'll keep it a mystery for now.

Finally, it cloned the `go-demo-6` code inside the Pod.

Many other things happened in the background. We'll explore them in due time. For now, assuming that the process finished, we'll enter inside the Pod and explore the newly created development environment.

```
1 jx rsh --devpod
```

The `jx rsh` command opens a terminal inside a Pod. The `-d` argument indicated that we want to connect to the DevPod we just created.

Before we proceed, we'll confirm that the `go-demo-6` code was indeed cloned inside the Pod.

```
1 cd go-demo-6
2
3 ls -l
```

The output is as follows.

```
1 charts
2 Dockerfile
3 functional_test.go
4 go.mod
5 jenkins-x.yml
6 main.go
7 main_test.go
8 Makefile
9 OWNERS
10 OWNERS_ALIASES
11 production_test.go
12 README.md
13 skaffold.yaml
14 vendor
15 watch.sh
```

Since we created the DevPod while inside our local copy of the `go-demo-6` repository, Jenkins X knew that we want to work with that code, so it cloned it for us.

Next, we should check whether our development environment indeed contains everything we need. Was Jenkins X intelligent enough to figure out needs just by knowing the repository we're using to develop our application? We can (partly) check that by trying to compile it.

```
1 make linux
```

We created the initial module definition with `go mod init` and executed `make linux` to compile the binary. It was a success, so we proved that, as a minimum, our new environment contains Go compiler.

Jenkins X pipelines use skaffold to create container images and deploy our applications. We won't go into all the details behind skaffold just yet, but only through the parts that matter for our current goals. So, let's take a quick look at `skaffold.yaml`.

```
1 cat skaffold.yaml
```

The output is as follows.

```
1 apiVersion: skaffold/v1beta2
2 kind: Config
3 build:
4   artifacts:
5     - image: vfarcic/go-demo-6
6       context: .
7       docker: {}
8   tagPolicy:
9     envTemplate:
10       template: '{{.DOCKER_REGISTRY}}/{{.IMAGE_NAME}}:{{.VERSION}}'
11   local: {}
12 deploy:
13   kubectl: {}
14 profiles:
15 - name: dev
16   build:
17     tagPolicy:
18       envTemplate:
19         template: '{{.DOCKER_REGISTRY}}/{{.IMAGE_NAME}}:{{.DIGEST_HEX}}'
20   local: {}
21 deploy:
22   helm:
23     releases:
24     - name: go-demo-6
25       chartPath: charts/go-demo-6
26       setValueTemplates:
27         image.repository: '{{.DOCKER_REGISTRY}}/{{.IMAGE_NAME}}'
28         image.tag: '{{.DIGEST_HEX}}'
```

This might be the first time you're going through `skaffold.yaml`, so let us briefly describe what we have in front of us.

The two main sections of `skaffold.yaml` are `build` and `deploy`. As their names indicate, the former defines how we build container images, while the latter describes how we deploy them.

We can see that images are built with `docker`. The interesting part of the `build` section is `tagPolicy.envTemplate.template` that defines the naming scheme for our images. It expects environment variables `DOCKER_REGISTRY` and `VERSION`.

The `deploy` section is uneventful and only defines `kubectl` as the deployment mechanism.

We won't be using root `build` and `deploy` sections of the config. They are reserved for the Jenkins pipeline. There are slight differences when building and deploying images to permanent and development environments. Our target is the `dev` profile. It overwrites the `build` and the `deploy` targets.

During the development builds, we won't be passing `VERSION`, but let it be autogenerated through skaffold's "special" variable `DIGEST_HEX`. Through it, every build will be tagged using a unique string (e.g., `27ffc7f...`). Unfortunately, as of this writing (May 2019), this variable is now deprecated and removed. We will have to replace it as there is no fix for this in Jenkins X's buildpacks yet.

```
1 cat skaffold.yaml \
2 | sed -e 's@DIGEST_HEX@UUID@g' \
3 | tee skaffold.yaml
```

We replaced the `DIGEST_HEX` with a `UUID`, which will have the same function and a similar format.

The `deploy` section of the `dev` profile changes the type from `kubectl` to `helm`. It sets the `chartPath` to `charts/go-demo-6` so that it deploys whatever we defined as the application chart in the repository. Further down, it overwrites `image.repository` and `image.tag` Helm values so that they match those of the image we just built.

The only unknown left in that YAML is the `DOCKER_REGISTRY`. It should point to the registry where we store container images. Should we define it ourselves? If we should, what is the address of the registry?

I already stated that DevPods contain everything we need to develop applications. So, it should come as no surprise that `DOCKER_REGISTRY` is already defined for us. We can confirm that by outputting its value.

```
1 echo $DOCKER_REGISTRY
```

The output will vary depending on the type of the Kubernetes cluster you're running and whether Docker registry is inside it or you're using a service. In GKE with the default setup it should be an IP and a port (e.g., `10.31.253.125:5000`), in AKS it should be the name of the cluster followed with `azurecr.io` (e.g.,

`THE_NAME_OF_YOUR_CLUSTER.azurecr.io`), and in EKS it is a combination of a unique ID ending with `amazonaws.com` (e.g., `036548781187.dkr.ecr.us-west-2.amazonaws.com`). The exact address does not matter since all we need to know is that it is stored in the environment variable `DOCKER_REGISTRY`.

Speaking of variables, it might be useful to know that many others were created for us.

```
1 env
```

The output should display over a hundred variables. Some of them were created through Kubernetes services, while others were injected through the process of creating the DevPod. I'll let you explore them yourself. We'll be using some of them soon.

Next, we should initialize Helm client residing in the DevPod so that we can use it to deploy charts.

```
1 kubectl create \n2     -f https://raw.githubusercontent.com/vfarcic/k8s-specs/master/helm/tiller-rbac\n3 ml \
4     --record --save-config\n5\n6 helm init --service-account tiller
```

Is> We are using `tiller` only to simplify the development. For a more secure cluster, you should consider using Helm with `tiller` (server-side Helm) by executing `helm template` command.

Now we're ready to build and deploy our application in the personal development environment. As Skaffold does not generate the `DIGEST_HEX` nor our replacement `UUID`, we will have to create one before we run the

dev profile. So we will prefix our skaffold run with `export UUID=$(uuidgen)`.

```
1 export UUID=$(uuidgen)
2
3 skaffold run --profile dev
```

We run skaffold using the `dev` profile.

If you inspect the output, you'll see that quite a few things happened. It built a new container image using the *go-demo-6* binary we built earlier. Afterward, it installed the application chart. But, where was that chart installed? We could find that out from the output, but there is an easier way. The Namespace where Skaffold installs applications is also defined as an environment variable.

```
1 echo ${SKAFFOLD_DEPLOY_NAMESPACE}
```

In my case, the output is `jx-edit-vfarcic` (yours will use a different user). That's the personal Namespace dedicated to my development of the *go-demo-6* application. If I'd work on multiple projects at the same time, I would have a Namespace for each. So, there will be as many `jx-edit-*` Namespaces as there are developers working in parallel, multiplied with the number of projects they are working on. Of course, those Namespaces are temporary, and we should delete them together with DevPods once we're finished working on a project (or when we're ready to go home). It would be a waste to keep them running permanently. We'll keep our DevPod for a while longer so that we can explore a few other goodies it gives us.

Let's confirm that the application and the associated database were indeed installed when we executed `skaffold run`.

```
1 kubectl -n ${SKAFFOLD_DEPLOY_NAMESPACE} \
2     get pods
```

The output is as follows.

```
1 NAME                           READY STATUS   RESTARTS AGE
2 exposecontroller-service-...    1/1   Running  0        16m
3 go-demo-6-go-demo-6-...        1/1   Running  3        4m
4 go-demo-6-go-demo-6-db-arbiter-0 1/1   Running  0        4m
5 go-demo-6-go-demo-6-db-primary-0 1/1   Running  0        4m
6 go-demo-6-go-demo-6-db-secondary-0 1/1   Running  0        4m
```

I hope that you can already see the benefits of using DevPod. If you are, you'll be pleased that there's quite a lot left to discover and quite a few benefits we did not yet explore.

When we import a project, or when we create a new one using one of the quickstarts, one of the files created for us is `watch.sh`. It is a simple yet handy script that combines the commands we run so far, and it adds a twist you can probably guess from its name. Let's take a look what's inside.

```
1 cat watch.sh
```

The output is as follows.

```
1 #!/usr/bin/env bash
2
3 # watch the java files and continuously deploy the service
4 make linux
5 skaffold run -p dev
6 reflex -r "\.go\$" -- bash -c 'make linux && skaffold run -p dev'
```

We can see that the first two lines (excluding misplaced comments) are the same as those we executed. It's building the binary (`make linux`) and executing the same `skaffold` we run. The last line is the one that matters.

As you can see, it is missing the `UUID` variable, so let's add that to the `watch.sh`

```
1 cat watch.sh | sed -e \
2   's@skaffold@UUID=$\{uuidgen\} skaffold@g' \
3   | tee watch.sh
```

[Reflex](#) is a nifty tool that watches a directory and reruns commands when specific files change. In our case, it'll rerun `make linux && skaffold run -p dev` whenever any of the `.go` files are changed. That way, we'll build a binary and a container image, and we'll deploy a Helm chart with that image every time we change any of the source code files ending with `.go`. In other words, we'll always have the application running the latest code we're working on. Isn't that nifty?

Let's try it out.

```
1 chmod +x watch.sh
2
3 nohup ./watch.sh &
```

To be on the safe side, we assigned executable permissions to the script before we executed it. Since we used `nohup` in the second command, it'll run even if we end our session, and `&` will make sure that the watcher is running in the background. Please press the enter key to go back to the terminal.

We run the script in the background so that we can execute a few commands that will validate whether everything works as expected. The downside is that we won't see the output of the script. We'll fix that later when we start working with multiple terminals. For now, please note that the script will run `make linux` and `skaffold run -p dev` commands every time we change any of the `.go` files.

Now, let's exit the DevPod and confirm that what's happening inside it indeed results in new builds and installations every time we change our Go source code.

```
1 exit
```

Before we change our code, we'll confirm that the application indeed runs. For that, we'll need to know the address `exposecontroller` assigned to it. We could do that by retrieving JSON of the associated Ingress, or we can introduce yet another `jx` helper command. We'll go with the former.

```
1 jx get applications
```

The output is as follows.

```
1 APPLICATION EDIT      PODS URL                      ST
2 ING PODS URL
3 go-demo-6   SNAPSHOT 1/1  http://go-demo-6.jx-edit-vfarcic.35.196.94.247.nip.io 0.
4 159 1/1  http://go-demo-6.jx-staging.35.196.94.247.nip.io
```

We listed all the applications installed through Jenkins X. For now, there's only one, so the output is rather sparse.

We can see that the `go-demo-6` application is available as a `SNAPSHOT` as well as a specific release in the `STAGING` environment (e.g., `0.0.159`). `SNAPSHOT` is the release running in our personal development environment (DevPod). Please copy its `URL`, and paste it instead of `[...]` in the command that follows.

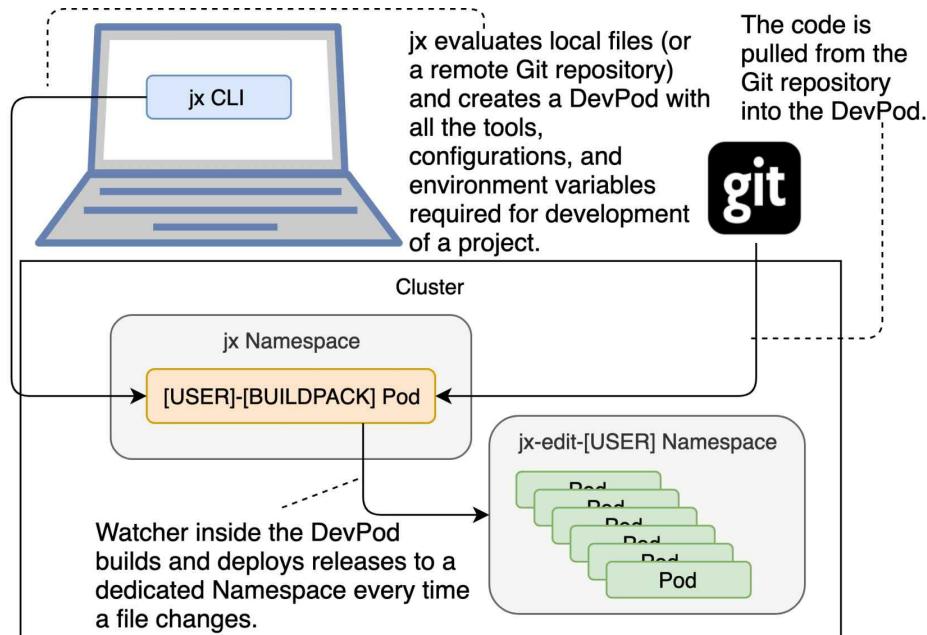
```
1 URL=[...]
2
```

```
3 curl "$URL/demo/hello"
```

The result of the `curl` command should output `hello, world!` thus confirming that we are still running the initial version of the application. Our next mission is to change the source code and confirm that the new release gets deployed in the personal development environment.



If the output is HTML with `503 Service Temporarily Unavailable`, you were too fast, and you did not give the process enough time. Please wait for a few moments until the application is up-and-running, and repeat the `curl` command.



**Figure 7-2: The process of creating a DevPod and building and deploying applications from inside it**

Now we need to figure out how to change the source code running inside the DevPod so that the process of building and deploying is repeated. As you will soon see, there is more than one way to do that.

## Working With The Code In The DevPod Using Browser-Based IDE

We could go back to the DevPod and modify the code from a terminal. We could use `vi` or a similar editor for that. While I do use terminal editors quite often, I find them sub-optimum when working on a project. I believe that `vi`, `emacs`, `nano`, and similar editors are useful when working on

individual scripts, but not that great when working on a full-fledged project. Call me lazy, but I need an IDE like Visual Studio Code, IntelliJ, Eclipse, or something similar. I need syntax highlighting, code complete, the ability to jump into a function with a single click, and other goodies provided by IDEs.

The problem is that the code we're interested in is in a DevPod running inside our cluster. That means that we need to synchronize our local files from a laptop to the DevPod or we can work with the code remotely. For now, we're interested in the latter option (we'll explore the former later). If we are to work with remote files, and we are not (yet) going to synchronize files between our laptop and the DevPod, the only available option is to use a remote IDE (unless you want to stick to `vi` or some other terminal-based editor).

If you remember, I already stated a couple of times that Jenkins X hopes to give you everything you might need to develop your applications. That even includes an IDE. We only need to figure out where it is or, to be more precise, how to access it. We'll do that by introducing yet another `jx` command.

```
1 jx open
```

The output is as follows.

```
1 NAME          URL
2 deck          http://deck.jx.34.206.148.101.nip.io
3 hook          http://hook.jx.34.206.148.101.nip.io
4 jenkins-x-chartmuseum http://chartmuseum.jx.34.206.148.101.nip.io
5 tide          http://tide.jx.34.206.148.101.nip.io
6 vfarcic-go-ide http://vfarcic-go-ide.jx.34.206.148.101.nip.io
7 vfarcic-go-port-2345 http://vfarcic-go-port-2345.jx.34.206.148.101.nip.io
8 vfarcic-go-port-8080 http://vfarcic-go-port-8080.jx.34.206.148.101.nip.io
```

The `open` command lists all the applications managed by Jenkins X and running inside our cluster. We can see that one of them is `ide` prefixed with our username and the programming language we're using. In my case that's `vfarcic-go-ide`.

If we add the name of the application as an argument to the `jx open` command, it'll (surprise, surprise) open that application in the default browser. Let's try it out.

Please replace [...] with the name of the `*-ide` application before executing the command that follows.

```
1 jx open [...]
```

What you see in front of you is Visual Studio Code. It is a browser-based IDE (it can run as a desktop app as well). It is, in my experience, the best browser-based IDE today (March 2019). If you've already used Visual Studio Code on your laptop, what you see should feel familiar. If you haven't, it's intuitive and straightforward, and you'll have no problem adopting it (if you think it's useful).

Let's give Visual Studio Code a spin.

Our next mission is to modify a few Go files and observe that the changes are build and deployed without us executing any additional commands. Remember that the watcher (`watch.sh`) is still running.

Please open the *Files* section located in the left-hand menu, expand the `go-demo-6` directory, and double-click the `main.go` file to open it in the main body of the IDE. Next, change `hello, world` (or whatever else you changed it to previously) to `hello, devpod`.

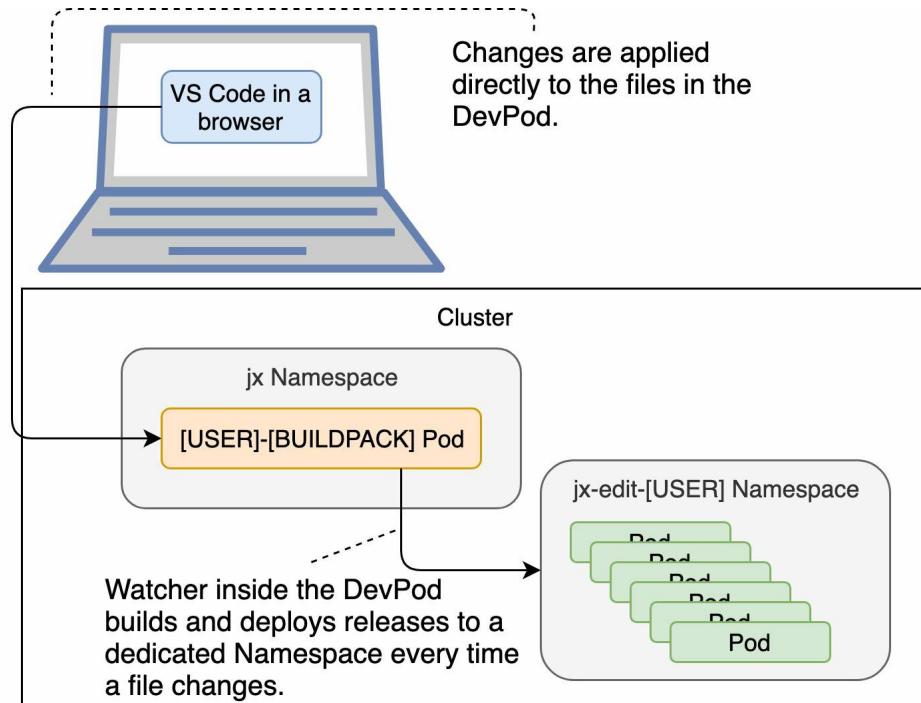
Since we have tests that validate that the correct message is returned, we'll need to change them as well. Open the `main_test.go` file next, search for `hello, world` (or whatever else you changed it to previously), and change it to `hello, devpod`.

Make sure to save the changes.

Now we can confirm that our changes are automatically built and deployed every time we change a Go source code.

```
1 curl "$URL/demo/hello"
```

The output should be `hello, devpod!`



**Figure 7-3: The process of using Visual Studio Code to modify files inside a DevPod and building and deploying applications**

We saw how we can work using personal development environments and modifying them from a browser.

I used Visual Studio Code quite a few times. It is beneficial when we do not have all the tools running inside our laptops (except `jx` CLI). But, a browser-based editor might not be your cup of tea. You might find a desktop IDE easier and faster. Or, maybe you are emotionally attached to a desktop version of Visual Studio Code, IntelliJ, or whatever else is your coding weapon of choice. Fear not, we can use them as well. Our next mission is to connect your favorite IDE with DevPods. But, before we do that, we'll delete the DevPod we're currently running and start a new one with a twist.

```
1 jx delete devpod
```

Please type `y` when asked whether you want to delete the DevPods and press the enter key.

The DevPod is no more.

The problem is that we did not push the code changes to GitHub. If we did that, we could pull them to our local hard disk. Since we forgot that vital

step, our changes were lost the moment we deleted the DevPod. That was silly of me, wasn't it? Or maybe I did that intentionally just to show you that we can also synchronize files from your laptop into the DevPod, and vice versa.

## Synchronizing Code From A Laptop Into A DevPod

I hope that you liked the idea of using a browser-based IDE like Visual Studio Code. On the other hand, the chances are that you believe that it might be useful in some scenarios, but that the bulk of your development will be done using desktop-based IDE. In other words, I bet that you prefer to work with local files. If that's the case, we need to figure out how to sync them with DevPod. But, before we do that, we'll add a critical component to our development process. We're missing tests, and that is, as I'm sure you already know, unacceptable.

Given that we are using Makefile to specify our targets (at least when working with Go), that's the place where we'll add unit tests. I assume that you want to run unit tests every time you change your code and that you'll leave slower types of tests (e.g., functional and integration tests) to Jenkins. If that's not the case, you should have no problem extending our examples to run a broader set of validations.



Remember what we said before about `Makefile`. It expects tabs as indentation. Please make sure that the command that follows is indeed using tabs and not spaces, if you're typing the commands instead of copying and pasting from the Gist.

```
1 echo 'unittest:  
2     CGO_ENABLED=$(CGO_ENABLED) $(GO) \  
3     test --run UnitTest -v  
4 ' | tee -a Makefile
```

We added a `unittest` target with `go test` command limited to functions that contain `UnitTest` in their names.

Next, we need to modify `watch.sh` so that it executes the new target.

```
1 cat watch.sh |  
2     sed -e \  
3     's@linux \&\& skaffold@linux \&\& make unittest \&\& skaffold@g' \  
4     | sed -e \  
5     's@linux \&\& skaffold@linux \&\& make unittest \&\& skaffold@g' |
```

```
5      's@skaffold@UUID=$ (uuidgen) skaffold@g' \
6      | tee watch.sh
```

Now that we added unit tests both to `Makefile` and `watch.sh`, we can go back to our original objective and figure out how to synchronize local files with those in a DevPod.

We'll use [ksync](#). It transparently updates containers running inside a cluster from a local checkout. That will enable us to use our favorite IDE to work with local files that will be synchronized with those inside the cluster.

To make things simpler, `jx` has its own implementation of `ksync` that will connect it with a DevPod. Let's fire it up.

```
1 jx sync --daemon
```

We executed `jx sync` in `daemon` mode. That will allow us to run it in the background instead of blocking a terminal session.

It'll take a few moments until everything is up and running. The final message should state that it looks like '`ksync watch`' is already running so we don't need to run it yet... When you see it, you'll know that it is fully operational, and all that's left is to press `ctrl+c` to go back to the terminal session. Since we specified `--daemon`, `ksync` will continue running in the background.

Now we can create yet another DevPod. This time, however, we'll add `--sync` argument. That will give it a signal that we want to use `ksync` to synchronize our local file system with the files in the DevPod.

```
1 jx create devpod \
2   --label go \
3   --sync \
4   --batch-mode
```

Now we need to repeat the same commands as before to start the watcher inside the DevPod. However, this time we will not run it in the background since it might be useful to see the output in case one of our tests fail and we might need to apply a fix before we proceed with the development. For that reason, we'll open a second terminal. I recommend that you resize two terminals so that both occupy half of the screen. That way you can see them both.

Open a second terminal session.



If you are using EKS, you'll need to recreate the environment variables `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, and `AWS_DEFAULT_REGION`. Otherwise, your second terminal will not be able to authenticate against Kube API.

Next, we'll enter the DevPod and execute the same commands that will end with running the `watch.sh` script.

```
1 jx rsh --devpod
2
3 unset GOPATH
4
5 go mod init
6
7 helm init --client-only
8
9 chmod +x watch.sh
10
11 ./watch.sh
```

Now that `watch.sh` is running in the foreground, we can see the results of building, testing, and deploying development releases created every time we change our source code.

The last time we modified the files in the DevPod, we did not push them to Git. Since we did not have synchronization, they were lost when we deleted the Pod. Let's confirm that we are still at square one by sending a request to the application.

Please return to the first terminal.

```
1 curl "$URL/demo/hello"
```

The output is `hello, world!` thus confirming that our source code is indeed intact and that the watcher did its job by deploying a new release based on the original code. If the output is `hello, devpod!`, the new deployment did not yet roll out. In that case, wait for a few moments and repeat the `curl` command.

Next, we'll make a few changes to the files on our laptop.

```
1 cat main.go | sed -e \
2   's@hello, world@hello, devpod with tests@g' \
3   | tee main.go
```

```

4
5 cat main_test.go | sed -e \
6   's@hello, world@hello, devpod with tests@g' \
7   | tee main_test.go

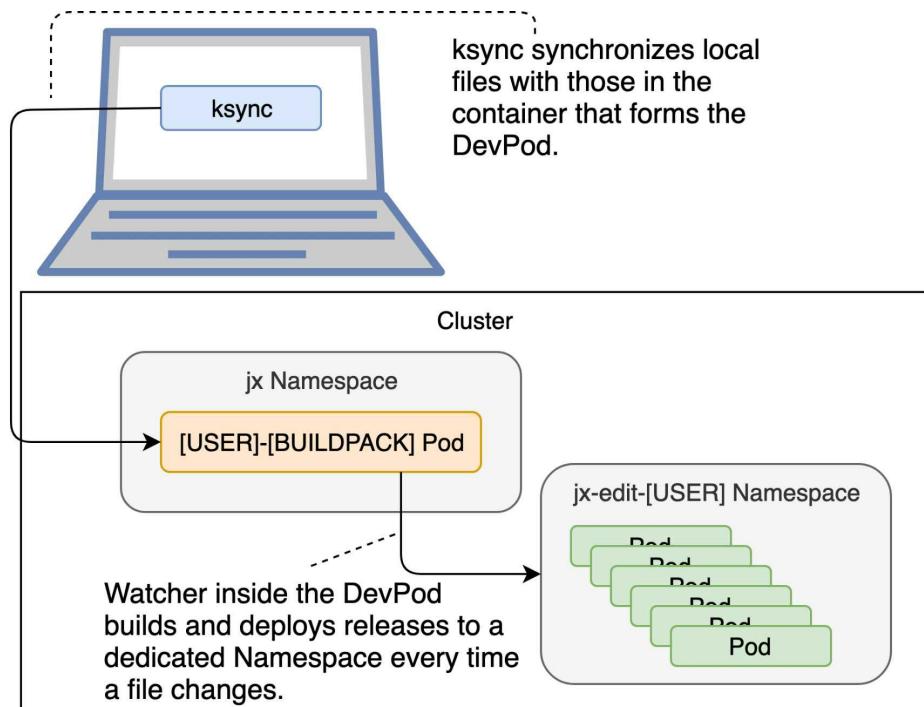
```

Since we changed the file, and if you are quick, we should see the result of the new iteration of `watch.sh`. Please go back to the second terminal. In there, you should see that the binary is built, that the unit tests are executed, and that skaffold built a new image and upgraded the development release using Helm.

Now that we observed that the process run through yet another iteration, we can send a request to the application and confirm that the new release indeed rolled out. Please go to the first terminal to execute the `curl` command that follows.

```
1 curl "$URL/demo/hello"
```

This time, the output is `hello, devpod with tests!`. From now on, every time we change any of the local Go files, the process will repeat. We will be notified if something (e.g., tests) fail by the output from the second terminal. Otherwise, the application running in our personal environment (Namespace) will always be up-to-date.



**Figure 7-4: The process of using ksync to synchronize local files with those in the container that forms the DevPod**

Next, we'll imagine that we continued making changes to the code until the new feature is done. The only thing left is to push them back to the GitHub repository. We'll ignore the fact that we should probably make a pull request (explanation is coming in the next chapter), and push directly to the master branch.

```
1 git add .
2
3 git commit \
4   --message "devpod"
5
6 git push
```

You should be familiar with the rest of the process. Since we pushed a change to the master branch, Jenkins will pick it up and run all the steps defined in the pipeline. As a result, it will deploy a new release to the staging environment. As always, we can monitor the activity of the Jenkins build.

```
1 jx get activity \
2   --filter go-demo-6 \
3   --watch
```

The new release should be available in the staging environment once all the steps succeeded and we can cancel the activity watcher by pressing *ctrl+c*.

Let's take another look at the available applications.

```
1 jx get applications
```

The output should be the same as before. However, this time we're interested in URL of the staging environment since that's where the new release was rolled out after we pushed the changes to the master branch.

Please copy the URL of the staging release (the second one) and paste it instead of [...] in the commands that follow.

```
1 STAGING_URL=[...]
2
3 curl "$STAGING_URL/demo/hello"
```

As expected, the output is `hello, devpod with tests!` thus confirming that the new release (the same one we have in the private development environment) is now rolled out to staging.

We're done with our DevPod combined with ksync synchronization so we can delete it from the system. We're not going to make any more changes to the code, so there is no need for us to waste resources on the DevPod.

```
1 jx delete devpod
```

Please press `y` followed with the enter key to confirm the deletion.

There's still one more development-related topic we should explore.

## Integrating IDEs With Jenkins X

We saw that we can work with local files and let ksync synchronize them with DevPod. That allows us to work with our favorite IDE while reaping the benefits of having a full environment somewhere else. But that's not all. We can integrate IDEs with Jenkins X.

Do you remember the screenshot from the beginning of the chapter? That was Visual Studio Code setup I used when working on the examples in this book. It's repeated below in case you're forgetful and you do not want to go back to see it again.

**Figure 7-5: Visual Studio Code With Jenkins X extension**

Let's see if we can configure it in the same way on your laptop.



I'll show how to integrate Visual Studio Code with Jenkins X. If you prefer IntelliJ instead, the steps are very similar and I hope you will not have trouble setting it up. On the other hand, if you do not like either of the two, you're in tough luck since those are the only ones supported so far (March 2019). Nevertheless, I strongly recommend Visual Studio Code, and I urge you to try it out. It is my favorite and, in my opinion, it is superior to any other IDE.



I could not configure Jenkins X extension to Visual Studio Code to work with EKS. As far as I know, there is no place where we can define the environment variables required for authentication. Truth be told, I did not try too hard since I tend to work with GKE most of the time. I would appreciate if you let me know if you solve that issue.

If you do not already use Visual Studio Code, but you do want to try it out, please download it and install it from the [official site](#).

With Visual Studio Code up-and-running, navigate to *File > Open*, select the *go-demo-6* directory, and click the *Open* button.

Next, we need to install the *jx* extension. To do that, go to *View > Extensions*, search for *jx*, click the *Install* button in *jx-tools*. Once it is installed, click the *Reload* button.

Now that the extension is installed, we can display some useful information opening *View > Explorer*, and expanding the *JENKINS X* tab.

You will see a few general options by clicking the ... button. Feel free to explore them.

More interesting information is available in the *Pipelines* section. Inside it should be your user. Expand it, and you'll see *go-demo-6*, with the *master* branch inside. Inside the master branch, you should see the builds executed for that project. If you expand one of them, you should notice the same steps as those we see when we execute `jx get activity` command. Some of them provide additional actions by clicking the right mouse button or two-finger click on Mac. They should be self-explanatory, so I'll let you explore them on your own.

Finally, I would typically open a terminal session inside Visual Studio Code. You can do that by opening *View > Integrated Terminal*. Inside that terminal, I would execute the command that would create a DevPod for that project.

Now you have everything in one place. You can write your Code and see the Jenkins X activities as well as the output of `watch.sh` running in a DevPod. Isn't that awesome?

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

```
1 cd ..
```

## What Now?

We're done with yet another chapter, and you are once again forced to decide whether to continue using the cluster or to destroy it. If the

destruction is what you crave for, you'll find the instructions at the bottom of the Gist you chose at the beginning of this chapter.

If you destroyed the cluster or you uninstalled Jenkins X, please remove the repositories and the local files we created. You can use the commands that follow for that.



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

```
1 GH_USER=[...]
2
3 hub delete -y \
4   $GH_USER/environment-jx-rocks-staging
5
6 hub delete -y \
7   $GH_USER/environment-jx-rocks-production
8
9 rm -rf ~/.jx/environments/$GH_USER/environment-jx-rocks-*
```

# Working With Pull Requests And Preview Environments

Pull Requests (or whatever their equivalents are called in your favorite Git distribution) are a norm. Most of us have adopted them as the primary way of reviewing and accepting changes that will ultimately be deployed to production. They work hand-in-hand with feature branches.



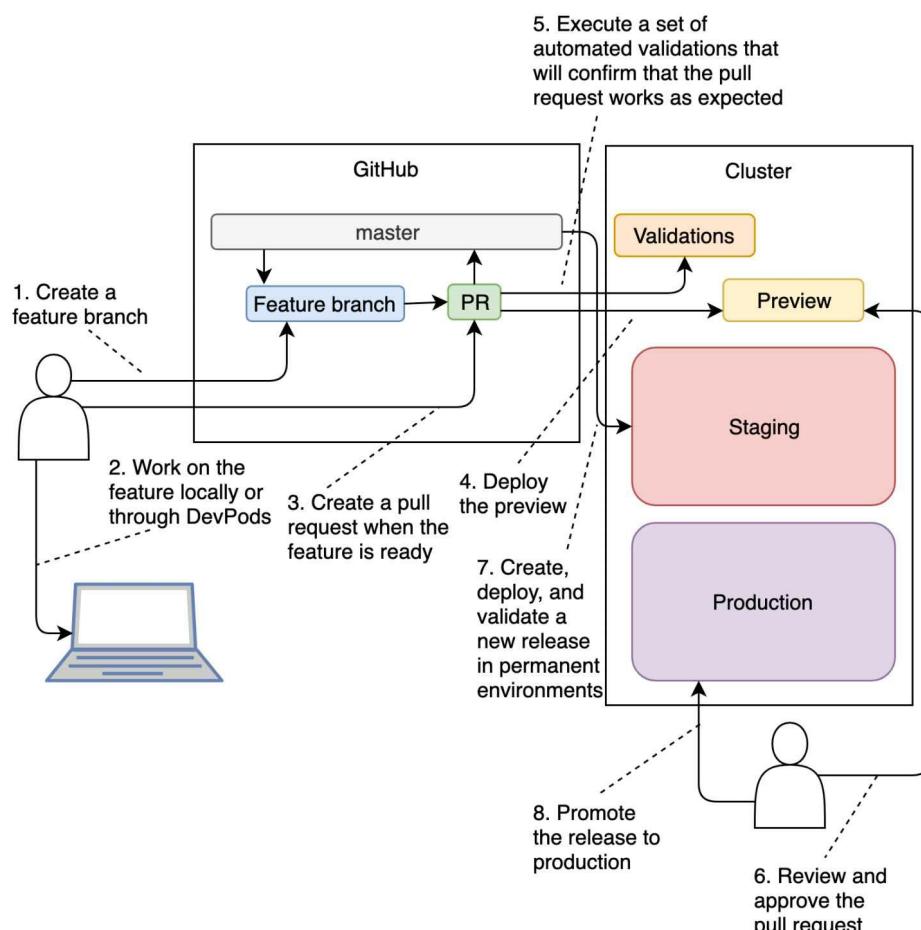
Some of you might cringe thinking about pull requests. You might prefer trunk-based development, and work directly on master. Even if that is your preference, I still recommend you go through this chapter and do all the exercises. If processes were manual, getting pull requests merged would be a slow, hands-on, and time-consuming effort. If you use pull requests, you should use automation to speed up the process, and reduce the human effort required. That's precisely what Jenkins X is doing for us. So put your reservations of pull requests aside, and follow along.

A common (and recommended) workflow is to create a new branch for each feature or change we want to release. Once we create a branch dedicated to a feature or change, we either work locally or in DevPods until we are satisfied with the outcome. From there on, we make a pull request, which should execute a set of automated steps that will deploy the preview and validate it. If all the steps are successful, we might have some manual actions like code review. Once finished, we merge the pull request and that, as you already saw, results in yet another round of automated tests that end with the deployment to one or more environments (those set to receive new releases automatically). Finally, the last step is to promote a release to production whenever we feel we're ready (unless the promotion to production is set to be automatic). Hopefully, the whole process from creating a branch all the way until it is deployed to permanent environments (e.g., staging, production) is measured in days or even hours.

So, a high-level process of a lifecycle of an application usually contains the steps that follow.

1. Create a feature branch

2. Work on the feature locally or through DevPods
3. Create a pull request when the feature is ready
4. Deploy the preview
5. Execute a set of automated validations that will confirm that the pull request works as expected
6. Review and approve the pull request
7. Create, deploy, and validate a new release in permanent environments (e.g., staging)
8. Promote the release to production (unless that part is automated as well)



**Figure 8-1: The high-level process of a lifecycle of an application**

In your case, there might be variations to the process. Still, at a very high level, the process works reasonably well and is widely adopted. The problem is that the process based on feature branches is in stark contrast to how we were developing applications in the past.

*A long time ago in a galaxy far, far away*, we used to have long application lifecycles, cumbersome and mostly manual processes, and an infinite number of gates with silly approval mechanisms. That reflected in our branching strategies. We'd have project or development branches that lived for months. Moving from one environment to another usually meant merging from one branch to another (e.g., from development to staging to integration to pre-production to production). We do not live in 1999 anymore, and those practices are today obsolete.

We split projects into features. We reduced lifecycles from months to weeks to days to hours. And, more importantly, we learned that it is pointless to test one set of binaries and deploy another to production. All that resulted in the feature branches model. Each feature gets a branch, and each branch is merged back to master. There's nothing in between. There are no staging, integration, pre-production and other branches. The reason for that lies in the process that tells us that we should build something only once and move the same artifact through environments. With such an approach, there is no reason for the existence of all those branches. You develop in a feature branch, you merge it to the master, you build the artifacts as part of the merge process, and you move them through environments until they reach production. Even that can be questioned, and many are now pushing directly to the master branch without feature or any other branches and without pull requests. We won't go that far, and I'll assume that, if you do want to push directly to master, you should be able to adapt the process we'll use. What I do NOT expect you to do is create a complicated branching schema only because you're used to it. Move on and travel forward from whichever year you live into the present time (2019 at the time of this writing).

We already explored how to *work on a feature locally or through DevPods*. In this chapter, we'll cover all the other steps except for the promotion to production (that's coming later). We'll go through most of the lifecycle of an application and see how we can add pull requests into the GitOps and Jenkins X processes we explored so far.

## **Creating A Kubernetes Cluster With Jenkins X And Importing The Application**

This part is getting boring, but it might be necessary if you destroyed the cluster or uninstalled Jenkins X at the end of the previous chapter. Long

story short, create a new Jenkins X cluster unless you kept the one from before.



All the commands from this chapter are available in the [08-pr.sh](#) Gist.

For your convenience, the Gists from the previous chapter are available below as well.

- Create a new serverless **GKE** cluster: [gke-jx-serverless.sh](#)
- Create a new serverless **EKS** cluster: [eks-jx-serverless.sh](#)
- Create a new serverless **AKS** cluster: [aks-jx-serverless.sh](#)
- Use an **existing** serverless cluster: [install-serverless.sh](#)

We'll continue using the *go-demo-6* application. Please enter the local copy of the repository, unless you're there already.

```
1 cd go-demo-6
```



The commands that follow will reset your `master` with the contents of the `dev` branch that contains all the changes we did so far. Please execute them only if you are unsure whether you did all the exercises correctly.

```
1 git pull
2
3 git checkout dev-tekton
4
5 git merge -s ours master --no-edit
6
7 git checkout master
8
9 git merge dev-tekton
10
11 git push
```

If you ever restored a branch at the beginning of a chapter, the chances are that there is a reference to my user (`vfarcic`). We'll change that to Google project since that's what Knative will expect to be the location of the container images.



Please execute the commands that follow only if you are using **GKE** and if you ever restored a branch at the beginning of a chapter (like in the snippet above).

```
1 cat charts/go-demo-6/Makefile \
2   | sed -e \
3     "s@vfarcic@$PROJECT@g" \
4   | tee charts/go-demo-6/Makefile
5
6 cat charts/preview/Makefile \
7   | sed -e \
8     "s@vfarcic@$PROJECT@g" \
9   | tee charts/preview/Makefile
10
11 cat skaffold.yaml \
12   | sed -e \
13     "s@vfarcic@$PROJECT@g" \
14   | tee skaffold.yaml
```



If you destroyed the cluster at the end of the previous chapter, you'll need to import the *go-demo-6* application again. Please execute the commands that follow only if you created a new cluster specifically for the exercises from this chapter.

```
1 jx import --batch-mode
2
3 jx get activities \
4   --filter go-demo-6 \
5   --watch
```

Please wait until the activity of the application shows that all the steps were executed successfully, and stop the watcher by pressing *ctrl+c*.

Now we can create some pull requests.

## Creating Pull Requests

What is the first thing a developer should do when starting to work on a new feature? If that developer is used to the “old processes”, he might wait until someone creates a set of branches, a few environments, infinite approvals, and a bunch of other silly things no one should need today. We already saw that creating a development environment is very easy and fast with DevPods. I’m going to ignore talking about the approvals, so the only thing left are branches.

Today, no self-respecting developer needs others to create branches. Since we are not developing against a common “development” branch any more but using feature branches, we can create one ourselves and start developing a new feature.

```
1 git checkout -b my-pr
```

We created a branch called `my-pr`.

We’ll skip the steps that create a personal project-specific development environment with DevPods. That would lead us astray from the main topic. Instead, we’ll use the commands that follow to make a few simple changes. Our goal here is not to develop something complex, but rather to make just enough changes to help us distinguish the new from the old release while moving it through the process of creating a pull request and a preview environment.

So, we’ll change our `hello` message to something else. I’ll save you from opening your favorite IDE and changing the code yourself by providing a few simple `cat`, `sed`, and `tee` commands that will do the changes for us.

```
1 cat main.go | sed -e \  
2   "s@hello, devpod with tests@hello, PR@g" \  
3   | tee main.go  
4  
5 cat main_test.go | sed -e \  
6   "s@hello, devpod with tests@hello, PR@g" \  
7   | tee main_test.go
```

We changed the code of the application and the tests so that the output message is `hello, PR` instead of `hello, devpod with tests`. Those changes are not much different than the changes we did in the previous chapters. The only notable difference is that this time we’re not working with the `master`, but rather with a feature branch called `my-pr`.

There’s one more change we’ll make.

Right now, MongoDB is defined in `charts/go-demo-6/requirements.yaml` and it will run as a replica set. We do not need that for previews. A single replica DB should be more than enough. We already added a non-replicated MongoDB as a dependency of the preview (`charts/preview/requirements.yaml`) when we created a custom build pack. Since we have the DB defined twice (once in the app chart, and once in the preview), we’d end up with two DBs installed. We don’t need that,

so we'll disable the one defined in the application chart and keep only the one from the preview chart (the single replica DB).

To speed things up, we'll also disable persistence of the DB. Since previews are temporary and used mostly for testing and manual validations, there is no need to waste time creating a persistent volume.

Please execute the command that follows.

```
1 echo "
2
3 db:
4   enabled: false
5
6 preview-db:
7   persistence:
8     enabled: false" \
9 | tee -a charts/preview/values.yaml
```

The next steps should be no different than what you're already doing every day at your job. We'll commit and push the changes to the upstream.

```
1 git add .
2
3 git commit \
4   --message "This is a PR"
5
6 git push --set-upstream origin my-pr
```

What comes next should also be something you're (hopefully) doing all the time. We'll create a pull request. But we might do it slightly differently than what you're used to.

Typically, you would open GitHub UI and click a few buttons that would create a pull request. If you prefer UIs and the “common” way of creating PRs, please do so. On the other hand, if you'd like to do it using CLI, you'll be glad to know that `jx` allows us to create PRs as well. If a terminal is your weapon of choice, please execute the command that follows to create a PR.

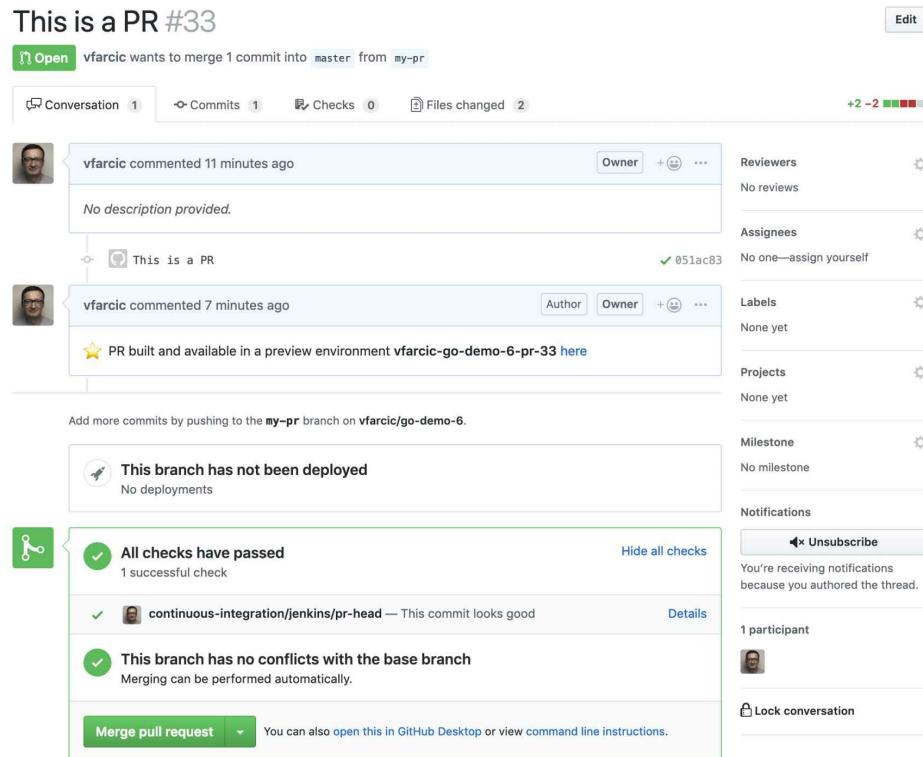
```
1 jx create pullrequest \
2   --title "My PR" \
3   --body "This is the text that describes the PR
4 and it can span multiple lines" \
5   --batch-mode
```

The output should show the ID of the pull request as well as a link to see the pull request in a browser. Please open the link.

When the build started, Jenkins X notified GitHub that the validation is in progress and you should be able to confirm that from the GitHub UI. If you do not see any progress, the build probably did not yet start, and you might need to refresh your screen after a while. Similarly, when the build is finished and if it is successful, we should see a new comment stating that the *PR* is *built and available in a preview environment [USER]-go-demo-6-pr-[PR\_ID]*. Bear in mind that you might need to refresh your browser to see the change of the pull request status.

What that comment tells us is that Jenkins X created a new environment (Namespace) dedicated to that PR and it will keep it up-to-date. If we push additional changes to the pull request, the process will repeat, and it will end with a new deployment to the same environment.

The end of the comment is a link (*here*). You can use it to open your application. Feel free to click the link and do not be confused if you see *503* or a similar error message. Remember that the *go-demo-6* application does not respond to requests to the root and that you need to add */demo/hello* to the address opened through the *here* link.



**Figure 8-2:** A pull request with checks completed and a PR built and available

If you are like me and you prefer CLIs, you might be wondering how to retrieve similar information from your terminal. How can we see which previews we have in the cluster, what are the associated pull requests, in which Namespaces they are running, and what are the addresses we can use to access the previews?

Just like `kubectl get` command allows you to retrieve any Kubernetes resource, `jx get` does the same but limited to the resources related to Jenkins X. We just need to know the name of the resource. The one we're looking for should be easy to guess.

```
1 jx get previews
```

The output is as follows.

```
1 PULL REQUEST                                NAMESPACE          APPLICATION
2 https://github.com/vfarcic/go-demo-6/pull/33 jx-vfarcic-go-demo-6-pr-33 http://go-
3 mo-6.jx-vfarcic-go-demo-6-pr-33.35.196.59.141.nip.io
```

We can see from the output the address of the pull request in GitHub, the Namespace where the application and its dependencies were deployed, and the address through which we can access the application (through auto-generated Ingress).

Next, we'll confirm that the preview was indeed installed and that it is working by sending a simple request to `/demo/hello`.



Before executing the commands that follow, please make sure to replace [...] with the address from the APPLICATION column from the output of the previous command.

```
1 PR_ADDR=[...]
2
3 curl "$PR_ADDR/demo/hello"
```

The output should reflect the changes to the code we made before creating the pull request. It should display `hello, PR!`

The critical thing to understand is that every pull request will be deployed to its own environment unless we change the default behavior.

I will let you “play” a bit with `jx` CLI. Explore the logs of the pull request, output the activities, and run the other commands we learned so far.

## Intermezzo

If you go through MongoDB Helm chart values, you’ll notice that one of them is `usePassword`. All we have to do is set it to `false`, both for previews as well as for the “official” application chart deployed to permanent environments. We’ll start with the latter by taking a quick look at the chart’s `values.yaml` file.

```
1 cat charts/go-demo-6/values.yaml
```

The output, limited to the relevant parts, is as follows.

```
1 ...
2 go-demo-6-db:
3   replicaSet:
4     enabled: true
```

We already have a set of values dedicated to `go-demo-6-db`. It’s the alias of the `mongodb` dependency we defined earlier in `requirements.yaml`. All we have to do is add `usePassword: false`.

```
1 echo "
2   usePassword: false" \
3   | tee -a charts/go-demo-6/values.yaml
```

We’ll repeat the same in the preview Chart’s `values.yaml` file.

Please execute the command that follows.

```
1 echo "
2   usePassword: false" \
3   | tee -a charts/preview/values.yaml
```

We’re done with changes, and we should push them to GitHub.

```
1 git add .
2
3 git commit \
4   --message "Removed MongoDB password"
5
6 git push
```

The only thing left is to wait until the Jenkins build initiated by the push to the pull request is finished. This time it should be successful, and I’ll leave it to you to confirm that it is indeed green. Feel free to observe the

outcome from the PR screen in GitHub or through one of the `jx` commands.

## Merging a PR

Now that we finished all the changes we wanted to make and that the PR is validated, we should probably let others in our team review the changes so that knowledge is spread. We'll skip that part because I assume you know how to review code, but also because we'll explore that in more detail later when we discuss GitChat principles. So, we'll skip code review and proceed with the merge of the pull request to the master branch.

Please open the pull request screen in GitHub (unless you already have it in front of you) and click *Merge pull request*, followed with the *Confirm merge* button.

You already know what happens when we push or merge code into the master branch, so we'll just relax for a moment or two while watching the activity of the newly initiated build.

```
1 jx get activity \
2   --filter go-demo-6 \
3   --watch
```

Please press *ctrl+c* to stop watching the progress when all the steps of the new build are reported as `Succeeded`.

The only thing left is to confirm that the new release was indeed deployed correctly to the staging environment. First, we'll list the applications managed by Jenkins X.

```
1 jx get applications
```

The output is as follows.

```
1 APPLICATION STAGING PODS URL
2 go-demo-6 0.0.169 1/1 http://go-demo-6.jx-staging.35.196.59.141.nip.io
```

Next, we'll send a request to the application in the staging environment to confirm that the output is indeed correct.



Make sure to replace [...] with the address from the URL column from the previous output before executing the commands that follow.

```
1 STAGING_ADDR=[...] # Replace ` [...]` with the URL
2
3 curl "$STAGING_ADDR/demo/hello"
```

The output should be the `hello, PR!` message.

Even though we merged the pull request to the master branch, we did not yet see all the phases. We still have some cleaning to do.

Before we move on, we'll checkout the master branch and pull the latest version of the code.

```
1 git checkout master
2
3 git pull
```

## Exploring Jenkins X Garbage Collection

It would be silly if each preview environment would run forever. That would be a waste of resources. So, we need to figure out how to remove everything deployed as part of the pull request process for those that are already merged or closed. We could manually delete previews, but that would result in too much human involvement in something repetitive and predictable. We could create a script that would do that for us, and that would be executed periodically (e.g., cronjob). But, before we do that, we should check whether Jenkins X already created some Kubernetes CronJobs.

```
1 kubectl get cronjobs
```

The output is as follows.

```
1 NAME                  SCHEDULE      SUSPEND ACTIVE  LAST   SCHEDULE AGE
2 jenkins-x-gcactivities 0/30 */* * False    0      1h        3h
3 jenkins-x-gcpods       0/30 */* * False    0      1h        3h
4 jenkins-x-gcpreviews   0 */3 */* * False    0      2h        3h
```

At the time of this writing (March 2019), there are three Jenkins X CronJobs. All three are garbage collectors (`gc`) that are executed periodically (every three hours). The names of the CronJobs should give

you an idea of their purpose. The first removes old Jenkins X activities (`jenkins-x-gcactivities`), the second deals with completed or failed Pods (`jenkins-x-gcpods`), while the third eliminates preview environments associated with merged or closed pull requests (`jenkins-x-gcpreviews`).

Given that we have a CronJob that deletes completed pull requests, the most we would wait before they are removed from the system is three hours. If that's too long (or too short) of a wait, we can change the schedule of the CronJob to some other value. In some cases, we might not want to wait at all. For that, we have a `jx` command that runs the same process as the CronJob. But, before we run it, we'll confirm that the preview is indeed still available.

```
1 jx get previews
```

The output is as follows.

```
1 PULL REQUEST                                NAMESPACE          APPLICATION
2 https://github.com/vfarcic/go-demo-6/pull/33 jx-vfarcic-go-demo-6-pr-33 http://go-
3 mo-6...nip.io
```

If in your case the output is empty, it means that an iteration of the CronJob was executed since you merged the pull request and, as a result, the preview was removed from the cluster. If that's not the case (if the preview is indeed still available), we'll run garbage collector manually.

```
1 jx gc previews
```

The output is as follows.

```
1 Deleting helm release: jx-vfarcic-go-demo-6-pr-34
2 Deleting preview environment: vfarcic-go-demo-6-pr-34
3 Deleted environment vfarcic-go-demo-6-pr-34
```

We can see from the output that both the Helm chart of the preview as well as the Namespace were removed. We can confirm that by listing the previews one more time.

```
1 jx get previews
```

This time the output should be empty.

In some other cases, we might want to remove only a specific preview and leave the others intact. We could have used the `jx delete preview`

command for that.

In most cases, there is no need for us to run Jenkins X garbage collectors manually. That's what the CronJobs are for. I showed you that it can be done, but that does not mean that it should. I'm sure you have more exciting things to do than to waste your time removing unused applications, given that the system already does that for us.

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

```
1 cd ..
```

## What Now?

There are still quite a few other features related to pull requests that we did not yet explore. Specifically, we did not dive into GitChat-related functionalities. The reason for that is technical. The type of Jenkins X installation we performed does not allow GitChat. We'll fix that later on. For now, you should have a solid base to deal with your pull request.

We're done with yet another chapter, and you are once again forced to decide whether to continue using the cluster or to destroy it. If the destruction is what you crave for, you'll find the instructions at the bottom of the Gist you chose at the beginning of this chapter.

If you destroyed the cluster or you uninstalled Jenkins X, please remove the repositories and the local files we created. You can use the commands that follow for that.



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

```
1 GH_USER=[...]
2
3 hub delete -y \
4   $GH_USER/environment-jx-rocks-staging
5
6 hub delete -y \
7   $GH_USER/environment-jx-rocks-production
8
9 rm -rf ~/.jx/environments/$GH_USER/environment-jx-rocks-*
```

# Promoting Releases To Production

We reached the final stage, at least from the application lifecycle point of view. We saw how to import an existing project and how to create a new one. We saw how to develop build packs that will simplify those processes for the types of applications that are not covered with the existing build packs or for those that deviate from them. Once we added our app to Jenkins X, we explored how it implements GitOps processes through environments (e.g., staging and production). Then we moved into the application development phase and explored how DevPods help us to set a personal application-specific environment that simplifies the “traditional” setup that forced us to spend countless hours setting it on our laptop and, at the same time, that avoids the pitfalls of shared development environments. Once the development of a feature, a change, or a bug fix is finished, we created a pull request, we executed automatic validations, and we deployed the release candidate to a PR-specific preview environment so that we can check it manually as well. Once we were satisfied with the changes we made, we merged it to the master branch, and that resulted in deployment to the environments set to receive automatic promotions (e.g., staging) as well as in another round of testing. Now that we are comfortable with the changes we did, all that’s left is to promote our release to production.

The critical thing to note is that promotion to production is not a technical decision. By the time we reach this last step in the software development lifecycle, we should already know that the release is working as expected. We already gathered all the information we need to make a decision to go live. Therefore, the choice is business related. “When do we want our users to see the new release?” We know that every release that passed all the steps of the pipeline is production-ready, but we do not know when to release it to our users. But, before we discuss when to release something to production, we should decide who does that. The actor will determine when is the right time. Does a person approve a pull request, or is it a machine?

Business, marketing, and management might be decision-makers in charge of promotion to production. In that case, we cannot initiate the process

when the code is merged to the master branch (as with the staging environment), and that means that we need a mechanism to start the process manually through a command. If executing a command is too complicated and confusing, it should be trivial to add a button (we'll explore that through the UI later). There can also be the case when no one makes a decision to promote something to production. Instead, we can promote each change to the master branch automatically. In both cases, the command that initiates the promotion is the same. The only difference is in the actor that executes it. Is it us (humans) or Jenkins X (machines)?

At the moment, our production environment is set to receive manual promotions. As such, we are employing continuous delivery that has the whole pipeline fully automated and requires a single manual action to promote a release to production. All that's left is to click a button or, as is our case, to execute a single command. We could have added the step to promote to production to Jenkinsfile, and in that case, we'd be practicing continuous deployment (not delivery). That would result in a deployment of every merge or push to the master branch. But, we aren't practicing continuous deployment today, and we'll stick with the current setup and jump into the last stage of continuous delivery. We'll promote our latest release to production.

## Creating A Kubernetes Cluster With Jenkins X And Importing The Application

If you kept the cluster from the previous chapter, you can skip this section. Otherwise, we'll need to create a new Jenkins X cluster.



All the commands from this chapter are available in the [09-promote.sh](#) Gist.

For your convenience, the Gists from the previous chapter are available below as well.

- Create a new serverless **GKE** cluster: [gke-jx-serverless.sh](#)
- Create a new serverless **EKS** cluster: [eks-jx-serverless.sh](#)
- Create a new serverless **AKS** cluster: [aks-jx-serverless.sh](#)
- Use an **existing** serverless cluster: [install-serverless.sh](#)

We'll continue using the *go-demo-6* application. Please enter the local copy of the repository, unless you're there already.

```
1 cd go-demo-6
```



The commands that follow will reset your `master` with the contents of the `pr` branch that contains all the changes we did so far. Please execute them only if you are unsure whether you did all the exercises correctly.

```
1 git pull
2
3 git checkout pr-tekton
4
5 git merge -s ours master --no-edit
6
7 git checkout master
8
9 git merge pr-tekton
10
11 git push
```



Please execute the commands that follow only if you are using **GKE** and if you ever restored a branch at the beginning of a chapter (like in the snippet above).

```
1 cat charts/go-demo-6/Makefile \
2   | sed -e \
3     "s@vfarcic@$PROJECT@g" \
4   | tee charts/go-demo-6/Makefile
5
6 cat charts/preview/Makefile \
7   | sed -e \
8     "s@vfarcic@$PROJECT@g" \
9   | tee charts/preview/Makefile
10
11 cat skaffold.yaml \
12   | sed -e \
13     "s@vfarcic@$PROJECT@g" \
14   | tee skaffold.yaml
```



If you destroyed the cluster at the end of the previous chapter, you'll need to import the *go-demo-6* application again. Please execute the commands that follow only if you created a new cluster specifically for the exercises from this chapter.

```
1 jx import --batch-mode
2
3 jx get activities \
4   --filter go-demo-6 \
5   --watch
```

Please wait until the activity of the application shows that all the steps were executed successfully, and stop the watcher by pressing *ctrl+c*.

Now we can promote our last release to production.

## Promoting A Release To The Production Environment

Now that we feel that our new release is production-ready, we can promote it to production. But, before we do that, we'll check whether we already have something running in production.

```
1 jx get applications --env production
```

The output states that no applications **were** found in environments production.

How about staging? We must have the release of our *go-demo-6* application running there. Let's double check.

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

The output is as follows.

```
1 APPLICATION STAGING PODS URL
2 go-demo-6 0.0.184 1/1 http://go-demo-6.jx-staging.35.237.161.58.nip.io
```

For what we're trying to do, the important piece of the information is the version displayed in the STAGING column.



Before executing the command that follows, please make sure to replace [...] with the version from the STAGING column from the output of the previous command.

```
1 VERSION=[...]
```

Now we can promote the specific version of *go-demo-6* to the production environment.

```
1 jx promote go-demo-6 \
2   --version $VERSION \
3   --env production \
4   --batch-mode
```

It'll take a minute or two until the promotion process is finished.

The command we just executed will create a new branch in the production environment (*environment-jx-rocks-production*). Further on, it'll follow the same practice based on pull requests as the one employed in anything else we did so far. It'll create a PR and wait until a Jenkins X build is finished and successful. You might see errors stating that it failed to query the Pull Request. That's normal. The process is asynchronous, and `jx` is periodically querying the system until it receives the information that confirms that the pull request was processed successfully.

Once the pull request is processed, it'll be merged to the master branch, and that will initiate yet another Jenkins X build. It'll run all the steps we defined in the repository's `Jenkinsfile`. By default, those steps are only deploying the release to production, but we could have added additional validations in the form of integration or other types of tests. Once the build initiated by the merge to the master branch is finished, we'll have the release running in production and the final output will state that `merge status checks all passed so the promotion worked!`

The process of manual promotion (e.g., production) is the same as the one we experienced through automated promotions (e.g., staging). The only difference is who executes promotions. Those that are automated are initiated by application pipelines pushing changes to Git. On the other hand, manual promotions are triggered by us (humans).

Next, we'll confirm that the release is indeed deployed to production by retrieving all the applications in that environment.

```
1 jx get applications --env production
```

The output is as follows.

```
1 APPLICATION PRODUCTION PODS URL
2 go-demo-6 0.0.184 1/1 http://go-demo-6.jx-production.35.237.161.58.nip.io
```

In my case, the output states that there is only one application (`go-demo-6`) running in production and that the version is `0.0.184`.

To be on the safe side, we'll send a request to the release of our application running in production.



Before executing the commands that follow, please make sure to replace [...] with the URL column from the output of the previous command.

```
1 PROD_ADDR=[...]
2
3 curl "$PROD_ADDR/demo/hello"
```

The output should be the familiar message `hello, PR!`. We confirmed that promotion to production works as expected.

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

```
1 cd ..
```

## What Now?

This was a very short chapter. Wasn't it? There is not much more to learn about manual promotions. All we have to do is execute a single command.

Bear in mind that there are many other things we could do to improve deployment to production. We could add HorizontalPodAutoscaler that will scale the application depending on memory and CPU consumption or based on custom metrics. We could also add additional tests beyond those we added in the [Applying GitOps Principles](#) chapter. We won't work any of the many improvements we could do since I assume that you already know how to enhance your Helm charts and how to write tests and add their execution to Jenkinsfile. What matters is that the process works and it is up to you to change it and enhance it to suit your specific needs.

We'll conclude that we explored the whole lifecycle of an application and that our latest release is running in production.

Now you need to decide whether to continue using the cluster or to destroy it. If you choose to destroy it or to uninstall Jenkins X, you'll find the instructions at the bottom of the Gist you chose at the beginning of this chapter.

If you destroyed the cluster or you uninstalled Jenkins X, please remove the repositories and the local files we created. You can use the commands that follow for that.



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

```
1 GH_USER=[...]
2
3 hub delete -y \
4   $GH_USER/environment-jx-rocks-staging
5
6 hub delete -y \
7   $GH_USER/environment-jx-rocks-production
8
9 rm -rf ~/.jx/environments/$GH_USER/environment-jx-rocks-*
```

# Versioning Releases

Versioning is one of those things that can be done in many different ways. Almost every team I worked with came up with their own versioning schema. When starting a new project, quite often we would spend time debating how we are going to version our releases. And yet, coming up with our own versioning schema is usually a waste of time. The goals of versioning are simple. We need a unique identifier of a release as well as an indication of whether a change breaks backward compatibility. Given that others already agreed on the format that fulfills those objectives, the best we can do, just as with many other things related to software development, is to use the convention. Otherwise, we are probably wasting our time reinventing the wheel without understanding that a few things are likely going to go wrong down the line.

First of all, our users might need to know whether our release is production ready and whether it breaks compatibility with the previous releases. To be honest, most users do not even want to know that, and they will merely expect our applications always to work, but for now, we'll assume that they do care about those things. Somebody always does. If the end users don't, the internal ones (e.g., engineers from the same or from other teams) do. They will assume that you, just as most of the others, follow one of the few commonly accepted naming schemes. If you do, your users should be able to deduce the readiness and backward compatibility through a simple glance at the release ID, and without any explanation how your versioning works.

Just as users expect specific versioning scheme, many tools expect it as well. If we focus on the out-of-the-box experience, tools can implement only a limited number of variations, and their integration expectations cannot be infinite. If we take Jenkins X as an example, we can see that it contains a lot of assumptions. It assumes that every Go project uses Makefile, that versioning control system is one of the Git flavors, that Helm is used to package Kubernetes applications, and so on. Over time, the assumptions are growing. For example, Jenkins X initially assumed that Docker is the only way to build container images, and kaniko was added to the list later. In some other cases, none of the assumptions will fit

your use case, and you will have to extend the solution to suit your needs. That's OK. What is not a good idea is to confuse specific needs with those generated by us not following a commonly adopted standard for no particular reason. Versioning is often one of those cases.

By now, you can safely assume that I am in favor of naming conventions. They help us have a common understanding. When I see a release identifier, I assume that it is using semantic versioning simply because that is the industry standard. If most of the applications use it, I should not be judged by assuming that yours are using it as well, unless you have a good reason not to. If that's the case, make sure that it is indeed justified to change a convention that is good enough for most of the others in the industry. In other words, if you do not use a commonly accepted versioning scheme, you are a type of the company that is hoping to redefine a standard, or you are just silly for no particular reason (other than not knowing the standards).

So, what are the commonly used and widely accepted versioning schemes? The answer to that question depends on who the audience is. Are we creating a release for humans or machines? If it's the former, are they engineers or the end users?

We'll take a quick look at some of the software I'm running. That might give us a bit of insight into how others treat versioning, and it might help us answer a few of the questions.

I'm writing this on my MacBook running macOS *Mojave*. My Pixel 3 phone is running *Android Pie*. My "playground" cluster is running Jenkins X *Next Generation* (we did not explore it yet). The list of software releases with "strange" names can go on and on, and the only thing that they all have in common is that they have exciting and intriguing names that are easy to remember. Mojave, Pie, and Next Generation are names good for marketing purposes because they are easy to remember. For example, macOS Mojave is more catchy and easier to memorize than macOS 10.14.4. Those are releases for end users that probably do not care about details like a specific build they're running. Those are not even releases, but rather significant milestones.

For technical users, we need something more specific and less random, so let's take a look at the release identifiers of some of the software we are already using.

I am currently running `kubectl version v1.13.2`, and my Kubernetes cluster is `1.12.6-gke.10`. My Helm client is at the version `v2.12.1` (I'm not using tiller). I have `jx CLI` version `1.3.1074` and Git `2.17.2`. What do all those versions have in common? They all contain three numbers, with an optional prefix (e.g., `v`) or a suffix (e.g., `-gke.10`). All those tools are using semantic versioning. That's the versioning engineers should care about. That's the versioning most of us should use.

## Semantic Versioning Explained

Semantic versioning is very easy to explain and leaves very little to interpretation. We have three numbers called *major*, *minor*, and *patch*. If we increment *minor*, *patch* is reset to zero (or one). Similarly, if *major* is incremented, both the *minor* and the *patch* are set to zero (or one). Which number is incremented depends on the type of the change we're releasing.

Given a version number `MAJOR.MINOR.PATCH`, increment each of the segments using the rules that follow.

- *PATCH* is incremented when we release bug fixes.
- *MINOR* is incremented when new functionality is added in a backward-compatible manner.
- *MAJOR* is incremented when changes are not backward compatible.

If, for example, our application has an API, incrementing the major version would be a clear signal to our users that they would need to adapt or continue using the previous (older) major version (assuming we keep both, as we should). A change in the minor version would mean that users do not need to adapt, even though there are new features included in the release. All other cases would increment only the patch version.

Deducing which version to increment for an application that does not contain an API is a bit harder. When a change to a client-side web application is backward compatible or not largely depends on how humans (the only users of a web app) perceive backward compatibility. If, for example, we change the location of the fields in the login screen, we are likely backward compatible. However, if we add a new required field, we are not and, therefore, we should increase the major version. That might sound confusing and hard to figure out for non-API based applications. But there is a more reliable way if we have automated testing. If one of the

existing tests fail, we either introduced a bug, or we made a change that is not backward compatible and therefore needs to increment major version.

I believe that semantic versioning should be used (almost) always. I cannot think of a reason why we shouldn't. It contains a clear set of rules that everyone can apply, and it allows everyone to know which type of change is released. However, I've seen too many times teams that do not want to follow those rules. For example, some want to increment a number at the end of each sprint. Others want to use dates, quarters, project numbers, or any other versioning system they are used to. I believe that in most of those cases the problem is in accepting change and not a "real" need for custom versioning. We tend to be too proud of what we did, even when the rest of the industry tells us that its time to move on.

The problem with using our own versioning schema is in expectations. New developers joining your company likely expect semantic versioning since it is by far the most commonly used. Developers working on applications that communicate with your application expect semantic versioning because they need to know whether you released a new feature and whether it is backward compatible. The end users (e.g., people visiting your Web site) generally do not care about versioning, so please don't use them as an excuse to "reinvent the wheel".

I would say that there two significant reasons why you should use semantic versioning. First of all, something or someone depends on your application. Those who depend on it need to know when an incompatibility is introduced (a change of the major version). Even if that's not the case, the team developing the application should have an easy way to distinguish types of releases.

Now, you might say that you do not care for any of the reasons for using semantic versioning, so let me give additional motivation. Many of the tools you are using expect it. That's the power of conventions. If most of the teams use semantic versioning, many of the tools that interact with releases will assume it. As a result, by choosing a different versioning schema, you might not be able to benefit from "out-of-the-box" experience. Jenkins X is one of those tools. It assumes that you do want to use semantic versioning because that is the most commonly used schema. We'll see that in practice soon.

Before we proceed, I must make it clear that I am not against you using some other versioning schema. Please do if you have a very good reason for it and if the benefits outweigh the effort you might need to invest in overcoming hurdles created by not using something that is widely adopted. Coming up with your own way to do stuff is truly great, as long as the reason for deviation is based on the hope that you can do something better than others, not only because you like it more. When we do things differently for no good reason, we are likely going to pay a potentially high price later in maintenance and investment that you'll have to make in tweaking the tools you're using. The need to comply with standards and conventions is not unique to versioning, but to almost everything we do. So, the short version of what I'm trying to say is that *you should use standards and conventions unless you have a good reason not to*. What that means is that you should be able to defend why you are not using an industry-defined convention, and not to seek a reason why you should jump into something most of the others already adopted.

That was enough of theory. We'll explore through practical hands-on examples how Jenkins X helps us to version our releases.

## Creating A Kubernetes Cluster With Jenkins X And Importing The Application

If you kept the cluster from the previous chapter, you can skip this section. Otherwise, we'll need to create a new Jenkins X cluster.



All the commands from this chapter are available in the [10-versioning.sh](#) Gist.

For your convenience, the Gists from the previous chapter are available below as well.

- Create a new serverless **GKE** cluster: [gke-jx-serverless.sh](#)
- Create a new serverless **EKS** cluster: [eks-jx-serverless.sh](#)
- Create a new serverless **AKS** cluster: [aks-jx-serverless.sh](#)
- Use an **existing** serverless cluster: [install-serverless.sh](#)

We'll continue using the *go-demo-6* application. Please enter the local copy of the repository, unless you're there already.

```
1 cd go-demo-6
2
3 git checkout master
```



Please execute the commands that follow only if you are using **GKE** and if you ever restored a branch at the beginning of a chapter (like in the snippet above).

```
1 cat charts/go-demo-6/Makefile \
2   | sed -e \
3     "s@vfarcic@$PROJECT@g" \
4   | tee charts/go-demo-6/Makefile
5
6 cat charts/preview/Makefile \
7   | sed -e \
8     "s@vfarcic@$PROJECT@g" \
9   | tee charts/preview/Makefile
10
11 cat skaffold.yaml \
12   | sed -e \
13     "s@vfarcic@$PROJECT@g" \
14   | tee skaffold.yaml
```



If you destroyed the cluster at the end of the previous chapter, you'll need to import the *go-demo-6* application again. Please execute the commands that follow only if you created a new cluster specifically for the exercises from this chapter.

```
1 jx import --batch-mode
2
3 jx get activities \
4   --filter go-demo-6 \
5   --watch
6
7 cd ..
```

Please wait until the activity of the application shows that all the steps were executed successfully, and stop the watcher by pressing *ctrl+c*.

Now we can promote our last release to production.

## Versioning Releases Through Tags

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

```
1 jx get applications
```

```
1 APPLICATION STAGING PODS URL
2 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 likely 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`. 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`. You can visit the [releases](#) and install one that matches your operating system. You are free to do that, but I will not provide a more detailed walkthrough. Since I don't know whether you are using macOS, Linux, or Windows, I would need to prepare instructions for all three. The alternative is to create a DevPod. Since that 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. At the same time, that will act as a refresher of what we learned in the [Improving And Simplifying Software Development](#) chapter.



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

```
1 jx create devpod --label go --batch-mode
2
3 jx rsh -d
4
5 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. Next, we'll

```
install jx-release-version-linux.
```

```
1 curl -L \
2   -o /usr/local/bin/jx-release-version \
3   https://github.com/jenkins-x/jx-release-version/releases/download/v1.0.17/jx-rel
4 se-version-linux
5
6 chmod +x \
7   /usr/local/bin/jx-release-version
```

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.

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

```
1 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.

```
1 ...
2 v0.0.210
3 v0.0.211
4 ...
```

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.

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

```
1 jx-release-version
```

In my case, the output is `0.0.212`. `jx-release-version` examined the existing tags, found which is 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 what the latest tag is, 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 having the major version `1`.

```
1 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`.

```
1 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`s, the new release should be `v1.0.1`s.

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 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.

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

```
1 exit
2
3 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. Creating dummy tags only for the sake of bumping future major and minor releases is not very elegant. On top of that, 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'll fix that next.

## Controlling Release Versioning From Jenkins X Pipelines

One way to take better control of versioning is to add variable `VERSION` to the project's Makefile. Please open it in your favorite editor and add the snippet that follows as the first line.

```
1 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.

Please save the changes before proceeding.

If we'd execute `jx-release-version`, the output would be `1.0.0` (or whichever 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 is its usage defined.

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

```
1 git add .
2
3 git commit \
4   --message "Finally 1.0.0"
5
6 git push
7
8 jx get activities \
```

```
9      --filter go-demo-6 \
10     --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.

```
1 ...
2 vfarcic/go-demo-6/master #2 1m20s Running Version: 1.0.0
3 ...
```

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.

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

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

The output is as follows.

```
1 APPLICATION STAGING PODS URL
2 go-demo-6 1.0.0 1/1 http://go-demo-6.jx-staging.52.42.36.56.nip.io
```

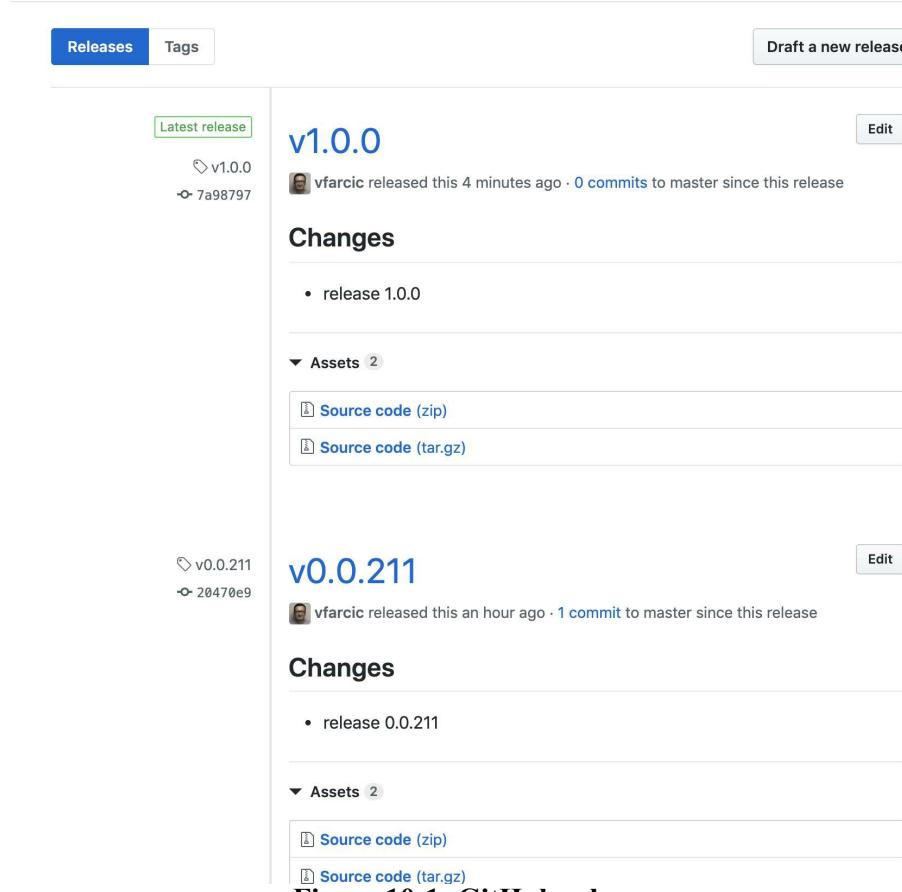
We can see that in my case the *go-demo-6* release running in the staging environment is 1.0.0. Finally, the last thing we'll do is to validate that the release stored in GitHub is also based on the new major version.



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

```
1 GH_USER=[...]
2
3 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.



**Figure 10-1: GitHub releases**

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

```

1 echo "A silly change" | tee README.md
2
3 git add .
4
5 git commit \
6   --message "A silly change"
7
8 git push
9
10 jx get activity \
11   --filter go-demo-6 \
12   --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 would 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 do not want to use semantic versioning?

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

```
1 cd ..
```

## Customizing Versioning Logic

Sometimes we do want to use semantic versioning, but we want to add additional information. We usually do that by adding prefixes or suffixes to the three versions (major, minor, and patch). As a matter of fact, we are already doing that. So far, our pipelines use “pure” semantic versioning, except for GitHub releases. As a refresher, the step that creates release notes is `jx step changelog --version v\$ (cat ../../VERSION)`. You can see that `v` is added as a prefix to the version.

If we look at [Kubernetes releases](#), we can see those like `v1.14.0-rc.1`, `v1.14.0-beta.2`, `v1.14.0-alpha.3`, and `v1.14.0-alpha.2`. They are using a variation of semantic versioning. Instead of increasing patch version with each release, Kubernetes community is creating `alpha`, `beta`, and release candidate (`rc`) releases leading to the final `v1.14.0` release. Once `v1.14.0` is released, batch versions are incremented following semantic versioning rules until a new minor or major release. Jenkins X does not support such a versioning scheme out of the box. Nevertheless, it should be a relatively simple change to Jenkinsfile to accomplish that or some other variation of semantic versioning. We won't go into practical exercises that would demonstrate that since I assume that you should be able to make the necessary changes on your own.

One of the ways we could modify how versioning works is by changing Makefile. Instead of having a fixed entry like `VERSION := 1.0.0`, we could use a function as a value. Many of the variable declarations in the existing Makefile are getting dynamically generated values, and I encourage you to explore them. As long as you can script generation of the version, you should have no problem implementing your own logic. Just as with custom prefixes and suffixes, we won't go into practical examples.

Finally, you might choose to ignore the existence of `jx-release-version` and replace the parts of your pipeline that use it with something completely different.

The choice of whether to adopt Jenkins X implementation of semantic versioning as-is, to modify it to suit your needs, or to change the logic to an entirely different versioning scheme is up to you. Just remember that the more you move away from a standard, the more difficulty you will have in adopting tools and their out-of-the-box behavior. Sometimes there is a good reason not to use conventions, while in others we invent our own for no particular reason. If you do choose not to use semantic versioning, I hope you are in the former group.

## Versioning With Maven, NodeJS, And Other Build Tools

The chances are that your applications are not written in Go. Even if some are, that's probably not the case with all of them. You might have applications written in Java, NodeJS, Python, Ruby, or a myriad of other languages. You might be using Maven, Gradle, or any other build and packaging tool. What should you do in that case?

If you are using Maven, everything we explored so far is applicable with a difference that the version is defined in *pom.xml* instead of Makefile. Just as we added `VERSION := 1.0.0` to Makefile, you'd need to add `<version>` inside `<project>` in *pom.xml*. Everything else stays the same.

In the case of NodeJS, `jx-release-version` is not used at all. Instead, pipelines rely on the [semantic-release plugin](#) to determine the next version number by fetching information from commit messages.

What about other build tools? You can still add Makefile that would serve only for versioning, or you can control major and minor version increments by creating dummy Git tags. We explored both, so you should have no problem adopting one of those. Or you might choose to adopt versioning through a plugin in your favorite build tool (e.g., Gradle) or to roll out your own. Finally, you might want to contribute to [jx-release-version](#) by adding support for your favorite build tool. That would be the best option. After all, it is open-source, and it relies on the community of volunteers.

## What Now?

We saw how Jenkins X approaches releases by adopting semantic versioning. We also explored a few strategies we can employ to control

increments of major and minor versions. It's up to you to either adopt semantic versioning or to adapt your pipelines to the versioning type you prefer instead.

Now you need to decide whether to continue using the cluster or to destroy it. If you choose to destroy it or to uninstall Jenkins X, you'll find the instructions at the bottom of the Gist you chose at the beginning of this chapter.

If you destroyed the cluster or you uninstalled Jenkins X, please remove the repositories and the local files we created. You can use the commands that follow for that.



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

```
1 GH_USER=[...]
2
3 hub delete -y \
4   $GH_USER/environment-jx-rocks-staging
5
6 hub delete -y \
7   $GH_USER/environment-jx-rocks-production
8
9 rm -rf environment-jx-rocks-production
10
11 rm -rf ~/.jx/environments/$GH_USER/environment-jx-rocks-*
```

# Going Serverless

We saw that Jenkins X packs a lot of useful tools and processes and that it is very different from the “traditional” Jenkins. Serverless Jenkins X solves many problems we had with Jenkins used in the static flavor. Jenkins does not scale. If we need more power because the number of concurrent builds increased, we cannot scale Jenkins. We cannot hook it into Kubernetes HorizontalPodAutoscaler that would increase or decrease the number of replicas based on metrics like the number of concurrent builds.

Simply put, Jenkins does not scale. At times, our Jenkins is struggling under heavy load. At others, it is wasting resources when it is underutilized. As a result, we might need to increase its requested memory and CPU as well as its limits to cover the worst-case scenario. As a result, when fewer builds are running in parallel, it is wasting resources, and when more builds are running, it is slow due to insufficient amount of assigned resources. And if it reaches its memory limit, it’ll be shut down and rerun (potentially on a different node), thus causing delays or failed builds.

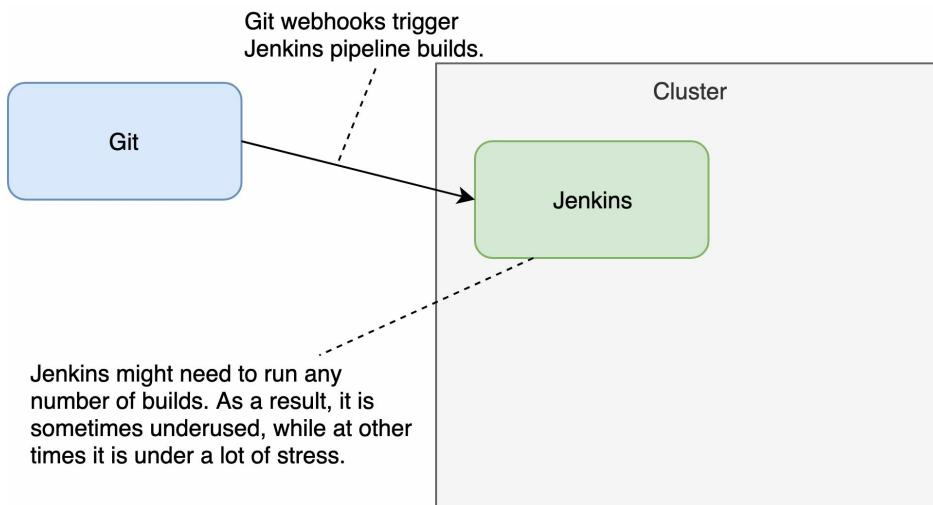
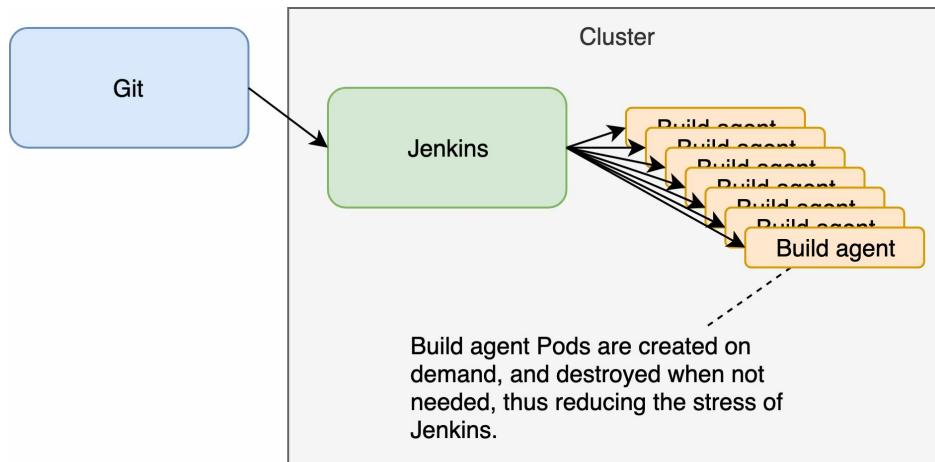


Figure 11-1: A single static Jenkins

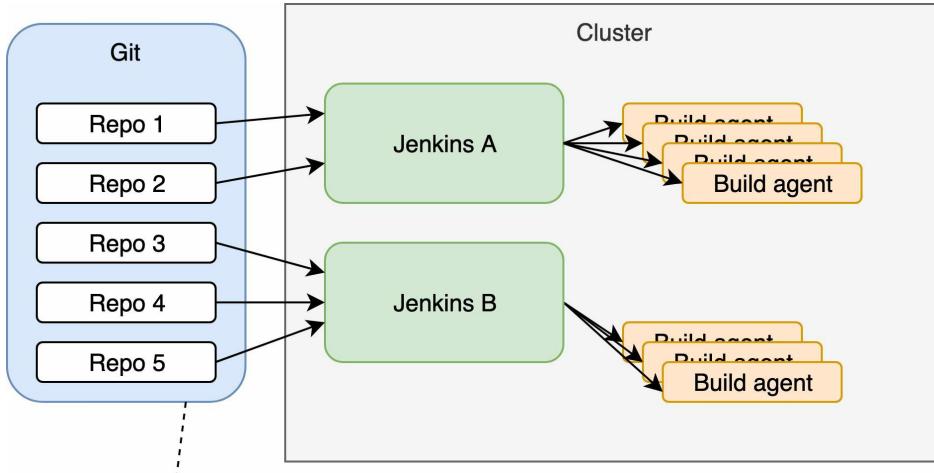
Truth be told, the need to scale Jenkins is smaller than for many other tools, since a significant part of the workload is delegated to agents which,

in our case, are running inside disposable one-shot Pods. However, even though most of the tasks are executed inside agents, sooner or later you will reach the limits of what a single Jenkins can do without affecting performance and stability. That is, assuming that your workload is big enough. If you're having only a few projects and a limited number of concurrent builds, a single Jenkins will satisfy most of your needs.



**Figure 11-2: Static Jenkins with on-demand one-shot agent Pods**

The only way we can prevent Jenkins from being overloaded is either to increase its memory and CPU or to create new instances. However, creating new instances does not produce the same results as scaling. If we define scaling as an increase (or a decrease) of the number of replicas of a single instance, having multiple replicas means that the load is shared across them. Since that is not possible with Jenkins, we need to increase the number of instances. That means that we cannot have all our pipelines defined in a single Jenkins, but that we can create different instances, each hosting different sets of pipelines.



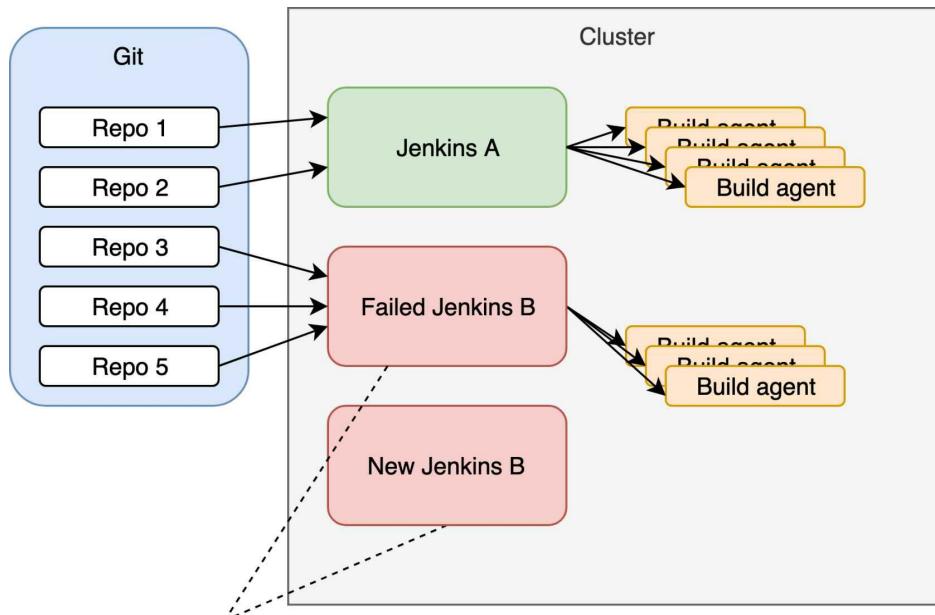
Git repositories fire webhooks to dedicated Jenkins instances, instead to the platform that scales to accommodate the needs.

**Figure 11-3: Workload split into multiple Jenkins instances**

While it's true that we can accomplish resemblance of scaling by spinning multiple Jenkins instances, that does not tackle the problem of high availability. If every Jenkins instance runs only a single replica, what happens when it fails? The good news is that Kubernetes takes care of rerunning failed instances. The bad news is that a failure of a single-replica application inevitably produces downtime. The period from the moment an instance fails until it is recreated by Kubernetes and the processes inside it are fully operational is the period when Jenkins is not available. Whether that is a couple of seconds or a few minutes does not change the fact that it does not always work. No single-replica application can guarantee high-availability in modern terms (e.g., 99.999% availability).

What is the real problem if we do not have Jenkins available all the time? Do we care if every once in a while it stops working for a few minutes? The answer to those questions depends primarily on you. If you are a small company, a few minutes of unavailability that affects only developers (not the end-users) is often not a big deal. But, the real issues are coming from a different direction. What happens when we push code to Git which, in turn, triggers a webhook request? If there is no one to receive that request (e.g., Jenkins is down), no one will even know that a build was not triggered. We could be living in oblivion thinking that everything went fine and that a new release passed all validations and was deployed to staging or even production. When Jenkins fails, all the builds that were running at that time will fail as well. That's not a problem by itself since builds can fail for a variety of other reasons (e.g., a bug that results in a

failed test). The issue is that we are notified of failed builds only when Jenkins is running. If it goes down (ungracefully), a build will stop abruptly, and the step that is supposed to notify us will never be executed. Nevertheless, if you are small to medium size company, avoiding those problems might be more expensive than the cost of fixing them, unless the solution is already available out of the box.



Failed Jenkins results in failed builds without notifications. A new instance spin up by Kubernetes as the replacement needs time to initialize. As a result, there is a period when none of the instances are running, and webhooks will fail.

**Figure 11-4: Temporary downtime and broken builds due to a failed Jenkins instance**

What can we do when faced with two options, and none of them are acceptable? We can come up with a third one. If Jenkins cannot scale, and the only way to share the load is by splitting the pipelines across multiple instances, then maybe we can go to the other extreme and dedicate a Jenkins instance to each build. All we'd have to do is create a highly available API that would accept incoming webhooks, and let it spin up a new Jenkins instance for each build. That could not be a generic Jenkins, but a customized one that would contain not only the engine but all the tools that a build needs. It would need to be something similar to a combination of a Jenkins master and a pipeline specific agent.

But that poses a few new challenges. First of all, what should we use as a highly available API? How will it spin up new Jenkins instances that contain pipeline-specific tools? Will it be fast enough? Is Jenkins designed

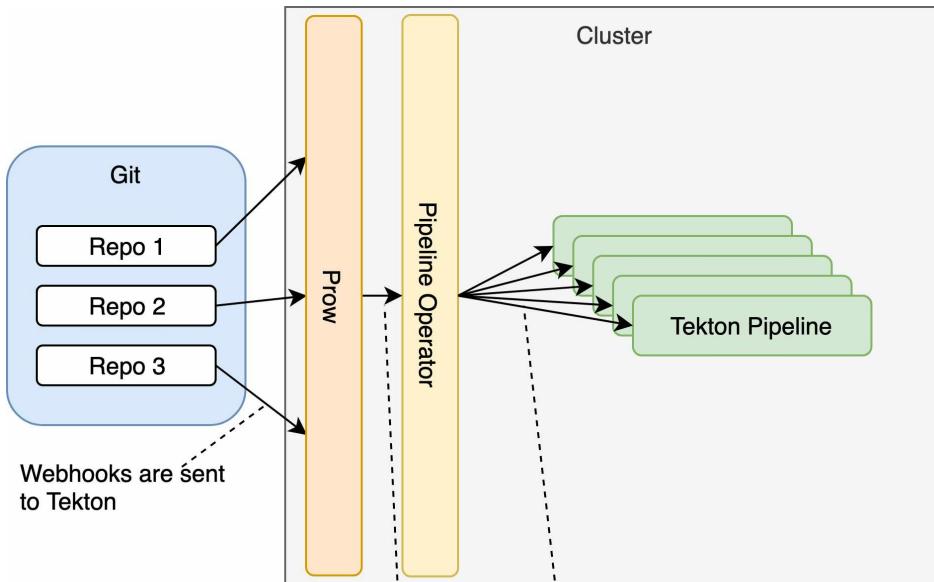
to work like that, or should we use something else? The answer to those questions requires a more in-depth analysis. For now, what matters is that Jenkins X solved those problems. There is now a highly available API that accepts webhook requests. There is a mechanism that will translate your pipelines into a set of tasks that are created and run on-demand, and destroyed when finished.

If we spin up pipeline-runners on demand and we destroy them when builds are finished, we can say that we'd get a serverless solution. Each request from a webhook would result in one or more new processes that would run all the steps defined in a pipeline. As long as the API that accepts webhook requests is highly available and knows how to spin up and destroy processes required for each pipeline build, we should have a highly-available and fault-tolerant solution that uses only the resources it needs.



I really dislike the word “serverless” because it is misleading. It makes you believe that there are no servers involved. It should rather be called servers-not-managed-by-us. Nevertheless, I will continue using the word serverless to describe processes that spin up on demand and get destroyed when done, simply because that is the most commonly used name today.

Jenkins X in serverless mode uses [Prow](#) as the highly available API that receives webhook requests. Jenkins is replaced with Jenkins X Pipeline Operator and [Tekton](#). It translates the events coming to Prow into Tekton pipelines. Given that Tekton is very low-level and its pipelines are hard to define, Pipeline operator translates Jenkins pipelines into Tekton Pipelines which, in turn, spin up Tekton Tasks, one for each step of the pipeline. Those pipelines are defined in jenkins-x.yml.



Prow forwards events to Pipeline Operator that translates jenkins-x.yml pipeline into Tekton Pipeline resources.

**Figure 11-5: Serverless flavor of Jenkins X**

One of the things you'll notice is that there is no Jenkins in that diagram. It's gone or, to be more precise, replaced with other components, more suitable for today's challenges. But, introducing entirely new components poses a new set of challenges, especially for those already experienced with Jenkins.

Learning that new components like Prow, Pipeline Operator, and Tekton might make you feel uncomfortable unless you already used them. If that's the case, you can rest at ease. Jenkins X makes their usage transparent. You can go deep into their inner workings, or just ignore their existence since Jenkins X does all the heavy lifting and integration between them.

What matters is that serverless Jenkins X is here and it brings a whole new set of opportunities and advantages compared to other solutions. It is the first continuous delivery solution designed specifically for Kubernetes. It is cloud-native and based on serverless principles, and it is fault-tolerant and highly-available. It scales to meet any demand, and it does not waste resources. And all that is done in a way transparent to the user and without the need to spend eternity trying to figure out how to assemble all the pieces of a very complex machine.

In this chapter, we will explore how to increase the number of Jenkins instances, as well as how to transition into a serverless solution. By

understanding both options (multi-instance Jenkins, and serverless on-demand pipelines), you should be able to make a decision which direction to take.

## Exploring Prow, Jenkins X Pipeline Operator, And Tekton

The serverless flavor of Jenkins X or, as some call it, Jenkins X Next Generation, is an attempt to redefine how we do continuous delivery and GitOps inside Kubernetes clusters. It does that by combining quite a few tools into a single easy-to-use bundle. As a result, most people will not have a need to understand intricacies of how the pieces work independently, nor how they are all integrated. Instead, many will merely push a change to Git and let the system do the rest. But, there are always those who would like to know what's happening behind the hood. To satisfy those craving for insight, we'll explore the processes and the components involved in the serverless Jenkins X platform. Understanding the flow of an event initiated by a Git webhook will give us insight into how the solution works and help us later on when we go deeper into each of the new components.

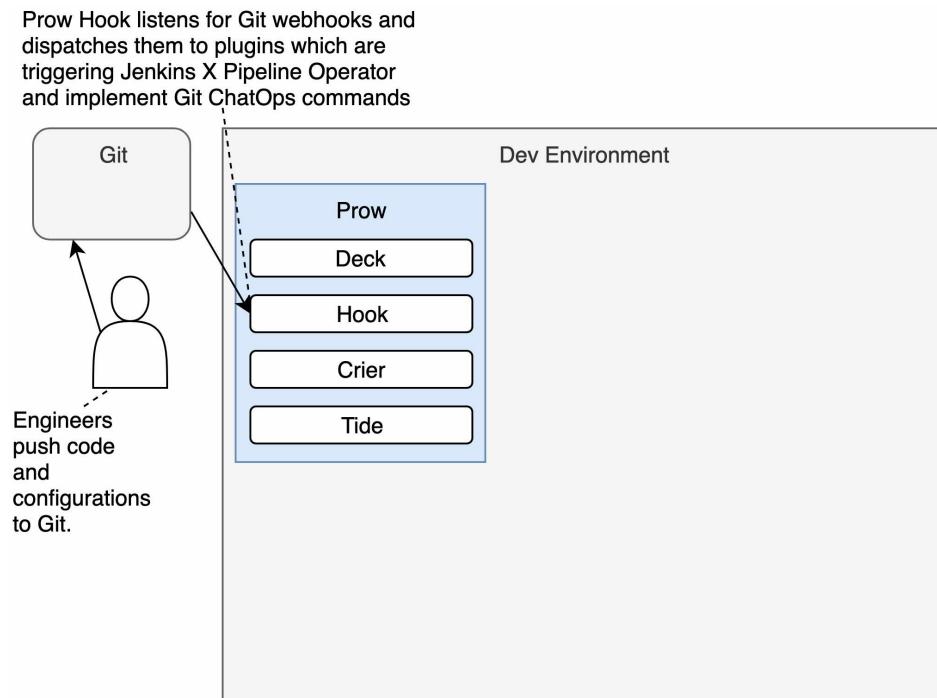


The description that follows might be too much for some people. I will not think less of you if you choose to skip it and enjoy using serverless Jenkins X without knowing its inner workings. After all, the main objective of Jenkins X is to abstract the details and let people practice continuous delivery without spending months trying to learn the intricacies of complex systems like Kubernetes and Jenkins X.

Everything starts with a push to a Git repository which, in turn, sends a webhook request to the cluster. Where things differ is that there is no Jenkins to accept those requests. Instead, we have [Prow](#). It does quite a few things but, in the context of webhooks, its job is to receive requests and decide what to do next. Those requests are not limited only to push events, but also include slash commands (e.g., `/approve`) we can specify through pull request comments.

Prow consists of a few distinct components (e.g., Deck, Hook, Crier, Tide, and a few more). However, we won't go into the roles of each of them just yet. For now, the vital thing to note is that Prow is the entry point to the

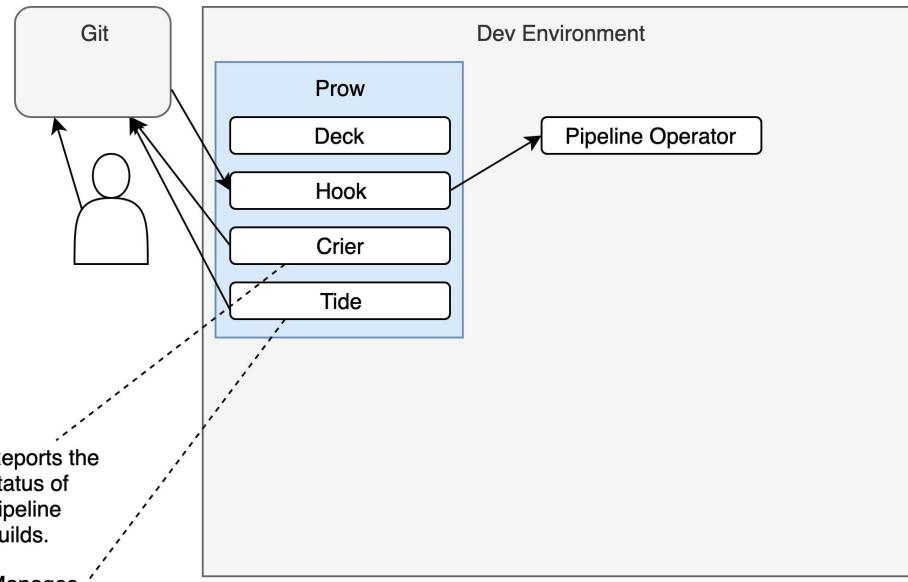
cluster. It receives Git requests generated either by Git actions (e.g., push) or through slash commands in comments.



**Figure 11-6: The role of Prow in serverless Jenkins X**

Prow might do quite a few things upon receiving a request. If it comes from a command from a Git comment, it might re-run tests, merge a pull request, assign a person, or one of the many other Git related actions. If a webhook informs it that a new push was made, it will send a request to the Jenkins X Pipeline Operator which will make sure that a build corresponding to a defined pipeline is run. Finally, Prow also reports the status of a build back to Git.

Those features are not the only types of actions Prow might perform but, for now, you probably got the general Gist. Prow is in charge of communication between Git and the processes inside our cluster.



**Figure 11-7: The relations between Git, Prow, and Pipeline Operator**

When a Prow Hook receives a request from a Git webhook, it forwards it to Jenkins X Pipeline Operator. The role of the operator is to fetch jenkins-x.yml file from the repository that initiated the process and to transform it into Tekton Tasks and Pipelines. They, in turn, define the complete pipeline that should be executed as a result of pushing a change to Git.

The reason for the existence of the Pipeline Operator is to simplify definitions of our continuous delivery processes. Tekton does the heavy lifting, but it is a very low-level solution. It is not meant to be used directly. Writing Tekton definitions can be quite painful and complicated. Pipeline Operator simplifies that through easy to learn and use YAML format for defining pipelines.

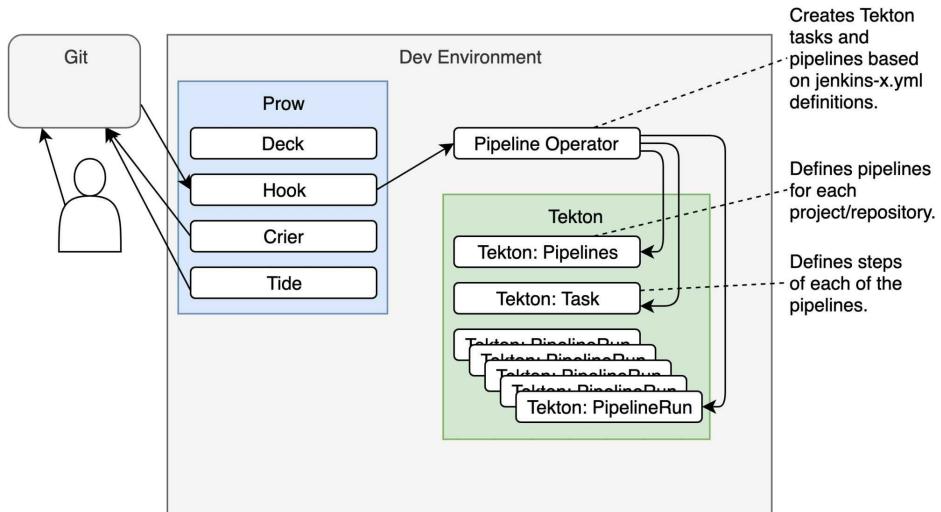
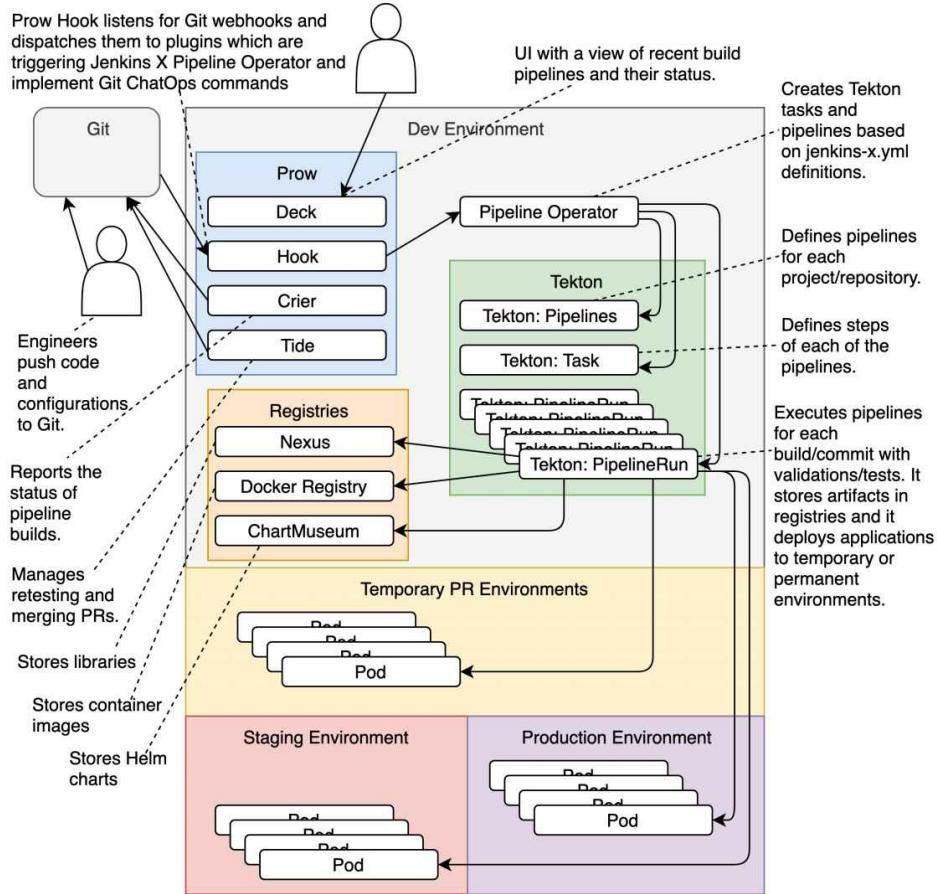


Figure 11-8: The relations between Git, Prow, and Pipeline Operator

Tekton creates a PipelineRun for each build initiated by each push to one of the associated branches (e.g., master branch, PRs, etc.). It performs all the steps we need to validate a push. It runs tests, stores binaries in registries (e.g., Docker Registry, Nexus, and ChartMuseum), and it deploys a release to a temporary (PR) or a permanent (staging or production) environment.

The complete flow can be seen in the diagram 11-9.



**Figure 11-9: The complete flow of events in serverless Jenkins X**



You might notice the same diagram in [jenkins-x.io](https://jenkins-x.io). The community thought that it is useful, so I contributed it.

As I already mentioned, not everyone needs to understand the flow of events nor to have a deep understanding of all the components involved in the process. For most users, the only important thing to understand is that pushing changes to Git will result in the execution of the builds defined in jenkins-x.yml pipeline. That's the beauty of Jenkins X. It simplifies our lives by making complicated processes simple.

If, on the other hand, you are interested in details, we will cover each of the components in more details in the next chapters.

# Implementing ChatOps



The examples in this chapter work only with serverless Jenkins X.

Jenkins X main logic is based on applying GitOps principles. Every change must be recorded in Git, and only Git is allowed to initiate events that result in changes in our clusters. That logic is the cornerstone of Jenkins X, and it served us well so far. However, there are actions we might need to perform that do not result in changes to the source code or configurations.

We might need to assign a pull request to someone for review. That someone might have to review the changes and might need to run some manual tests if they are not fully automated. A pull request could need additional changes, and the committer might need to be notified about that. Someone might need to approve and merge a pull request or choose to cancel it altogether. The list of the actions that we might have to perform once a pull request is created can be quite extensive, and many of them do not result in changes to source code. The period starting with the creation of a pull request and until it is merged to master is mostly filled with communication, rather than changes to the project in question. As such, we need an effective way to facilitate that communication.

The communication and decision making that surrounds a pull request needs to be recorded. We need to be able to track who said what and who made which decision. Otherwise, we would not be able to capture the events that lead to a merge of a pull request to the master branch. We'd be running blind. That means that verbal communication must be discarded since it would not be recorded.

Given that such communication should be closely related to the pull request, we can discard emails and wiki pages as well, thus leading us back to Git. Almost every Git platform has a mechanism to create comments tied to pull requests. We can certainly use those comments to

record the communication. But, communication by itself is useless if it does not result in concrete actions.

If we do need to document the communication surrounding a pull request, it would be a waste of effort to have to perform related actions separately. Ideally, communication should result in actions.

When we write a comment that a pull request should be assigned to a reviewer, it should trigger an action that will do the actual assignment and notify that person that there is a pending action. If we comment that a pull request should be labeled as “urgent”, that comment should add the label. If a reviewer writes that a pull request should be canceled, that comment should close the pull request. Similarly, if a person with sufficient permissions comments that the pull request is approved, it should be merged automatically.

There are a couple of concepts that need to be tied together for our process surrounding pull request to be effective. We need to be able to communicate, and we already have comments for that. People with sufficient privileges need to be able to perform specific actions (e.g., merge a pull request). Git platforms already implement some form of RBAC (Role Based Authentication), so that part is already solved. Furthermore, we need to be notified that there is a pending action we should perform as well as that a significant milestone is reached. This is solved as well. Every Git flavor provides a notification mechanism. What we’re missing is a process that will tie all that together by executing actions based on our comments.

We should be able to implement ChatOps principles if we manage to convert comments into actions controlled by RBAC and if we receive notifications when there is a pending task.

The idea behind ChatOps is to unify communication about the work that should be done with the history of what has been done. The expression of a desire (e.g., approve this pull request) becomes an action (execution of the approval process) and is at the same time recorded. ChatOps is similar to verbal communication or, to be more precise, commands we might give if we would have a butler. “Please make me a meal” is an expression of a desire. Your butler would transmit your wish to a cook, wait until the meal is ready, and bring it back to you. Given that we are obsessed (for a good reason) to record everything we do when developing software, verbal

expressions are not good enough, so we need to write them down. Hence the idea of ChatOps.



ChatOps converts parts of communication into commands that are automatically executed and provides feedback of the results.

In a ChatOps environment, a chat client is the primary source of communication for ongoing work. However, since we adopted Git as the only source of truth, it should come as no surprise that the role of a chat client is given to Git. After all, it has comments, and that can be considered chat (of sorts). Some call it GitChat, but we'll stick with a more general term ChatOps.

If we assume that only Git should be able to initiate a change in our clusters, it stands to reason that such changes can be started either by a change in the source code, by writing comments in Git, or by creating an issue.

We can define ChatOps as conversation driven development. Communication is essential for all but single-person teams. We need to communicate with others when the feature we're developing is ready. We need to ask others to review our changes. We might need to ask for permission to merge to the master branch. The list of the things we might need to communicate is infinite. That does not mean that all communication becomes ChatOps, but rather that parts of our communication does. It's up to the system to figure out which parts of communication should result in actions, and what is a pure human-to-human messaging without tangible outcomes.

As we already saw, four elements need to be combined into a process. We need communication (comments), permissions (RBAC), notifications (email), and actions. All but the last are already solved in every Git platform. We just need to figure out how to combine comments, permissions, and notifications into concrete actions. We'll do that by introducing Prow to our solution.

[Prow](#) is a project created by the team managing continuous delivery processes for [Kubernetes](#) projects. It does quite a few things, but we will

not use everything it offers because there are better ways to accomplish some of its tasks. The parts of Prow we are primarily interested in are those related to communication between Git and processes running in our cluster. We'll use it to capture Git events created through *slash commands* written in comments. When such events are captured, Prow will either forward them to other processes running in the cluster (e.g., execute pipeline builds), or perform Git actions (e.g., merge a pull request).

Since this might be the first time you hear the term slash commands, so a short explanation might be in order.

Slash commands act as shortcuts for specific actions. Type a slash command in the Git comment field, click the button, and that's it. You executed a task or a command. Of course, our comments are not limited to slash commands. Instead, they are often combined with "conversational" text. Unfortunately, commands and the rest of communication must be in separate lines. A command must be at the start of a line, and the whole line is considered a single command. We could, for example, write the following text.

```
1 This PR looks OK.  
2  
3 /lgtm
```

Prow parses each line and will deduce that there is a slash command /lgtm.

Slash commands are by no means specific to Git and are widely used in other tools. Slack, for example, is known for its wide range of supported slash commands and the ability to extend them. But, since we are focused on Git, we'll limit our ChatOps experience with Slash commands to what Prow offers as the mechanism adopted by Jenkins X (and by Kubernetes community).

All in all, Prow will be our only entry point to the cluster. Since it accepts only requests from Git webhooks or slash commands, the only way we will be able to change something in our cluster is by changing the source code or by writing commands in Git comments. At the same time, Prow is highly available (unlike static Jenkins), so we'll use it to solve yet another problem.



At the time of this writing (April 2019), Prow only supports GitHub. The community is working hard on adding support for other Git platforms. Until that is finished, we are restricted to GitHub. If you do use a different Git platform (e.g., GitLab), I still recommend going through the exercises in this chapter. They will provide a learning experience. The chances are that, by the time you start using Jenkins X in production, support for other Git flavors will be finished.



We cannot use Prow with anything but GitHub, and serverless Jenkins X doesn't work without Prow. However, that is not an issue. Or, at least, it is not a problem anymore. The team created a new project called Lighthouse that performs all the same functions that Prow provides while supporting all major Git providers. Everything you read about Prow applies equally to Lighthouse, which should be your choice if you do not use GitHub.

As always, we need a cluster with Jenkins X to explore things through hands-on exercises.

## Creating A Kubernetes Cluster With Jenkins X

If you kept the cluster from the previous chapter and it contains serverless Jenkins X, you can skip this section. Otherwise, we'll need to create a new Jenkins X cluster.



All the commands from this chapter are available in the [12-prow.sh](#) Gist.

For your convenience, the Gists that will create a new Jenkins X cluster or install it inside an existing one are as follows.

- Create a new serverless **GKE** cluster: [gke-jx-serverless.sh](#)
- Create a new serverless **EKS** cluster: [eks-jx-serverless.sh](#)
- Create a new serverless **AKS** cluster: [aks-jx-serverless.sh](#)
- Use an **existing** serverless cluster: [install-serverless.sh](#)

Now we can explore Prow inside the serverless Jenkins X bundle.

# Exploring The Basic Pull Request Process Through ChatOps

The best way to explore the integration Jenkins X provides between Git, Prow, and the rest of the system is through practical examples. The first thing we'll need is a project, so we'll create a new one.



Make sure that you are not inside an existing repository before executing the command that follows.

```
1 jx create quickstart \
2   --filter golang-http \
3   --project-name jx-prow \
4   --batch-mode
5
6 cd jx-prow
7
8 jx get activities \
9   --filter jx-prow --watch
```

We created a Go-based project called `jx-prow`, entered into the local copy of the Git repository `jx` created for us, and started watching the activity. After a while, all the steps in the output will be in the `Succeeded` status, and we can stop the watcher by pressing `ctrl+c`.

Since most of the ChatOps features apply to pull requests, we need to create one.

```
1 git checkout -b chat-ops
2
3 echo "ChatOps" | tee README.md
4
5 git add .
6
7 git commit \
8   --message "My first PR with prow"
9
10 git push --set-upstream origin chat-ops
```

We created a new branch `chat-ops`, we made a silly change to `README.md`, and we pushed the commit.

Now that we have the branch with the change to the source code, we should create a pull request. We could do that by going to GitHub UI but, as you already know from the [Working With Pull Requests And Preview Environments](#) chapter, `jx` already allows us to do that through the

command line. Given that I prefer terminal screen over UIs (and you don't have a say in that matter), we'll go with the latter option.

```
1 jx create pullrequest \
2   --title "PR with prow" \
3   --body "What I can say?" \
4   --batch-mode
```

We created a pull request and are presented with a confirmation message with a link. Please open it in your favorite browser.

You will notice a few things right away. A comment was created describing the process we should follow with pull requests. In a nutshell, the PR needs to be approved. Someone should review the changes we are proposing. That might mean going through the code, performing additional manual tests, or anything else that the approver might think is needed before he gives his OK.

Near the bottom, you'll see that a few checks are running. The *serverless-jenkins* process is executing the part of the pipeline dedicated to pull requests. At the end of the process, the application will be deployed to a temporary PR-specific environment. All that is the same as before. However, there is an important aspect of PR that we did not yet explore. There are rules that we need to follow before we merge to master and the communication happening between Git and Prow.

The second activity is called *tide*. It will be in the *pending* state until we complete the process described in the comment.

Tide is one of the Prow components. It is in charge of merging the pull request to the master branch, and it is configured to do so only after we send it the `/approve` command. Alternatively, Tide might close the pull request if it receives `/approve cancel`. We'll explore both soon.

[APPROVALNOTIFIER] This PR is NOT APPROVED

This pull-request has been approved by:  
 To fully approve this pull request, please assign additional approvers.  
 We suggest the following additional approver: [vfarcic](#)

If they are not already assigned, you can assign the PR to them by writing `/assign @vfarcic` in a comment when ready.

The full list of commands accepted by this bot can be found [here](#).

The pull request process is described [here](#)

▼ Details  
 Needs approval from an approver in each of these files:  

- [OWNERS](#)

 Approvers can indicate their approval by writing `/approve` in a comment  
 Approvers can cancel approval by writing `/approve cancel` in a comment

vfarcic added the `size/XS` label 4 minutes ago

---

Add more commits by pushing to the **chat-ops** branch on [vfarcic/jx-prow](#).

The screenshot shows a GitHub pull request page with a Jenkins X pipeline status summary. It includes sections for pending checks (serverless-jenkins and tide), a conflict-free status, and a merge button.

Some checks haven't completed yet  
 2 pending checks

- serverless-jenkins Pending — Not all Tasks in the Pipeline have finished exec...
- tide Pending — Not mergeable. Needs approved label.

This branch has no conflicts with the base branch  
 Merging can be performed automatically.

Merge pull request ▾ You can also [open this in GitHub Desktop](#) or view command line instructions.

**Figure 12-1: The message that explains the rules governing the pull request process**

Next, we'll wait until the Jenkins X build initiated by the creation of the PR is finished. Once it's done, you'll see a comment stating *PR built and available in a preview environment*. Feel free to click the link next to it to open the application in a browser.

The “real” indication that the build is finished is the *serverless-jenkins* “*All Tasks have completed executing*” message in the *checks* section near the bottom of the screen. When we created the PR, a webhook request was sent to Prow which, in turn, notified the system that it should run a build of the associated pipeline (the one defined in `jenkins-x.yml`). Not only that Prow initiated the build, but it also monitored its progress. Once the build was finished, Prow communicated the outcome to GitHub.

If you take another look at the description of the pull request process, you'll see that you can assign it to someone. We could, as you probably

already know, do that through the standard GitHub process by clicking a few buttons. Since we're trying to employ the ChatOps process, we'll write a comment with a slash command instead.

Please type `/assign` as the comment. Feel free to add any text above or below the command. You can, for example, write something similar to the following text.

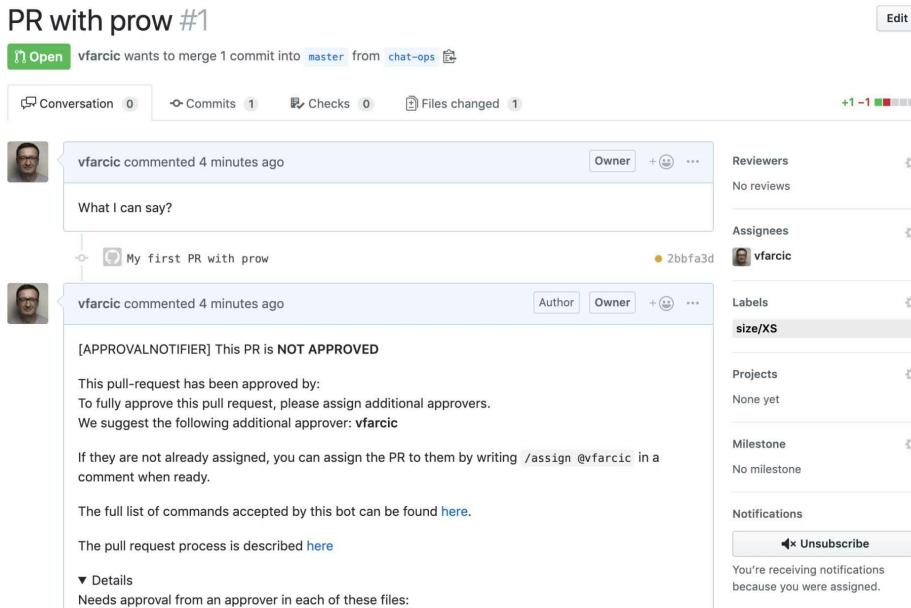
```
1 /assign  
2  
3 This PR is urgent, so please review it ASAP
```

The text does not matter. It is only informative, and you are free to write anything you want. What matters is the slash command `/assign`. When we create the comment, GitHub will notify Prow. It will parse it, it will process all slash commands, and it will discard the rest. Please click the *Comment* button once you're done writing a charming humanly readable message that contains `/assign`. Just remember that you cannot mix commands and text in the same line.



GitHub might not refresh automatically. If you do not see the expected result, try reloading the screen.

A moment after we created the comment, the list of assignees in the right-hand menu changed. Prow automatically assigned the PR to you. Now, that might seem silly at first. Why would you assign it to yourself? We need someone to review it, and that someone could be anybody but you. You already know what is inside the PR and you are confident that it most likely work as expected. Otherwise, you wouldn't create it. What we need is the second pair of eyes. However, we do not yet have any collaborators on the project, so you're the only one Prow could assign it to. We'll change that soon.



**Figure 12-2: Assigned pull request**

Just as we can assign a PR to someone, we can also unassign it through a slash command. Please type `/unassign` as a new comment and click the *Comment* button. A moment later your name will disappear from the list of assignees.



From now on, I'll skip surrounding slash commands with humanly-readable text. It's up to you to choose whether to make it pretty or just type bare-bones commands like in the examples. From the practical perspective, the result will be the same since Prow cares only about the commands and it discards the rest of the text. Just remember that a command and text cannot be mixed in the same line.

When we issued the command `/assign`, and since we were not specific, Prow made a decision who should it be assigned to. We can, instead, be more precise and choose who should review our pull request. Since you are currently the only collaborator on the project, we have limited possibilities, but we'll try it out nevertheless.

Please type `/assign @YOUR_GITHUB_USER` as the comment. Make sure to replace `YOUR_GITHUB_USER` with your user (keep `@`). Once you're done, click the *Comment* button. The result should be the same as when we issued the `/assign` command without a specific user simply because there

is only one collaborator this pull request can be assigned to. If we'd have more (as we will soon), we could have assigned it to someone else.

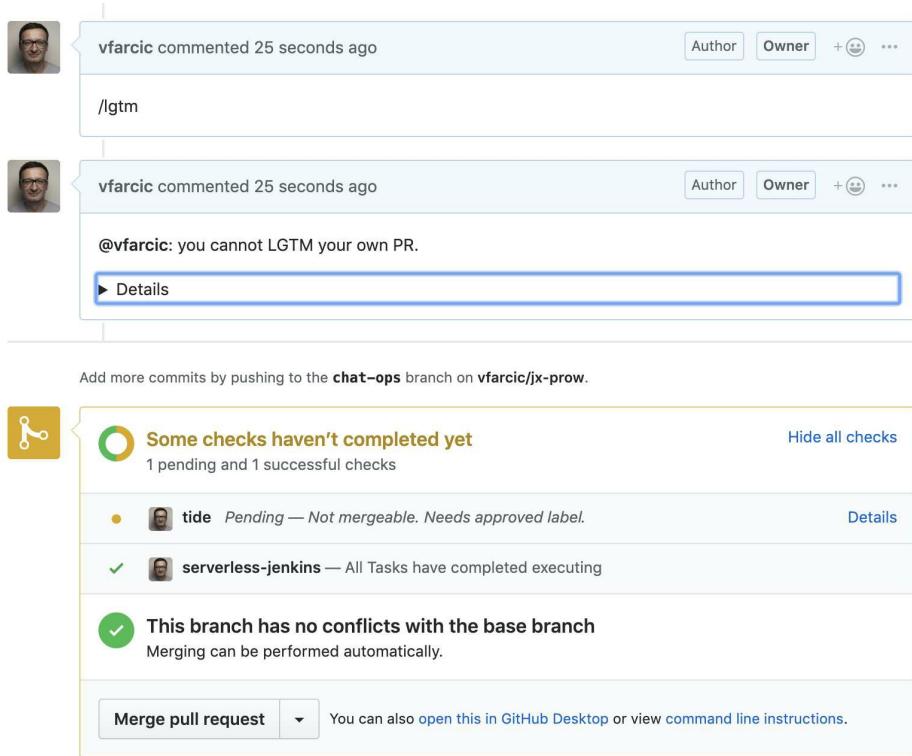
Next, a reviewer would go through the code changes and confirm that everything works correctly. The fact that *serverless-jenkins* check completed without issues provides an indication that all the validations executed as part of the pipeline build were successful. If still in doubt, a reviewer can always open the application deployed to the PR-specific temporary environment by clicking the link next to the *PR built and available in a preview environment* comment.

We'll assume that you (acting as a reviewer) believe that the change is safe to merge to the master branch. You already know that will initiate another pipeline build that will end with the deployment of the new release to the staging environment.

As you already saw by reading the description of the PR process, all we have to do is type `/approve`. But we won't do that. Instead, we'll use the `lgtm` abbreviation that stands for *looks good to me*. Originally, `/lgtm` is meant to provide a label that is typically used to gate merging. It is an indication that a reviewer confirmed that the pull request can be approved. However, Jenkins X implementation sets it to act as approval as well. Therefore, both `/approve` and `/lgtm` commands serve the same purpose. Both can be used to approve PRs. We'll use the latter mostly because I like how it sounds.

So, without further ado, please type `/lgtm` and click the *Comment* button.

A moment later, we should receive a notification (a comment) saying that *you cannot LGTM your own PR* (remember that you might need to refresh your screen). That makes sense, doesn't it? Why would you review your own code? There is no benefit in that since it would neither result in knowledge sharing nor additional validations. The system is protecting us from ourselves and from making silly mistakes.



**Figure 12-3: Rejected lgtm attempt**

If we are to proceed, we'll need to add a collaborator to the project. Before we do that, I should comment that if `/lgtm` worked, we could use `/lgtm cancel` command. I'm sure you can guess what it does.

Before we explore how to add collaborators, approvers, and reviewers, we'll remove you from the list of assignees. Since you cannot approve your own PR, it doesn't make sense for it to be assigned to you.

Please type `/unassign` and click the *Comment* button. You'll notice that your name disappeared from the list of assignees.

We need to define who is allowed to review and who can approve our pull requests. We can do that by modifying the `OWNERS` file generated when we created the project through the Jenkins X quickstart. Since it would be insecure to allow a person who made the PR to change that file, the one that counts is the `OWNERS` file in the master branch. So, that's the one we'll explore and modify.

```

1 git checkout master
2
3 cat OWNERS

```

The output is as follows.

```
1 approvers:  
2 - vfarcic  
3 reviewers:  
4 - vfarcic
```

The `OWNERS` contains the list of users responsible for the codebase of this repository. It is split between `approvers` and `reviewers` sections. Such split is useful if we'd like to implement a two-phase code review process in which different people would be in charge of reviewing and approving pull requests. However, more often than not, those two roles are performed by the same people so Jenkins X comes without two-phase review process out of the box (it can be changed though).

To proceed, we need a real GitHub user (other than yours) so please contact a colleague or a friend and ask him to give you a hand. Tell her that you'll need her help to complete some of the steps of the exercises that follow. Also, let her know that you need to know her GitHub user.



Feel free to ask someone for help in [DevOps20](#) Slack if you do not have a GitHub friend around. I'm sure that someone will be happy to act as a reviewer and an approver of your pull request.

We'll define two environment variables that will help us create a new version of the `OWNERS` file. `GH_USER` will hold your username, while `GH_APPROVER` will contain the user of the person that will be allowed to review and approve your pull requests. Typically, we would have more than one approver so that the review and approval tasks are distributed across the team. For demo purposes, the two of you should be more than enough.



Before executing the commands that follow, please replace the first [...] with your GitHub user and the second with the user of the person that will approve your PR.

```
1 GH_USER=[...]  
2  
3 GH_APPROVER=[...]
```

Now we can create a new version of the `OWNERS` file. As already discussed, we'll use the same users as both reviewers and approvers.

```
1 echo "approvers:
2 - $GH_USER
3 - $GH_APPROVER
4 reviewers:
5 - $GH_USER
6 - $GH_APPROVER
7 " | tee OWNERS
```

All that's left, related to the `OWNERS` file, is to push the changes to the repository.

```
1 git add .
2
3 git commit \
4   --message "Added an owner"
5
6 git push
```

Even though the `OWNERS` file defines who can review and approve pull requests, that would be useless if those users are not allowed to collaborate on your project. We need to tell GitHub that your colleague works with you by adding a collaborator (other Git platforms might call it differently).

```
1 open "https://github.com/$GH_USER/jx-prow/settings/collaboration"
```

Please login if you're asked to do so. Type the user and click the *Add collaborator* button.

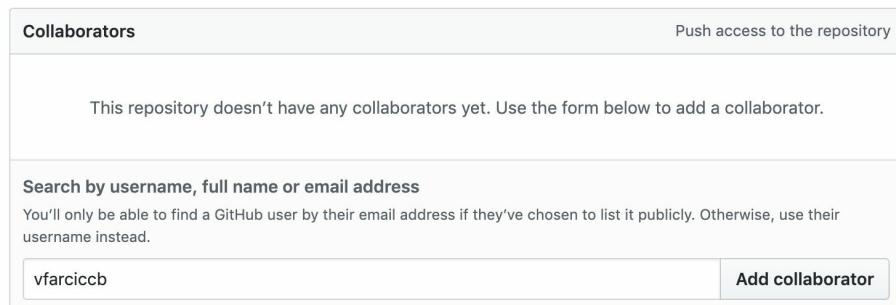


Figure 12-4: GitHub collaborators screen

Your colleague should receive an email with an invitation to join the project as a collaborator. Make sure that she accepts the invitation.



Not all collaborators must be in the `OWNERS` file. You might have people who collaborate on your project but are not allowed to review or approve pull requests.

Now we can assign the pull request to the newly added approver. Please go to the pull request screen and make sure that you are logged in (as you, not as the approver). Type `/assign @[...]` as the comment, where [...] is replaced with the username of the approver. Click the *Comment* button.

The approver should receive a notification email. Please let her know that she should go to the pull request (instructions are in the email), type `/approve`, and click the *Comment* button.



Please note that `/approve` and `/lgtm` have the same purpose in this context. We're switching from one to another only to show that both result in the pull request being merged to the master branch.

After a while, the PR will be merged, and a build of the pipeline will be executed. That, as you already know, results in a new release being validated and deployed to the staging environment.

You will notice that email notifications are flying back and forth between you and the approver. Not only that we are applying ChatOps principles, but we are at the same time solving the need for notifications that let each involved know what's going on as well as whether there are pending actions. Those notifications are sent by Git itself as a reaction to specific actions. The way to control who receives which notifications is particular to each Git platform and I hope that you already know how to subscribe, unsubscribe, or modify Git notifications you're receiving.

As an example, the email sent as the result of approving the PR is as follows.

```
1 [APPROVALNOTIFIER] This PR is APPROVED
2
3 This pull-request has been approved by: vfarciccb
4
5 The full list of commands accepted by this bot can be found here.
6
7 The pull request process is described here
```

```
8  
9 Needs approval from an approver in each of these files:  
10 OWNERS [vfarciccc]  
11 Approvers can indicate their approval by writing /approve in a comment  
12 Approvers can cancel approval by writing /approve cancel in a comment
```

All in all, the pull request is approved. As a result, Prow merged it to the master branch, and that initiated a pipeline build that ended with the deployment of the new release to the staging environment.

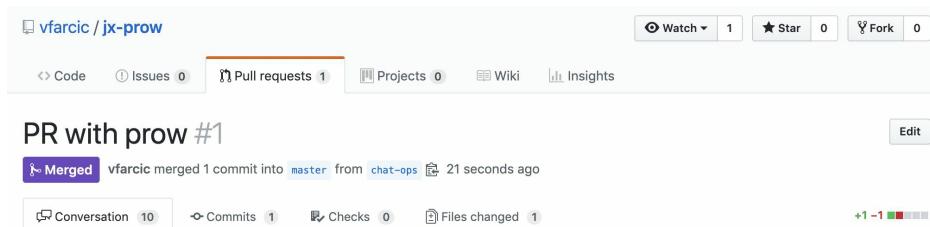


Figure 12-5: Merged pull request

Please wait until the *All checks have passed* message appears in the PR.

We already know from past that a merge to the master branch initiates yet another build. That did not change with the introduction of ChatOps. When we approved the PR, Prow merged it to the master branch, and from there the same processes were executed as if we merged manually.

## Exploring Additional Slash Commands

We saw some a few of the most commonly employed slash commands used through a typical pull request process. We'll expand on that next by creating a new PR and experimenting with a few other commands.

```
1 git checkout master  
2  
3 git pull  
4  
5 git checkout -b my-pr
```

We created a new branch called `my-pr`.

Next, we'll make a minor change to the source code and push it to the newly created branch. Otherwise, GitHub would not allow us to make a pull request if nothing changed.

```
1 echo "My PR" | tee README.md  
2  
3 git add .
```

```
4
5 git commit \
6   --message "My second PR with prow"
7
8 git push --set-upstream origin my-pr
```

We are finally ready to create a pull request.

```
1 jx create pullrequest \
2   --title "My PR" \
3   --body "What I can say?" \
4   --batch-mode
```

Please open the link from the output in your favorite browser.

You will notice that this time your colleague is automatically assigned as a reviewer. Prow took the list of reviewers from the `OWNERS` file and saw that there are only two available. Since you made the pull request, it decided to assign the other user as the reviewer. If wouldn't make sense to assign it to you anyways.

We can also observe that the system automatically added a label `size/XS`. It deduced that the changes we're proposing in this pull request are *extra small*. That is done through the Prow plugin `size`. While some plugins (e.g., `approve`) react to slash commands, others (e.g., `size`) are used as a reaction to other events (e.g., creation of a pull request).

Size labels are applied based on the total number of changed lines. Both new and deleted lines are counted, and the thresholds are as follows.

- `size/XS`: 0-9
- `size/S`: 10-29
- `size/M`: 30-99
- `size/L`: 100-499
- `size/XL`: 500-999
- `size/XXL`: 1000+

Since we rewrote `README.md` with a single line, the number of changed lines is one plus whatever was the number of lines in the file before we overwrote it. We know that up to nine lines were changed in total since we got the label `size/XS`.

Sometimes we might use a code generator and might want to exclude from the calculation the files it creates. In such a case, all we'd need to do is

place `.generated_files` to the project root.

Let's imagine that we reviewed the pull request and that we decided that we do not want to proceed in the near future. In such a case, we probably do not want to close it, but we still want to make sure that everyone is aware that it is on hold.

Please type `/hold` and click the *Comment* button. You'll see that the *do-not-merge/hold* label was added. Now it should be clear to everyone that the PR should not be merged. Hopefully, we would also add a comment that would explain the reasons behind such a decision. I will let your imagination kick in and let you compose it.

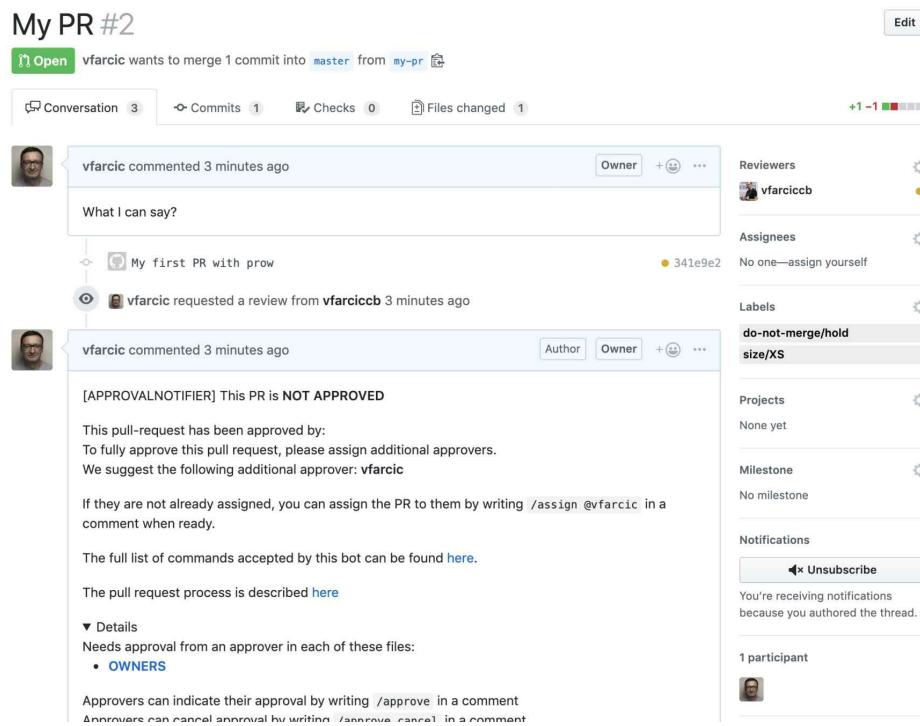


Figure 12-6: Pull request with Prow-created labels

At this moment, you might wonder whether it is more efficient to create comments with slash commands that add labels, instead of merely creating them directly. It is not. We could just as easily create a label using GitHub console. But the reason for applying ChatOps principles is not primarily efficiency. Instead, it is more focused on documenting and executing actions. When we write a comment with the `/hold` command, we did not only create a label, but we also recorded who did that (the person who wrote the comment), as well as when it was done. The comments serve as a ledger and can be used to infer the flow of actions and decisions. We can

find everything that happened to a pull request by reading the comments from top to bottom.

Now, let's say that the situation changed and that the pull request should not be on hold anymore. Many of the slash commands can be written with `cancel`. So, to remove the label, all we have to do is to write a new comment with `/hold cancel`. Try it out and confirm that the *do-not-merge/hold* label is removed.

All commonly performed actions with pull requests are supported. We can, for example, `/close` and `/reopen` a pull request. Both belong to the *lifecycle* plugin which also supports `/lifecycle frozen`, `/lifecycle stale`, and `/lifecycle rotten` commands that add appropriate labels. To remove those labels, all we have to do is add `remove` as the prefix (e.g., `/remove-lifecycle stale`).

Finally, if you want to bring a bit of life to your comments, try the `/meow` command.

Once you're done experimenting with the commands, please tell the approver to write a comment with the `/lgtm` command and the pull request will be merged to the master branch which, in turn, will initiate yet another Jenkins X build that will deploy a new release to the staging environment.

By now, you hopefully see the benefits of ChatOps, but you are probably wondering how to know which commands are available. We'll explore that next.

## How Do We Know Which Slash Commands Are Available?

The list of Prow plugins and their configuration is stored in ConfigMap `plugins`. Let's describe it and see what we'll get.

```
1 kubectl -n cd describe cm plugins
```

The interesting part is the `plugins` section. We can see that each repository we imported into Jenkins X contains a list of plugins. Jenkins X makes sure that the list is always up-to-date with the imported projects.

The output limited to `jx-prow` is as follows.

```
1 ...
2 plugins.yaml:
3 ----
4 ...
5 plugins:
6 ...
7   vfarcic/jx-prow:
8     - config-updater
9     - approve
10    - assign
11    - blunderbuss
12    - help
13    - hold
14    - lgtm
15    - lifecycle
16    - size
17    - trigger
18    - wip
19    - heart
20    - cat
21    - override
22 ...
```

We already used many of the available plugins (e.g., `approve`). The others (e.g., `config-updater`) do not react to slash commands but are triggered automatically. Feel free to explore the plugins we're using through the [Prow Plugin Catalog](#) documentation. The details of the commands provided through the plugins are available in [Command Help](#).

## What Now?

There is at least one crucial Prow-related subject we did not yet explore. We need to figure out how to change the Prow plugins we're using, as well as to find out how to modify the rules that govern the pull request process. We'll go through those and a few other Prow and ChatOps related subjects later. For now, we know how to use ChatOps capabilities that Jenkins X provides out of the box. That is a notable improvement by itself, so we'll leave tweaking for one of the next chapters.

Now you need to decide whether to continue using the cluster or to destroy it. If you choose to destroy it or to uninstall Jenkins X, you'll find the instructions at the bottom of the Gist you chose at the beginning of this chapter.

If you destroyed the cluster or you uninstalled Jenkins X, please remove the repositories and the local files we created. You can use the commands that follow for that.



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

```
1 cd ..
2
3 GH_USER=[...]
4
5 hub delete -y \
6   $GH_USER/environment-jx-rocks-staging
7
8 hub delete -y \
9   $GH_USER/environment-jx-rocks-production
10
11 hub delete -y $GH_USER/jx-prow
12
13 rm -rf jx-prow
14
15 rm -rf ~/.jx/environments/$GH_USER/environment-jx-rocks-*
```

# Using The Pipeline Extension Model



The examples in this chapter work only with serverless Jenkins X.

Jenkins X pipeline extension model is, in my opinion, one of the most exciting and innovative improvements we got with Jenkins X. It allows us to focus on what really matters in our projects and to ignore the steps that are common to others. Understanding the evolution of Jenkins pipelines is vital if we are to adopt the extension model. So, before we dive into extensions, we need to understand better how pipelines evolved over time.

## The Evolution Of Jenkins Jobs And How We Got To The YAML-Based `jenkins-x.yml` Format

When Jenkins appeared, its pipelines were called Freestyle jobs. There was no way to describe them in code, and they were not kept in version control. We were creating and maintaining those jobs through Jenkins UI by filling input fields, marking checkboxes, and selecting values from drop-down lists. The results were impossible-to-read XML files stored in the Jenkins home directory. Nevertheless, that approach was so great (compared to what existed at the time) that Jenkins became widely adopted overnight. But, that was many years ago and what was great over a decade ago is not necessarily as good today. As a matter of fact, Freestyle jobs are the antithesis of the types of jobs we should be writing today. Tools that create code through drag-and-drop methods are extinct. Not having code in version control is a cardinal sin. Not being able to use our favorite IDE or code editor is unacceptable. Hence, the Jenkins community created Jenkins pipelines.

Jenkins pipelines are an attempt to rethink how we define Jenkins jobs. Instead of the click-type-select approach of Freestyle jobs, pipelines are defined in Jenkins-specific Groovy-based domain specific language (DSL) written in Jenkinsfile and stored in code repositories together with the rest of the code of our applications. That approach managed to accomplish

many improvements when compared to Freestyle jobs. A single job could define the entire pipeline, the definition could be stored in the code repository, and we could avoid part of the repetition through the usage of Jenkins Shared Libraries. But, for some users, there was a big problem with Jenkins pipelines.

Using Groovy-based DSL was too big of a jump for some Jenkins users. Switching from click-type-select with Freestyle jobs into Groovy code was too much. We had a significant number of Jenkins users with years of experience in accomplishing results by clicking and selecting options, but without knowing how to write code.

I'll leave aside the discussion in which I'd argue that anyone working in the software industry should be able to write code no matter their role. Instead, I'll admit that having non-coders transition from UI-based into the code-only type of Jenkins jobs posed too much of a challenge. Even if we would agree that everyone in the software industry should know how to write code, there is still the issue of Groovy.

The chances are that Groovy is not your preferred programming language. You might be working with NodeJS, Go, Python, .Net, C, C++, Scala, or one of the myriads of other languages. Even if you are Java developer, Groovy might not be your favorite. Given that the syntax is somehow similar to Java and the fact that both use JVM does not diminish the fact that Groovy and Java are different languages. Jenkins DSL tried to "hide" Groovy, but that did not remove the fact that you had to know (at least basic) Groovy to write Jenkins pipelines. That meant that many had to choose whether to learn Groovy or to switch to something else. So, even though Jenkins pipelines were a significant improvement over Freestyle jobs, there was still work to be done to make them useful to anyone no matter their language preference. Hence, the community came up with the declarative pipeline.



Declarative pipelines are not really declarative. No matter the format, pipelines are by their nature sequential.

Declarative pipeline format is a simplified way to write Jenkinsfile definitions. To distinguish one from the other, we call the older pipeline

syntax scripted, and the newer declarative pipeline. We got much-needed simplicity. There was no Groovy to learn unless we employ shared libraries. The new pipeline format was simple to learn and easy to write. It served us well, and it was adopted by static Jenkins X. And yet, serverless Jenkins X introduced another change to the format of pipeline definitions. Today, we can think of static Jenkins X with declarative pipelines as a transition towards serverless Jenkins X.

With serverless Jenkins X, we moved into pipelines defined in YAML. Why did we do that? One argument could be that YAML is easier to understand and manage. Another could claim that YAML is the golden-standard for any type of definition, Kubernetes resources being one example. Most of the other tools, especially newer ones, switched to YAML definitions. While those and many other explanations are valid and certainly played a role in making the decision to switch to YAML, I believe that we should look at the change from a different angle.

All Jenkins formats for defining pipelines were based on the fact that it will be Jenkins who will execute them. Freestyle jobs used XML because Jenkins uses XML to store all sorts of information. A long time ago, when Jenkins was created, XML was all the rage. Scripted pipelines use Groovy DSL because pipelines need to interact with Jenkins. Since it is written in Java, Groovy is a natural choice. It is more dynamic than Java. It compiles at runtime allowing us to use it as a scripting mechanism. It can access Jenkins libraries written in Java, and it can access Jenkins itself at runtime. Then we added declarative pipeline that is something between Groovy DSL and YAML. It is a wrapper around the scripted pipeline.

What all those formats have in common is that they are all limited by Jenkins' architecture. And now, with serverless Jenkins X, there is no Jenkins any more.

Saying that Jenkins is gone is not entirely correct. Jenkins lives in Jenkins X. The foundation that served us well is there. The experience from many years of being the leading CI/CD platform was combined with the need to solve challenges that did not exist before. We had to come up with a platform that is Kubernetes-first, cloud-native, fault-tolerant, highly-available, lightweight, with API-first design, and so on and so forth. The result is Jenkins X or, to be more precise, its serverless flavor. It combines some of the best tools on the market with custom code written specifically

for Jenkins X. And the result is what we call serverless or next generation Jenkins X.

The first generation of Jenkins X (static flavor) reduced the “traditional” Jenkins to a bare minimum. That allowed the community to focus on building all the new tools needed to bring Jenkins to the next level. It reduced Jenkins’ surface and added a lot of new code around it. At the same time, static Jenkins X maintains compatibility with the “traditional” Jenkins. Teams can move to Jenkins X without having to rewrite everything they had before while, at the same time, keeping some level of familiarity.

Serverless Jenkins X is the next stage in the evolution. While static flavor reduced the role of the “traditional” Jenkins, serverless eradicated it. The end result is a combination of Prow, Jenkins Pipeline Operator, Tekton, and quite a few other tools and processes. Some of them (e.g., Prow, Tekton) are open-source projects bundled into Jenkins X while others (e.g., Jenkins Pipeline Operator) are written from scratch. On top of those, we got `jx` as the CLI that allows us to control any aspect of Jenkins X.

Given that there is no “traditional” Jenkins in the serverless flavor of Jenkins X, there is no need to stick with the old formats to define pipelines. Those that do need to continue using `Jenkinsfile` can do so by using static Jenkins X. Those who want to get the most benefit from the new platform will appreciate the benefits of the new YAML-based format defined in `jenkins-x.yml`. More often than not, organizations will combine both. There are use cases when static Jenkins with the support for `Jenkinsfile` is a good choice, especially in cases when projects already have pipelines running in the “traditional” Jenkins. On the other hand, new projects can be created directly in serverless Jenkins X and use `jenkins-x.yml` to define pipelines.

Unless you just started a new company, there are all sorts of situations and some of them might be better fulfilled with static Jenkins X and `Jenkinsfile`, others with serverless Jenkins X and `jenkins-x.yml`, while there are likely going to exist projects that started a long time ago and do not see enough benefit to change. Those can stay in “traditional” Jenkins running outside Kubernetes or in any other tool they might be using.

To summarize, static Jenkins is a “transition” between the “traditional” Jenkins and the final solution (serverless flavor). The former has reduced

Jenkins to a bare minimum, while the latter consists of entirely new code written specifically to solve problems we're facing today and leveraging all the latest and greatest technology can offer.

So, Jenkins served us well, and it will continue living for a long time since many applications were written a while ago and might not be good candidates to embrace Kubernetes. Static Jenkins X is for all those who want to transition to Kubernetes without losing all their investment (e.g., Jenkinsfile). Serverless Jenkins X is for all those who seek the full power of Kubernetes and want to be genuinely cloud-native.



For now (April 2019), serverless Jenkins X works only with GitHub. Until that is corrected, using any other Git platform might be yet another reason to stick with static Jenkins X. The rest of the text will assume that you do use GitHub or that you can wait for a while longer until the support for other Git platforms is added.

Long story short, Serverless Jenkins X uses YAML in jenkins-x.yml to describe pipelines, while more traditional Jenkins as well as static Jenkins X rely on Groovy DSL defined in Jenkinsfile. Freestyle jobs are deprecated for quite a long time, so we'll ignore their existence. Whether you prefer Jenkinsfile or jenkins-x.yml format will depend on quite a few factors, so let's break down those that matter the most.

If you already use Jenkins, you are likely used to Jenkinsfile and might want to keep it for a while longer.

If you have complex pipelines, you might want to stick with the scripted pipeline in Jenkinsfile. That being said, I do not believe that anyone should have complex pipelines. Those that do usually tried to solve problems in the wrong place. Pipelines (of any kind) are not supposed to have complex logic. Instead, they should define orchestration of automated tasks defined somewhere else. For example, instead of having tens or hundreds of lines of pipeline code that defines how to deploy our application, we should move that logic into a script and simply invoke it from the pipeline. That logic is similar to what `jx promote` does. It performs semi-complex logic, but from the pipeline point of view it is a simple step with a single command. If we do adopt that approach, there is no need for complex pipelines and, therefore, there is no need for Jenkins' scripted pipeline. Declarative is more than enough when using Jenkinsfile.

If you do want to leverage all the latest and greatest that Jenkins X (serverless flavor) brings, you should switch to YAML format defined in the `jenkins-x.yml` file.

Therefore, use `Jenkinsfile` with static Jenkins X if you already have pipelines and you do not want to rewrite them. Stop using scripted pipelines as an excuse for misplaced complexity. Adopt serverless Jenkins X with YAML-based format for all other cases.

But you probably know all that by now. You might be even wondering why do we go through history lessons now? The reason for the reminiscence lies in the “real” subject I want to discuss. We’ll talk about *code repetition*, and we had to set the scene for what’s coming next.

## Getting Rid Of Repetition

Copying and pasting code is a major sin among developers. One of the first things we learn as software engineers is that duplicated code is hard to maintain and prone to errors. That’s why we are creating libraries. We do not want to repeat ourselves, so we even came up with a commonly used acronym DRY (don’t repeat yourself).

Having that in mind, all I can say is that Jenkins users are sinful.

When we create pipelines through static Jenkins X, every project gets a `Jenkinsfile` based on the pipeline residing in the build pack we chose. If we have ten projects, there will be ten identical copies of the same `Jenkinsfile`. Over time, we’ll modify those `Jenkinsfiles`, and they might not all be precisely the same. Even in those cases, most of the `Jenkinsfile` contents will remain untouched. It does not matter whether 100% of `Jenkinsfile` is repeated across projects or it that number drops to 25%. There is a high level of repetition.

In the past, we fought such repetition through shared libraries. We would encapsulate repeated lines into Groovy libraries and invoke them from any pipeline that needs to use those features. But we abandoned that approach with Jenkins X since Jenkins shared libraries have quite a few deficiencies. They can be written only in Groovy, and that might not be a language everyone wants to learn. They cannot be (easily) tested in isolation. We’d need to run a Jenkins pipeline that invokes the library as a way of testing it. Finally, we could not (easily) run them locally (without Jenkins).

While shared libraries can be used with static Jenkins X, we probably should not go down that route. Instead, I believe that a much better way to encapsulate features is by writing executables. Instead of being limited to Groovy, we can write an executable in Bash, Go, Python, or any other language that allows us to execute code. Such executables (usually scripts) can be easily tested locally, they can be used by developers with Jenkins X, and can be executed from inside pipelines. If you take another look at any Jenkins X pipeline, you'll see that there are no plugins, and there are no shared libraries. It's mostly a series of `sh` commands that execute Shell commands (e.g., `cat`, `jx`, etc.). Such pipelines are easy to understand, easy to run with or without Jenkins X (e.g., on a laptop), and easy to maintain. Both plugins and shared libraries are dead.

Why do I believe that plugins are dead? To answer that question we need to take a look behind the reasons for their existence. Most plugins are created either to isolate users from even basic commands or because the applications they integrate with do not have decent API.

Isolating anyone from basic commands is just silly. For example, using a plugin that will build a Docker image instead of merely executing `docker image build` is beyond comprehension. On the other hand, if we do need to integrate with an application that does not have an API and CLI, we are better off throwing that application to thrash. It's not 1999 anymore. Every application has a good API, and most have a CLI. Those that don't are unworthy our attention.

So, there are no more plugins, and we should not use shared libraries. All repetitive features should be in executables (e.g., script). With that in mind, do we still have repetition? We do. Even if all the features (e.g., deployment to production) are in scripts reused across pipelines, we are still bound to repeat a lot of orchestration code.

A good example is Jenkinsfile pipelines created when we import a project or create a new one through the `jx create quickstart` command. Each project gets around 80 lines of Jenkinsfile. All those based on the same language will have exactly the same Jenkinsfile. Even those based on different programming languages will be mostly the same. They all have to check out the code, to create release notes, to run some form of tests, to deploy releases to some environments, and so on and so forth. All those lines in Jenkinsfile only deal with orchestration since most of the features are in executables (e.g., `jx promote`). Otherwise, we'd jump from around

80 to hundreds or even thousands of lines of repetitive code. Now, even if half of those 80 lines are repeated, that's still 40 lines of repetition. That is not bad by itself. However, we are likely going to have a hard time if we need to apply a fix or change the logic across multiple projects.

The serverless flavor of Jenkins X solves the problem of unnecessary repetition through the **pipeline extension model**. We'll see it in action soon. For now, we need a cluster with Jenkins X up-and-running.

## Creating A Kubernetes Cluster With Jenkins X

You can skip this section if you kept the cluster from the previous chapter and it contains serverless Jenkins X. Otherwise, we'll need to create a new Jenkins X cluster.



All the commands from this chapter are available in the [13-pipeline-extension-model.sh](#) Gist.

For your convenience, the Gists that will create a new serverless Jenkins X cluster or install it inside an existing one are as follows.

- Create a new serverless **GKE** cluster: [gke-jx-serverless.sh](#)
- Create a new serverless **EKS** cluster: [eks-jx-serverless.sh](#)
- Create a new serverless **AKS** cluster: [aks-jx-serverless.sh](#)
- Use an **existing** serverless cluster: [install-serverless.sh](#)

We will not need the `jx-prow` project we created in the previous chapter. If you are reusing the cluster and Jenkins X installation, you might want to remove it and save a bit of resources.



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

```
1 GH_USER=[...]
2
3 jx delete application \
4   ${GH_USER}/jx-prow \
5   --batch-mode
```



The commands that follow will reset your `go-demo-6` master with the contents of the versioning branch that contains all the changes we did so far. Please execute them only if you are unsure whether you did all the exercises correctly.

```
1 cd go-demo-6
2
3 git pull
4
5 git checkout versioning-tekton
6
7 git merge -s ours master --no-edit
8
9 git checkout master
10
11 git merge versioning-tekton
12
13 git push
14
15 cd ..
```

If you ever restored a branch at the beginning of a chapter, the chances are that there is a reference to my user (`vfarcic`). We'll change that to Google project since that's what Knative will expect to be the location of the container images.



Please execute the commands that follow only if you are using **GKE** and if you ever restored a branch at the beginning of a chapter (like in the snippet above).

```
1 cd go-demo-6
2
3 cat charts/go-demo-6/Makefile \
4   | sed -e \
5     "s@vfarcic@$PROJECT@g" \
6   | tee charts/go-demo-6/Makefile
7
8 cat charts/preview/Makefile \
9   | sed -e \
10    "s@vfarcic@$PROJECT@g" \
11   | tee charts/preview/Makefile
12
13 cat skaffold.yaml \
14   | sed -e \
15     "s@vfarcic@$PROJECT@g" \
16   | tee skaffold.yaml
17
18 cd ..
```



If you destroyed the cluster at the end of the previous chapter, you'll need to import the `go-demo-6` application again. Please execute the commands that follow only if you created a new cluster specifically for the exercises from this chapter.

```
1 jx import --batch-mode
2
3 jx get activities \
4   --filter go-demo-6 \
5   --watch
6
7 cd ..
```

Now we can explore Jenkins X Pipeline Extension Model.

## Exploring Build Pack Pipelines

As a reminder, we'll take another quick look at the `jenkins-x.yml` file.

```
1 cat jenkins-x.yml
```

The output is as follows.

```
1 buildPack: go
```

The pipeline is as short as it can be. It tells Jenkins that it should use the pipeline from the build pack `go`.

We already explored build packs and learned how we can extend them to fit our specific needs. But, we skipped discussing one of the key files in build packs. I intentionally avoided talking about the `pipeline.yaml` file because it uses the new format introduced in serverless Jenkins X. Given that at the time we explored build packs we did not yet know about the existence of serverless Jenkins X, it would have been too early to discuss new pipelines. Now that you are getting familiar with the serverless flavor, we can go back and explore `pipeline.yaml` located in each of the build packs.

```
1 open "https://github.com/jenkins-x-buildpacks/jenkins-x-kubernetes"
```

Open `packs` followed with the `go` directory. We can see that `pipeline.yaml` is one of the build pack files, so let's take a closer look.

```
1 curl "https://raw.githubusercontent.com/jenkins-x-buildpacks/jenkins-x-kubernetes/
2 ster/packs/go/pipeline.yaml"
```

If you remember Jenkinsfile we used before, you'll notice that this pipeline is functionally the same. However, it is written in a different format. It's the same format used for pipelines in serverless Jenkins X. If we import that build pack into static Jenkins X, it gets translated into Jenkinsfile, mainly for compatibility reasons. On the other hand, if we are using serverless Jenkins X, it does not get translated into anything. Instead, we get the `jenkins-x.yml` file with the single line `buildPack: go` that tells the system to use the pipeline from the build pack, instead of copying it into the application repository.

Let us quickly digest the key sections of the `pipeline.yaml` file. It starts with the `extends` instruction that is as follows.

```
1 extends:
2   import: classic
3   file: go/pipeline.yaml
```

The `extends` section is similar to how we extend libraries in most programming languages. It tells the system that it should use the build pack `go/pipeline.yaml` from the `classic` mode. That's the one we'd use if we choose `Library Workloads: CI+Release but no CD` when we installed Jenkins X. Those pipelines are meant for the flows that do not involve deployments. Since there is an overlap between `Library` and `Kubernetes` and we do not want to repeat ourselves, the `Kubernetes` one extends the `Library` (the name `classic` might be misleading).

Further down is the collection of pipelines. The output, limited to the pipeline names is as follows.

```
1 ...
2 pipelines:
3   pullRequest:
4     ...
5   release:
6     ...
```

There are three types of pipelines; `pullRequest`, `release`, and `feature`. In the definition we see in front of us there are only the first two since `feature` pipelines are not very common.

It should be evident that the `pullRequest` pipeline is executed when we create PR. The `release` is run when we push or merge something into the

master branch. Finally, `feature` is used with long term feature branches. Since trunk based or short term branches are the recommended model, `feature` is not included in build pack pipelines. You can add it yourself if you do prefer the model of using long-term branches

Each pipeline can have one of the following lifecycles: `setup`, `setversion`, `prebuild`, `build`, `postbuild`, and `promote`. If we go back to the definition in front of us, the output, limited to the lifecycles of the `pullRequest` pipeline, is as follows.

```
1 ...
2 pipelines:
3   pullRequest:
4     build:
5       ...
6     postBuild:
7       ...
8     promote:
9       ...
```

In the YAML we're exploring, the `pullRequest` pipeline has lifecycles `build`, `postbuild`, and `promote`, while `release` has only `build` and `promote`.

Inside a lifecycle is any number of steps containing `sh` (the command) and an optional `name`, `comment`, and a few other instructions.

As an example, the full definition of the `build` lifecycle of the `pullRequest` pipeline is as follows.

```
1 ...
2 pipelines:
3   pullRequest:
4     build:
5       steps:
6         - sh: export VERSION=$PREVIEW_VERSION && skaffold build -f skaffold.yaml
7           name: container-build
8 ...
```

Do not worry if the new pipeline format is confusing. It is very straightforward, and we'll explore it in more depth soon.

Before we move on, if you were wondering whether does the extension of the `classic` pipeline come from, it is in the `jenkins-x-buildpacks/jenkins-x-classic` repository. The one used with Go can be retrieved through the command that follows.

```
1 curl "https://raw.githubusercontent.com/jenkins-x-buildpacks/jenkins-x-classic/master/packs/go/pipeline.yaml"
```

The classic pipeline that serves as the base follows the same logic with pipelines, lifecycle, and steps. The significant difference is that it introduces a few additional instructions like `comment` and `when`. I'm sure you can get what the former does. The latter (`when`) is a conditional that defines whether the step should be executed in static (`!prow`) or in serverless (`prow`) Jenkins X.

As you can see, even though the new format for pipelines is used directly only in serverless Jenkins X, it is vital to understand it even if you're using the static flavor. Pipelines defined in build packs are using the new format and translating it into `Jenkinsfile` used with static Jenkins X, or extending them if we're using the serverless flavor. If you decide to extend community-maintained buildpacks, you will need to know how the new YAML-based format works, even if the end result will be `Jenkinsfile`.

By now, the first activity of the newly imported `go-demo-6` project should have finished. Let's take a look at the result.

```
1 jx get activities \
2   --filter go-demo-6 \
3   --watch
```

The output is as follows.

STEP	STARTED AGO	DURATION	STATUS	Vers
2 vfarcic/go-demo-6/master #1	6m57s	5m2s	Succeeded	
3 : 1.0.56				
4 from build pack	6m57s	5m2s	Succeeded	
5 Credential Initializer Pmvfj	6m57s	0s	Succeeded	
6 Git Source Vfarcic Go Demo 6 Master Nck5w	6m55s	0s	Succeeded	http://github.com/vfarcic/go-demo-6
7 /github.com/vfarcic/go-demo-6				
8 Place Tools	6m52s	0s	Succeeded	
9 Git Merge	5m41s	0s	Succeeded	
10 Setup Jx Git Credentials	3m51s	1s	Succeeded	
11 Build Make Build	3m49s	16s	Succeeded	
12 Build Container Build	3m30s	4s	Succeeded	
13 Build Post Build	3m24s	0s	Succeeded	
14 Promote Changelog	3m23s	6s	Succeeded	
15 Promote Helm Release	3m16s	9s	Succeeded	
16 Promote Jx Promote	3m6s	1m11s	Succeeded	
17 Promote: staging	3m2s	1m7s	Succeeded	
18 PullRequest	3m2s	1m7s	Succeeded	Pull
19 quest: https://github.com/vfarcic/environment-jx-rocks-staging/pull/1			Merge	SHA: d094a9803681dab1d0ae7b8501eac10e7703a
20 Update	1m55s	0s	Succeeded	

We saw a similar output many times before and you might be wondering where does Jenkins X gets the names of the activity steps. They are a

combination of the lifecycle and the name of a step from the pipeline which, in this case, is defined in the build pack. That's the same pipeline we explored previously. We can see all that from the output of the activity. It states that the pipeline is `from build pack`. Further down are the steps.

The first few steps are not defined in any pipeline. No matter what we specify, Jenkins X will perform a few setup tasks. The rest of the steps in the activity output is a combination of the lifecycle (e.g., `Promote`) and the step name (e.g., `jx-promote > Jx Promote`).

Now that we demystified the meaning of `buildPack: go` in our pipeline, you are probably wondering how to extend it.

## Extending Build Pack Pipelines

We already saw that pipelines based on the new format have a single line `buildPack: go`. To be more precise, those that are created based on a build pack are like that. While you can certainly create a pipeline from scratch, most use cases benefit from having a good base inherited from a build pack. For some, the base will be everything they need, but for others it will not. There is a high probability that you will need to extend those pipelines by adding your own steps or even to replace a whole lifecycle (e.g., `promote`). Our next mission is to figure out how to accomplish that. We'll explore how to extend pipelines used with serverless Jenkins X.

As any good developer (excluding those who work directly with the master branch), we'll start by creating a new branch.

```
1 git checkout -b extension
```

Since we need to make a change that will demonstrate how pipelines work, instead of making a minor modification like adding a random text to the `README.md` file, we'll do something we should have done a long time ago. We'll increase the number of replicas of our application. We'll ignore the fact that we should probably create a `HorizontalPodAutoscaler` and simply increase the `replicaCount` value in `values.yaml` from 1 to 3.

Given that I want to make it easy for you, we'll execute a command that will make a change instead of asking you to open the `values.yaml` file in your favorite editor and update it manually.

```
1 cat charts/go-demo-6/values.yaml \
2   | sed -e \
3     's@replicaCount: 1@replicaCount: 3@g' \
4   | tee charts/go-demo-6/values.yaml
```

We'll need to make another set of changes. The reason will become apparent later. For now, please note that the tests have a hard-coded `http://` followed with a variable that is used to specify the IP or the domain. As you'll see soon, we'll fetch a fully qualified address so we'll need to remove `http://` from the tests.

```
1 cat functional_test.go \
2   | sed -e \
3     's@fmt.Sprintf("http://@fmt.Sprintf("@g' \
4   | tee functional_test.go
5
6 cat production_test.go \
7   | sed -e \
8     's@fmt.Sprintf("http://@fmt.Sprintf("@g' \
9   | tee production_test.go
```

Now that we fixed the problem with the tests and increased the number of replicas (even though that was not necessary for the examples we'll run), we can proceed and start extending our out-of-the-box pipeline.

As it is now, the pipeline inherited from the build pack does not contain validations. It does almost everything else we need it to do except to run tests. We'll start with unit tests.

Where should we add unit tests? We need to choose the pipeline, the lifecycle, and the mode. Typically, this would be the moment when I'd need to explain in greater detail the syntax of the new pipeline format. Given that would be a lengthy process, we'll take a shortcut provided with the `jx create step` command that will help us make the right choices and update `jenkins-x.yml` for us.

```
1 jx create step
```

Now we need to answer a series of questions. The first one is to pick the pipeline kind. We can choose between `release`, `pullrequest`, and `feature`. We already explored under which conditions each of those are executed, so all we have to do is pick one. Unit tests should probably be executed when we create a pull request, so please choose `pullrequest` and press the enter key.

Next, we need to pick the lifecycle. Each time a pipeline is executed, it goes through a series of lifecycles. The `setup` lifecycle prepares the pipeline environment, `setversion` configures the version of the release, `prebuild` prepares the environment for the build steps, `build` builds the binaries and other artifacts, `postbuild` is usually used for additional validations like security scanning, and, finally, `promote` executes the process of installing the release to one or more environments.

In most cases, running unit tests does not require compilation nor a live application, so we can execute them early. The right moment is probably just before we build the binaries. That might compel you to think that we should select the `prebuild` lifecycle assuming that building is done in the `build` phase. That would be true only if we would be choosing the lifecycle we want to replace. While that is possible as well, we'll opt for extending one of the phases. So, please select `build` and press the enter key.

We already commented that we want to extend the existing `build` lifecycle, so the next question comes just in time. We are asked to pick the `create mode`. We could choose between `pre`, `post`, and `replace`. The `pre` mode would add a new step before those in the `build` lifecycle inherited from the build pack. Similarly, `post` would add a new step after. Finally, we can use the `replace` mode to replace the steps inherited from the build pack. Since we already argued that we should execute our unit tests even before we build the artifacts, please select the `pre` mode and press the enter key.

We came to the last question asking us to specify the `command` for the new step. We already went through the exercises of adding unit tests to Jenkinsfile in static Jenkins X, so you probably remember the command. If you don't, here's the reminder. Please type `make unittest` and press the enter key.

The final output from which you can see all the choices we selected is as follows.

```
1 ? Pick the pipeline kind: pullrequest
2 ? Pick the lifecycle: build
3 ? Pick the create mode: pre
4 ? Command for the new step: make unittest
5 Updated Jenkins X Pipeline file: jenkins-x.yml
```

That's it. We added a new step to our `jenkins-x.yml`, and we can see from the output that the command updated Jenkins X Pipeline file.

Let's see what we got.

```
1 cat jenkins-x.yml
```

The output is as follows.

```
1 buildPack: go
2 pipelineConfig:
3   pipelines:
4     pullRequest:
5       build:
6         preSteps:
7           - command: make unittest
```

We can see that the `buildPack: go` is still there so our pipeline will continue doing whatever is defined in that build pack. Below is the `pipelineConfig` section that, in this context, extends the one defined in build pack. The `agent` is empty (`{}`), so it will continue using the agent defined in the build pack.

The `pipelines` section extends the `build` lifecycle of the `pullRequest` pipeline. By specifying `preStep`, we know that it will execute `make unittest` before any of the out-of-the-box steps defined in the same lifecycle.

Before we proceed, please note that the `jx create step` command does not offer all the options we can specify in our pipelines. At the time of this writing (May 2019) it provides a convenient way to define the most commonly used elements (`pipeline`, `lifecycle`, `mode`, and the `command`). That's all we need for most use cases. Nevertheless, we will go through more advanced or, to be more precise, less commonly used constructs later. For now, what `jx create step` offers should be enough.

Now that we extended our pipeline, we'll push the changes to GitHub, create a pull request, and observe the outcome.

```
1 git add .
2
3 git commit \
4   --message "Trying to extend the pipeline"
5
6 git push --set-upstream origin extension
7
8 jx create pullrequest \
9   --title "Extensions" \
```

```
10      --body "What I can say?" \
11      --batch-mode
```

The last command created a pull request, and the address should be in the output. We'll put that URL together with the name of the branch into environment variables since we'll need them later.



Please replace the first [...] with the full address of the pull request (e.g., <https://github.com/vfarcic/go-demo-6/pull/56>) and the second with PR-[PR\_ID] (e.g., PR-56). You can extract the ID from the last segment of the pull request address.

```
1 PR_ADDR=[...] # e.g., `https://github.com/vfarcic/go-demo-6/pull/56`  
2  
3 BRANCH=[...] # e.g., `PR-56`
```

The easiest way to deduce whether our extended pipeline works correctly is through logs.

```
1 jx get build logs \
2     --filter go-demo-6 \
3     --branch $BRANCH
```

If you get the error: no Tekton pipelines have been triggered which match the current filter message, you were probably too fast, and the pipeline did not yet start running. In that case, wait for a few moments and re-execute the `jx get build logs` command.

The output is too big to be presented here. What matters is the part that follows.

```
1 ...
2 === RUN   TestMainUnitTestSuite
3 === RUN   TestMainUnitTestSuite/Test_HelloServer_Waits_WhenDelayIsPresent
4 === RUN   TestMainUnitTestSuite/Test_HelloServer_WritesHelloWorld
5 === RUN   TestMainUnitTestSuite/Test_HelloServer_WritesNokEventually
6 === RUN   TestMainUnitTestSuite/Test_HelloServer_WritesOk
7 === RUN   TestMainUnitTestSuite/Test_PersonServer_InvokesUpsertId_WhenPutPerson
8 === RUN   TestMainUnitTestSuite/Test_PersonServer_Panics_WhenFindReturnsError
9 === RUN   TestMainUnitTestSuite/Test_PersonServer_Panics_WhenUpsertIdReturnsError
10 === RUN  TestMainUnitTestSuite/Test_PersonServer_WritesPeople
11 === RUN  TestMainUnitTestSuite/Test_RunServer_InvokesListenAndServe
12 === RUN  TestMainUnitTestSuite/Test_SetupMetrics_InitializesHistogram
13 --- PASS: TestMainUnitTestSuite (0.01s)
14     --- PASS: TestMainUnitTestSuite/Test_HelloServer_Waits_WhenDelayIsPresent (0.
15     --- PASS: TestMainUnitTestSuite/Test_HelloServer_WritesHelloWorld (0.00s)
16     --- PASS: TestMainUnitTestSuite/Test_HelloServer_WritesNokEventually (0.00s)
17     --- PASS: TestMainUnitTestSuite/Test_HelloServer_WritesOk (0.00s)
18     --- PASS: TestMainUnitTestSuite/Test_PersonServer_InvokesUpsertId_WhenPutPers
19 (0.00s)
```

```

20      --- PASS: TestMainUnitTestSuite/Test_PersonServer_Panics_WhenFindReturnsError
21 .00s)
22      --- PASS: TestMainUnitTestSuite/Test_PersonServer_Panics_WhenUpsertIdReturnsE
23 r (0.00s)
24      --- PASS: TestMainUnitTestSuite/Test_PersonServer_WritesPeople (0.00s)
25      --- PASS: TestMainUnitTestSuite/Test_RunServer_InvokesListenAndServe (0.00s)
26      --- PASS: TestMainUnitTestSuite/Test_SetupMetrics_InitializesHistogram (0.00s
27 PASS
28 ok      go-demo-6          0.011s
29 ...

```

We can see that our unit tests are indeed executed.

In our context, what truly matters is the output after the tests. It shows that the application binary and container images were built. That proves that the new step did not replace anything, but that it was added among those defined in the build pack. Since we specified the mode as part of `preSteps`, it was added before the existing steps in the `build` lifecycle.

As I'm sure you already know, unit tests are often not enough and we should add some form of tests against the live application. We should probably add functional tests to the pipeline. But, before we do that, we need to create one more target in the Makefile.



Remember what we said before about `Makefile`. It expects tabs as indentation. Please make sure that the command that follows is indeed using tabs and not spaces, if you're typing the commands instead of copying and pasting from the Gist.

```

1 echo 'functest:
2     CGO_ENABLED=$(CGO_ENABLED) $(GO) \\
3     test -test.v --run FunctionalTest \\
4     --cover
5 ' | tee -a Makefile

```

Now we can add the `functest` target to the `pullrequest` pipeline. But, before we do that, we need to decide when is the best moment to run them. Since they require a new release to be installed, and we already know from the past experience that installations and upgrades are performed through promotions, we should already have an idea where to put them. So, we'll add functional tests to the `pullrequest` pipeline, into the `promote` lifecycle, and with the `post` mode. That way, those tests will run after the promotion is finished, and our release based on the PR is up-and-running.

But, there is a problem that we need to solve or, to be more precise, there is an improvement we could do.

Functional tests need to know the address of the application under tests. Since each pull request is deployed into its own namespace and the app is exposed through a dynamically created address, we need to figure out how to retrieve that URL. In the past, when we used Jenkinsfile, we had a command that combines quite a few environment variables (e.g., ORG, HELM\_RELEASE, etc.) and retrieves the address by querying Ingress. Fortunately, there is a command that allows us to retrieve the address of the current pull request. The bad news is that we cannot (easily) run it locally, so you'll need to trust me when I say that `jx get preview --current` will return the full address of the PR deployed with the `jx preview` command executed as the last step in the `promote` lifecycle of the `pullrequest` pipeline.

Finally, we might want to skip the questions and provide all the answers by executing `jx create step` in batch mode.

Having all that in mind, the command we will execute is as follows.

```
1 jx create step \
2   --pipeline pullrequest \
3   --lifecycle promote \
4   --mode post \
5   --sh 'ADDRESS=`jx get preview --current 2>&1` make functest'
```

Just as before, the output confirms that `jenkins-x.yml` was updated, so let's take a peek at how it looks now.

```
1 cat jenkins-x.yml
```

The output is as follows.

```
1 buildPack: go
2 pipelineConfig:
3   pipelines:
4     pullRequest:
5       build:
6         preSteps:
7           - command: make unittest
8       promote:
9         steps:
10           - command: ADDRESS=`jx get preview --current 2>&1` make functest
```

As you can see, the new step follows the same pattern. It is defined inside the `pullRequest` pipeline as the `promote` lifecycle and inside the `steps`

mode. You can easily conclude that `preSteps` are executed before those defined in the same lifecycle of the build pack, and `steps` are running after.

Now, let's push the change before we confirm that everything works as expected.

```
1 git add .
2
3 git commit \
4     --message "Trying to extend the pipeline"
5
6 git push
```

Please wait for a few moments until the new pipeline run starts, before we retrieve the logs.

```
1 jx get build logs \
2     --filter go-demo-6 \
3     --branch $BRANCH
```

You should be presented with a choice of pipeline runs. The choices should be as follows.

```
1 > vfarcic/go-demo-6/PR-64 #2 serverless-jenkins
2   vfarcic/go-demo-6/PR-64 #1 serverless-jenkins
```

If you do not see the new run, please cancel the command with `ctrl+c`, wait for a while longer, and re-execute the `jx get build logs` command.

The result of the functional tests should be near the bottom of the output. You should see something similar to the output that follows.

```
1 ...
2 CGO_ENABLED=0 GO15VENDOREXPERIMENT=1 go \
3 test -test.v --run FunctionalTest \
4 --cover
5 === RUN TestFunctionalTestSuite
6 === RUN TestFunctionalTestSuite/Test_Hello_ReturnsStatus200
7 2019/04/26 10:58:31 Sending a request to http://go-demo-6.jx-vfarcic-go-demo-6-pr
8 .34.74.193.252.nip.io/demo/hello
9 === RUN TestFunctionalTestSuite/Test_Person_ReturnsStatus200
10 2019/04/26 10:58:31 Sending a request to http://go-demo-6.jx-vfarcic-go-demo-6-pr
11 .34.74.193.252.nip.io/demo/person
12 --- PASS: TestFunctionalTestSuite (0.26s)
13     --- PASS: TestFunctionalTestSuite/Test_Hello_ReturnsStatus200 (0.13s)
14     --- PASS: TestFunctionalTestSuite/Test_Person_ReturnsStatus200 (0.13s)
15 PASS
16 coverage: 1.4% of statements
17 ok      go-demo-6      0.271s
```

While we are still exploring the basics of extending build pack pipelines, we might just as well take a quick look at what happens if a pipeline run fails. Instead of deliberately introducing a bug in the code of the application, we'll add another round of tests, but this time in a way that will certainly fail.

```
1 jx create step \
2   --pipeline pullrequest \
3   --lifecycle promote \
4   --mode post \
5   --sh 'ADDRESS=http://this-domain-does-not-exist.com make functest'
```

As you can see, we added the execution of the same tests. The only difference is that the address of the application under test is now `http://this-domain-does-not-exist.com`. That will surely fail and allow us to see what happens when something goes wrong.

Let's push the changes.

```
1 git add .
2
3 git commit \
4   --message "Added sully tests"
5
6 git push
```

Just as before, we need to wait for a few moments until the new pipeline run starts, before we retrieve the logs.

```
1 jx get build logs \
2   --filter go-demo-6 \
3   --branch $BRANCH
```

If the previous run was #2, you should be presented with a choice to select run #3. Please do so.

At the very bottom of the output, you should see that the pipeline run failed. That should not be a surprise since we specified an address that does not exist. We wanted it to fail, but not so that we can see that in logs. Instead, the goal is to see how we would be notified that something went wrong.

Let's see what we get if we open pull request in GitHub.

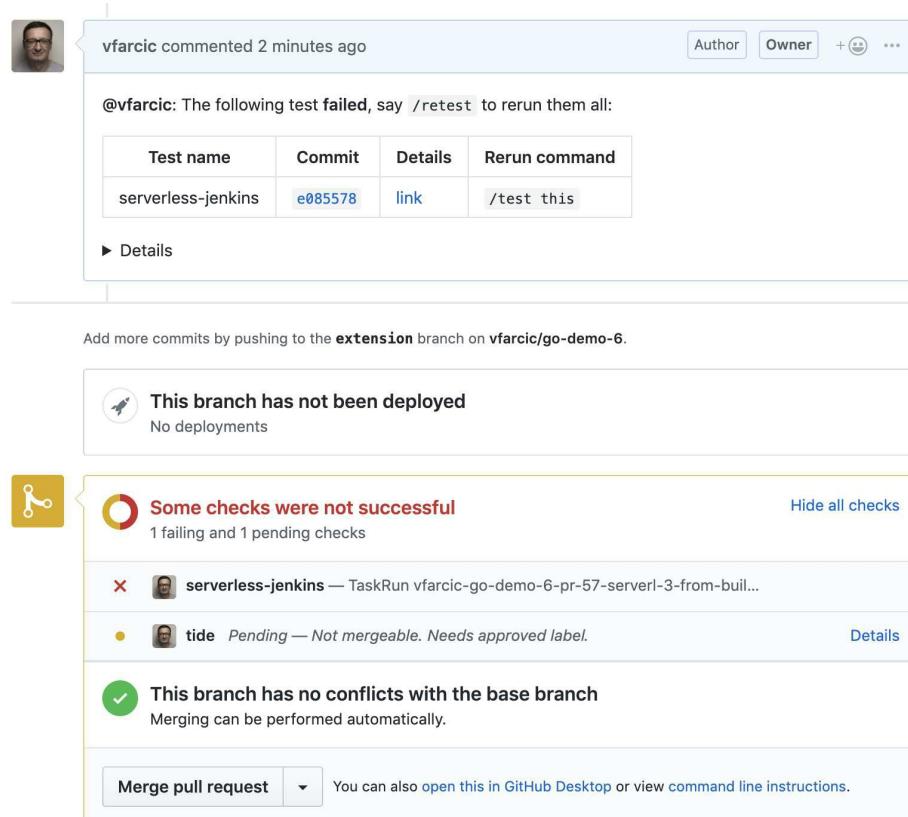
```
1 open "$PR_ADDR"
```

We can see the whole history of everything that happened to that pull request, including the last comment that states that the `serverless-jenkins` test failed. Later on, we'll explore how to run multiple pipelines in parallel. For now, it might be important to note that GitHub does not know which specific test failed, but that it treats the entire pipeline as a single unit. Nevertheless, it is recorded in GitHub that there is an issue. Its own notification mechanism should send emails to all interested parties, and we can see from the logs what went wrong. Later on we'll explore the new UI designed specifically for Jenkins X. For now, we can get all the information we need without it.

You will also notice that the comment states that we can re-run tests by writing `/retest` or `/test this` as a comment. That's how we can re-execute the same pipeline in case a failure is not due to a "real" problem but caused by flaky tests or some temporary issue.



At the time of this writing (May 2019) `/retest` and `/test this` commands do not yet work. But that does not mean that it doesn't work by the time you're reading this, so please try it out.



**Figure 13-1: A pull request with failed tests**

Before we move into the next subject, we'll remove the step with the silly test that always fails, and leave the repository in a good state.

```

1 cat jenkins-x.yml \
2   | sed '$ d' \
3   | tee jenkins-x.yml
4
5 git add .
6
7 git commit \
8   --message "Removed the silly test"
9
10 git push

```

The first command removed the last line from `jenkins-x.yml`, and the rest is the “standard” push to GitHub.

Now that our repository is back into the “working” state, we should explore the last available mode.

We used the `pre` mode to inject steps before those inherited from a build pack. Similarly, we saw that we can inject them after using the `post` mode. If we'd like to replace all the steps from a build pack lifecycle, we could

select the `replace` mode. There's probably no need to go through an exercise that would show that in action since the process is the same as for any other mode. The only difference is in what is added to `jenkins-x.yml`.

Before you start replacing lifecycles, be aware that you'd need to redo them completely. If, for example, you replace the steps in the `build` lifecycle, you'd need to make sure that you are implementing all the steps required to build your application. We rarely do that since Jenkins X build packs come with sane defaults and those are seldom removed. Nevertheless, there are cases when we do NOT want what Jenkins X offers and we might wish to reimplement a complete lifecycle by specifying steps through the `replace` mode.

Now that we added unit and functional tests, we should probably add some kind of integration tests as well.

## Extending Environment Pipelines

We could add integration tests in pipelines of our applications, but that's probably not the right place. The idea behind integration tests is to validate whether the system is integrated (hence the name). So, pipelines used to deploy to environments are probably better candidates for such tests. We already did a similar change with the static Jenkins X by modifying `Jenkinsfile` in staging and production repositories. Let's see how we can accomplish a similar effect through the new format introduced in serverless Jenkins X.



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

```
1 cd ..
2
3 GH_USER=[...]
4
5 git clone \
6   https://github.com/$GH_USER/environment-jx-rocks-staging.git
7
8 cd environment-jx-rocks-staging
```

We cloned the `environment-jx-rocks-staging` repository that contains the always-up-to-date definition of our staging environment. Let's take a

look at the `jenkins-x.yml` file that controls the processes executed whenever a change is pushed to that repo.

```
1 cat jenkins-x.yml
```

The output is as follows.

```
1 env:
2 - name: DEPLOY_NAMESPACE
3   value: jx-staging
4 pipelineConfig:
5   agent: {}
6   env:
7     - name: DEPLOY_NAMESPACE
8       value: jx-staging
9   pipelines: {}
```

This pipeline might be a bit confusing since there is no equivalent to `buildPack: go` we saw before. On the one hand, the pipeline is too short to be a full representation of the processes that result in deployments to the staging environment. On the other, there is no indication that this pipeline extends a pipeline from a buildpack. A pipeline is indeed inherited from a buildpack, but that is hidden by Jenkins X “magic” that, in my opinion, is not intuitive.

When that pipeline is executed, it will run whatever is defined in the build pack environment. Let’s take a look at the details.

```
1 curl https://raw.githubusercontent.com/jenkins-x-buildpacks/jenkins-x-kubernetes/master/packs/environment/pipeline.yaml
```

```
1 extends:
2   import: classic
3   file: pipeline.yaml
4 agent:
5   label: jenkins-go
6   container: gcr.io/jenkinsxio/builder-go
7 pipelines:
8   release:
9     build:
10       steps:
11         - dir: env
12         steps:
13           - sh: jx step helm apply
14             name: helm-apply
15
16   pullRequest:
17     build:
18       steps:
19         - dir: env
20         steps:
21           - sh: jx step helm build
22             name: helm-build
```

That should be familiar. It is functionally the same as environment pipelines we explored before when we used Jenkinsfile. Just as with the `go` buildpack, it extends a common `pipeline.yaml` located in the root of the `jenkins-x-classic` repository. The “classic” pipeline is performing common operations like checking out the code during the `setup` lifecycle, as well as some cleanup at the end of pipeline runs. If you’re interested in details, please visit the [jenkins-x-buildpacks/jenkins-x-classic](#) repository and open `packs/pipeline.yaml`.

If we go back to the `curl` output, we can see that there are only two steps. The first is running in the `build` lifecycle of the `release` pipeline. It applies the chart to the environment specified in the variable `DEPLOY_NAMESPACE`. The second step `jx step helm build` that is actually used to lint the chart and confirm that it is syntactically correct. That step is also executed during the `build` lifecycle but inside the `pullRequest` pipeline.

If our mission is to add integration tests, they should probably run after the application is deployed to an environment. That means that we should add a step to the `release` pipeline and that it must run after the current build step that executes `jx step helm apply`. We could add a new step as a `post` mode of the `build` lifecycle, or we can use any other lifecycle executed after `build`. In any case, the only important thing is that our integration tests run after the deployment performed during the `build` lifecycle. To make things more interesting, we’ll choose the `postbuild` lifecycle.

Please execute the command that follows.

```
1 jx create step \
2   --pipeline release \
3   --lifecycle postbuild \
4   --mode post \
5   --sh 'echo "Running integ tests!!!"'
```

As you can see, we won’t run “real” tests, but simulate them through a simple `echo`. Our goal is to explore serverless Jenkins X pipelines and not to dive into testing, so I believe that a simple message like that one should be enough.

Now that we added a new step we can take a look at what we got.

```
1 cat jenkins-x.yml
```

The output is as follows.

```
1 env:
2   - name: DEPLOY_NAMESPACE
3     value: jx-staging
4 pipelineConfig:
5   env:
6     - name: DEPLOY_NAMESPACE
7       value: jx-staging
8   pipelines:
9     release:
10       postBuild:
11         steps:
12           - command: echo "Running integ tests!!!"
```

As you can see, the `pipelines` section was expanded to include our new step following the same pattern as the one we saw when we extended the `go-demo-6` pipeline.

All that's left is to push the change before we confirm that everything works as expected.

```
1 git add .
2
3 git commit \
4   --message "Added integ tests"
5
6 git push
```

Just as before, we need to wait for a few moments until the new pipeline run starts, before we can retrieve the logs.

```
1 jx get build logs \
2   --filter environment=jx-rocks-staging \
3   --branch master
```

You should be presented with a choice which run to select. If the new one is present (e.g., #3), please select it. Otherwise, wait for a few moments more and repeat the `jx get build logs` command.

The last line of the output should display the message `Running integ tests!!!`, thus confirming that the change to the staging environment pipeline works as expected.

That's it. We created a PR that run unit and functional tests, and now we also have (a simulation of) integration tests that will be executed every time anything is deployed to the staging environment. If that would be “real” application with “real” tests, our next action should be to approve the pull request and let the system to the rest.

```
1 open "$PR_ADDR"
```

Feel free to go down the “correct” route of adding a colleague to the OWNERS file in the *master* branch and to the collaborators list. After you’re done, let the colleague write a comment with a slash command /approve or /lgtm. We already did those things in the previous chapter so you should know the drill. Or, you can be lazy (as I am), and just skip all that and click the *Merge pull request* button. Since the purpose of this chapter is not to explore ChatOps, you’ll be forgiven for taking a shortcut.

When you’re done merging the PR to the master branch, please click the *Delete branch* button in GitHub’s pull request screen. There’s no need to keep it any longer.

## What Now?

Now you need to decide whether to continue using the cluster or to destroy it. If you choose to destroy it or to uninstall Jenkins X, you’ll find the instructions at the bottom of the Gist you chose at the beginning of this chapter.

If you destroyed the cluster or you uninstalled Jenkins X, please remove the repositories and the local files we created. You can use the commands that follow for that.



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

```
1 cd ..
2
3 GH_USER=[...]
4
5 hub delete -y \
6   $GH_USER/environment-jx-rocks-staging
7
8 hub delete -y \
9   $GH_USER/environment-jx-rocks-production
10
11 rm -rf ~/.jx/environments/$GH_USER/environment-jx-rocks-*
12
13 rm -rf environment-jx-rocks-staging
```

# Extending Jenkins X Pipelines



The examples in this chapter work only with serverless Jenkins X. Nevertheless, pipelines defined in build packs use (almost) the same format as those used by serverless Jenkins X. When we create a new quickstart project or import an existing one into static Jenkins X, build pack pipelines are converted into Jenkinsfile. Therefore, even if you are not using serverless Jenkins X, advanced knowledge of writing YAML-based pipelines will help you when you choose to modify build pack pipelines.

So far, we relied mostly on pipelines created for us through build packs. No matter how much effort the community puts into creating build packs, it is almost sure that they will not fulfill all our needs. Every organization has something “special” and that inevitably leads to discrepancies between generic and tailor-made pipelines. So far, we extended our pipelines without knowing much about the syntax. We did not yet explore the benefits additional instructions might provide.

You can think of the subject of this chapter as advanced pipelines, but that would be an overstatement. No matter whether you’re using static or serverless pipelines, they are always simple. Or, to be more precise, they should be simple since their goal is not to define complex logic but rather to orchestrate automation defined somewhere else (e.g., scripts). That does not mean that there are no complex pipelines, but rather that those cases often reflect misunderstanding and the desire to solve problems in wrong places.



Pipelines are orchestrators of automation and should not contain complex logic.

Now, let’s define some objectives.

## What Are We Trying To Do?

It would be silly to explore Jenkins X pipeline syntax in more depth using random and irrelevant examples. Instead, we'll define some real and tangible goals. It does not matter whether they fit your specific needs since the objective is for them to guide us in our effort to learn by producing concrete outcomes.

Our next mission is to add code coverage reports, to ensure that functional tests are executed only after a release rolls out and that we build a version of our application binary for each of the popular operating systems (e.g., Windows, macOS, and Linux). Now, you might think that those goals are useful, or you might feel that they are a waste of time given your context. But, our real objective is not to accomplish those goals. Instead, we are using them as an excuse to learn some additional constructs that might come in handy. They will force us to learn a few new things.

- We'll add names to pipeline steps
- We'll learn how to define multi-line commands
- We'll start using additional environment variables
- We'll define custom agents
- We'll learn how to override pipelines, stages, and steps defined in build packs
- We'll learn how to implement loops

Those are only a fraction of what we could use. But, we need to start somewhere, and we have a set of improvements to our application that we'll be able to implement using the aforementioned concepts. Later on, it'll be up to you to expand your knowledge of pipeline constructs by exploring the other definitions we can use.

You might find those improvements useful as they are, or you might think of them as things you do not need. Both options are OK since the goal is not to show you how to add specific steps like code coverage, but rather to showcase some of the pipeline constructs that we might use in the context of our projects. All in all, focus on the value brought by additional pipeline instructions and not on examples I'll use to demonstrate how those constructs work.

As always, we need a cluster with Jenkins X so that we can experiment with some new concepts and hopefully improve our Jenkins X knowledge.

# Creating A Kubernetes Cluster With Jenkins X And Importing The Application

You can skip this section if you kept the cluster from the previous chapter and it contains **serverless Jenkins X**. Otherwise, we'll need to create a new Jenkins X cluster.



All the commands from this chapter are available in the [15-advanced-pipelines.sh](#) Gist.

For your convenience, the Gists that will create a new serverless Jenkins X cluster or install it inside an existing one are as follows.

- Create a new serverless **GKE** cluster: [gke-jx-serverless.sh](#)
- Create a new serverless **EKS** cluster: [eks-jx-serverless.sh](#)
- Create a new serverless **AKS** cluster: [aks-jx-serverless.sh](#)
- Use an **existing** serverless cluster: [install-serverless.sh](#)



The commands that follow will reset your `go-demo-6` master with the contents of the `extension-model-cd` branch that contains all the changes we did so far. Please execute them only if you are unsure whether you did all the exercises correctly.

```
1 cd go-demo-6
2
3 git pull
4
5 git checkout extension-tekton
6
7 git merge -s ours master --no-edit
8
9 git checkout master
10
11 git merge extension-tekton
12
13 git push
14
15 cd ..
```

If you ever restored a branch at the beginning of a chapter, the chances are that there is a reference to my user (`vfarcic`). We'll change that to Google project since that's what Knative will expect to be the location of the container images.



Please execute the commands that follow only if you are using **GKE** and if you ever restored a branch at the beginning of a chapter (like in the snippet above).

```
1 cd go-demo-6
2
3 cat charts/go-demo-6/Makefile \
4     | sed -e \
5     "s@vfarcic@$PROJECT@g" \
6     | tee charts/go-demo-6/Makefile
7
8 cat charts/preview/Makefile \
9     | sed -e \
10    "s@vfarcic@$PROJECT@g" \
11    | tee charts/preview/Makefile
12
13 cat skaffold.yaml \
14     | sed -e \
15     "s@vfarcic@$PROJECT@g" \
16     | tee skaffold.yaml
17
18 cd ..
```



If you destroyed the cluster at the end of the previous chapter, you'll need to import the *go-demo-6* application again. Please execute the commands that follow only if you created a new cluster specifically for the exercises from this chapter.

```
1 cd go-demo-6
2
3 jx import --pack go --batch-mode
4
5 cd ..
```

## Naming Steps And Using Multi-Line Commands

Let's take a quick look at the pipeline we have so far.

```
1 cd go-demo-6
2
3 cat jenkins-x.yml

1 buildPack: go
2 pipelineConfig:
3   pipelines:
4     pullRequest:
5       build:
6         preSteps:
7           - command: make unittest
8       promote:
9         steps:
10          - command: ADDRESS=`jx get preview --current 2>&1` make functest
```

We're extending the `go` pipeline defined as a build pack by adding two steps. We're executing unit tests (`make unittest`) before the build steps, and we added functional tests as a step after those pre-defined for the `promote` lifecycle. Both of those steps have issues we might want to fix.

So far, I tried my best to hide a big problem with the execution of functional tests in our pipelines. They are executed after promotion, but there is no guarantee that our application is fully operational before we run the tests. If you create a pull request right now, without modifying the pipeline, you are likely going to experience failure. A pipeline run triggered by the creation of a pull request will fail because the functional tests are executed not when the application in a preview environment is fully up-and-running but after the “deploy” instruction is sent to Kubernetes. As you probably already know, when we execute `kubectl apply`, Kube API responds with the acknowledgment that it received the instruction, not with the confirmation that the actual state converged to the desired one.

There are quite a few ways to ensure that an application is rolled out before we run tests against it. We're using Helm so you might be thinking that `--wait` should be enough. Usually, that would be the correct assumption if we use tiller (Helm server) in the cluster. But, Jenkins X does not use tiller due to security and quite a few other issues. To be more precise, it does not use tiller by default. You'll need to specify `--no-tiller false` argument when installing Jenkins X. If you followed the instructions (Gists) as they are, your cluster does not have it, and Jenkins X uses Helm only to convert charts (templates) into “standard” Kubernetes YAML files. In that case, the simplified version of the deployment process consists of executing `helm template` command followed with `kubectl apply`.

All in all, we do not have tiller (Helm server) unless you customized the installation so we cannot use `--wait` to tell the deployment (promotion) process to exit only after the application rolls out. Instead, we'll go back to basics and inject a step that will execute `kubectl rollout status`. It will serve two purposes. First, it will make the pipeline wait until the application rolls out so that we can run functional tests without the fear that it will fail if the app is not yet fully up-and-running. The second benefit is that the command will fail if rollout times out. If a command

fails, pipeline fails as well so we'll receive a notification that the new release could not roll out.

However, the problem is that we cannot merely execute `kubectl rollout status`. We need to know the namespace where our application is deployed, and each preview is deployed in a separate and unique one. Fortunately, namespaces are created using a known pattern, so we will not have a problem figuring it out. And now we're getting to a real issue. We need to execute three commands, the first to figure out the namespace of a preview environment, the second to wait for a few seconds to ensure that the deployment was indeed created by the promotion process, and the third with `kubectl rollout status`. We cannot put them into separate commands because each is a different session in a separate container. If we store the namespace in an environment variable in one step (`command`), that variable would not be available in the next. We could solve that by converting those three commands into one, but that would result in a very long single line instruction that would be very hard to read. We want readable code, don't we?

All that brings us to a new thing we'll learn about pipelines. We'll try to specify a multi-line command.

Before we move on and implement what we just discussed, there is one more problem we'll try to solve. The few custom steps we added to the pipeline consisted only of `command` instructions. They are nameless. If you paid closer attention, you probably noticed that Jenkins X auto-generated meaningless names for the steps we added. It does not have a "crystal ball" to figure out how we'd like to call the step with the `make unittest` command. So, our second improvement will be to add names to our custom steps. Those coming from buildpacks are already named, so we need to worry only for those we add directly to our pipelines.

All in all, we'll add a step with a multi-line command that will make the pipeline wait until the application rolls out, and we'll make sure that all our steps have a name.

Here we go.

We'll create a new branch and replace the content of `jenkins-x.yml` with the command that follows.



You'll see `# This is new` and `# This was modified` comments so that it's easier to figure out which parts of the pipeline are new or modified, and which are left unchanged.



Some of the lines might be too big, and the process of converting the manuscript into book formats might break them into multiple lines. Please use the Gists listed at the beginning of the chapter as a way to avoid potential problems.

```
1 git checkout -b better-pipeline
2
3 echo "buildPack: go
4 pipelineConfig:
5   pipelines:
6     pullRequest:
7       build:
8         preSteps:
9           # This was modified
10          - name: unit-tests
11            command: make unittest
12       promote:
13         steps:
14           # This is new
15           - name: rollout
16             command: |
17               NS=\`echo jx-\$REPO_OWNER-go-demo-6-\$BRANCH_NAME | tr '[[:upper:]]' '[
18 wer:]'\`'
19             sleep 15
20             kubectl -n \$NS rollout status deployment preview-preview --timeout 3
21           # This was modified
22           - name: functional-tests
23             command: ADDRESS=\`jx get preview --current 2>&1\` make functest
24 " | tee jenkins-x.yml
```

What did we change? We added the `rollout` step that contains a multi-line command that defines the namespace where the preview is deployed, sleeps for a while, and waits until the `deployment` is rolled out. All we had to do is specify pipe (`|`) and indent all the lines with the commands.

Since the branch is part of the namespace and it is in upper case (e.g., `PR-27`), the namespace is converted to lower case letters to comply with the standard. The second line of the `rollout` command sleeps for fifteen seconds. The reason for that wait is to ensure that the promotion build initiated by changing the repositories associated with automatic promotion to environments has started. Finally, the third line executes `kubectl rollout status`, thus forcing the pipeline to wait until the app is fully up

and running before executing functional tests. Additionally, if the rollout fails, the pipeline will fail as well.

In addition to the new step, we also added a `name` to all the steps we created so far. Now we have `unit-tests`, `rollout`, and `functional-tests`. Please note that you should not use space and “special” characters in names. Even though it is not a requirement, we prefer to have the names using lower case letters and words separated with dashes (-). We’ll see, later on, what Jenkins X does with those names.

Before we proceed and push the updates to the new branch, we should validate whether our changes to the pipeline comply with the schema.

```
1 jx step syntax validate pipeline
```

Assuming that you did not make a typo, the output should claim that the format of the pipeline was successfully validated.

Now we can push the changes to GitHub.

```
1 git add .
2
3 git commit -m "rollout status"
4
5 git push --set-upstream origin \
6   better-pipeline
```

Since all the changes we did so far were related to the `pullRequest` pipeline, so we need to create a PR if we’re going to test that everything works as expected.

```
1 jx create pullrequest \
2   --title "Better pipeline" \
3   --body "What I can say?" \
4   --batch-mode
```

Next, we’ll explore the activities related to this pull request. We’ll do that by limiting the activities for the branch that corresponds with the PR.



Please replace [...] with `PR-[PR_ID]` (e.g., `PR-72`). You can extract the ID from the last segment of the pull request address.

```
1 BRANCH=[...] # e.g., PR-72
```

Now we can, finally, take a look at the activities produced with the newly created pull request.

```
1 jx get activities \
2   --filter go-demo-6/$BRANCH \
3   --watch
```

The output is as follows.

```
1 ...
2 vfarcic/go-demo-6/PR-103 #1      1m36s 1m28s Succeeded
3   from build pack                 1m36s 1m28s Succeeded
4     Credential Initializer Qtb2s  1m36s  0s Succeeded
5     Working Dir Initializer Tchx7 1m35s  0s Succeeded
6     Place Tools                   1m34s  0s Succeeded
7     Git Source Vfarcic Go Demo ... 1m33s  1s Succeeded https://github.com/vfarc
8 go-demo-6
9   Git Merge                      1m32s  2s Succeeded
10  Build Unit Tests               1m32s  25s Succeeded
11  Build Make Linux               1m32s  27s Succeeded
12  Build Container Build         1m31s  30s Succeeded
13  Postbuild Post Build          1m31s  31s Succeeded
14  Promote Make Preview          1m31s  51s Succeeded
15  Promote Jx Preview            1m30s  1m17s Succeeded
16  Promote Rollout               1m30s  1m18s Succeeded
17  Promote Functional Tests     1m30s  1m22s Succeeded
18  Preview                       18s      https://github.com/vfarcic/go-de
19 /pull/103
20  Preview Application           18s      http://go-demo-6.jx-vfarcic-go-de
21 6-pr-103.35.196.24.179.nip.io
```

You'll notice that now we have a new step `Promote Rollout`. It is a combination of the name of the stage (`promote`) and the name of the step we defined earlier. Similarly, you'll notice that our unit and functional tests are now adequately named as well.

Feel free to stop watching the activities by pressing `ctrl+c`.

To be on the safe side, we'll take a quick look at the logs to confirm that the `rollout status` command is indeed executed.

```
1 jx get build logs --current
```

Even though we retrieved build logs quite a few times before, this time we used a new argument `--current`. With it, we do not need to specify the repository. Instead, `jx` assumed that the current folder is the repository name.

The output, limited to the relevant section, is as follows.

```
1 ...
2 Showing logs for build vfarcic-go-demo-6-pr-159-server-1 stage from-build-pack and
```

```
3 container step-promote-rollout
4 deployment "preview-preview" successfully rolled out
5 ...
```

We can see that the pipeline was waiting for deployment "preview-preview" rollout to finish and that the pipeline continued executing only after all three replicas of the application were rolled out. We can also see that the functional tests were executed only after the rollout, thus removing potential failure that could be caused by running tests before the application is fully up-and-running or, even worse, running them against the older release if the new one is still not rolled out.

Now that we saw how to define multi-line commands as well as how to name our steps, we'll explore how to work with environment variables and agents.

## Working With Environment Variables And Agents

Let's say that we want to add code coverage to our pipeline. We could do that through a myriad of tools. However, since the goal is not to teach you how to set up code coverage and how to explore which tool is better, we'll skip the selection process and use [Codecov](#) service. Just keep in mind that I'm not saying that it is better than the others nor that you must use a service for that, but rather that I needed an example to demonstrate a few new pipeline instructions. Codecov seems like the right candidate.

What do we need to do to integrate our pipeline with the Codecov service? If we check their instruction for Go applications, we'll see that we should output code coverage to a text file. Since I'm trying to make the examples as agnostic to programming languages as possible, we'll skip changing Makefile that contains testing targets assuming that you'll read the Codecov instructions later on if you choose to use it. So, instead of telling you to apply specific changes to Makefile, we'll download a Gist I prepared.

```
1 curl -o Makefile \
2     https://gist.githubusercontent.com/vfarcic/313bedd36e863249cb01af1f459139c7/raw
```

Now that we put Go internals out of the way, there are a few other things we need to do. We need to run [a script](#) provided by Codecov. That script expects a token that will authenticate us. So, we need three things. We need a container image with the script, an environment variable with the

token, and a pipeline step that will execute the script that will send the code coverage results toCodecov.

Let's start by retrieving a Codecov token for our *go-demo-6* repository.

```
1 open "https://codecov.io/"
```

I will skip giving you instructions on how to add your *go-demo-6* fork into Codecov. I'm sure that you will be able to sign up and follow the instructions provided on the site. Optionally, you can install their GitHub App (the message will appear at the top). What matter the most is the token you'll receive once you add the fork of the *go-demo-6* repository to Codecov.



Please replace [...] with the Codecov token for the *go-demo-6* repository.

```
1 CODECOV_TOKEN=[...]
```

There are a few ways we can provide the info Codecov needs to calculate code coverage. We'll use a Shell script they provide. To make things simple, I already created a container image that contains the script. It is a straightforward one, and you can explore Dockerfile used to create the image from the [vfarcic/codecov](#) repository.

```
1 open "https://github.com/vfarcic/codecov"
```

Open Dockerfile, and you'll see that the definition is as follows.

```
1 FROM alpine:3.9
2
3 RUN apk update && apk add bash curl git
4 RUN curl -o /usr/local/bin/codecov.sh https://codecov.io/bash
5 RUN chmod +x /usr/local/bin/codecov.sh
```

I already created a public image `vfarcic/codecov` based on that definition, so there is no action on your part. We can start using it right away.

So, what do we need to integrate Codecov with our pipeline? We need to define the environment variable `CODECOV_TOKEN` required by the `codecov.sh` script. We'll also need to add a new step that will execute that

script. We already know how to add steps, but this time there is a twist. We need to make sure that the new step is executed inside a container created from `vfarcic/codecov` image converted into a pipeline agent.

All in all, we need to figure out how to define environment variables as well as to define an agent based on a custom container image.

Please execute the command that follows to create the full pipeline that contains the necessary changes.

```
1 echo "buildPack: go
2 pipelineConfig:
3   # This is new
4   env:
5     - name: CODECOV_TOKEN
6       value: \"${CODECOV_TOKEN}\"
7   pipelines:
8     pullRequest:
9       build:
10         preSteps:
11           - name: unit-tests
12             command: make unittest
13           # This is new
14           - name: code-coverage
15             command: codecov.sh
16             agent:
17               image: vfarcic/codecov
18   promote:
19     steps:
20       - name: rollout
21         command: |
22           NS=\`echo jx-\$REPO_OWNER-go-demo-6-\$BRANCH_NAME | tr '[[:upper:]]' '[
23 wer:]'\`'
24         sleep 15
25         kubectl -n \$NS rollout status deployment preview-preview --timeout 3
26       - name: functional-tests
27         command: ADDRESS=\`jx get preview --current 2>&1\` make functest
28 " | tee jenkins-x.yml
```

Near the top of the pipeline is the `env` section that, in this case, defines a single variable `CODECOV_TOKEN`. We could have moved the `env` definition inside a pipeline, stage, or a step to limit its scope. As it is now, it will be available in all the steps of the pipeline.

We added a new step `code-coverage` inside the `pullRequest` pipeline. What makes it “special” is the `agent` section with the `image` set to `vfarcic/codecov`. As a result, that step will be executed inside a container based on that image. Just as with `env`, we could have defined `agent` in `pipelineConfig`, and then all the steps in all the pipelines would run in containers based on that image. Or we could have defined it on the level of a single pipeline or a stage.

All in all, for the sake of diversity, we defined an environment variable available in all the steps, and an agent that will be used in a single step.

Before we proceed, we'll check whether the syntax of the updated pipeline is correct.

```
1 jx step syntax validate pipeline
```

Next, we'll push the changes to GitHub and watch the activity that will be initiated by it.

```
1 git add .
2
3 git commit -m "Code coverage"
4
5 git push
6
7 jx get activities \
8   --filter go-demo-6/$BRANCH \
9   --watch
```

After the unit tests are executed, we should see the `Code Coverage` step of the `build` stage/lifecycle change the status to `succeeded`.

Feel free to cancel the watcher by pressing `ctrl+c`.

To be on the safe side, we'll take a quick look at the logs and confirm that `Codecov` script was indeed executed correctly.

```
1 jx get build logs --current
```

The output, limited to the relevant parts, is as follows (you might need to scroll through your output to find it).

```
1 ...
2
3
4   /__\|      |_|
5   ||  /__\ \ /__\|  |  | /__\ \ /__\ \ \ /__\
6   ||  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
7   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  v / 
8   \__\ \ \ /  \ \ , \ \ \ \ \ \ \ \ \ \ \ / \ \ \ \ \ /
9                                     Bash-8a28df4
10 ...
11 ==> Python coverageepy not found
12 ==> Searching for coverage reports in:
13     +
14     -> Found 1 reports
15 ==> Detecting git/mercurial file structure
16 ==> Reading reports
17     + ./coverage.txt bytes=1289
18 ==> Appending adjustments
19     http://docs.codecov.io/docs/fixing-reports
```

```

20      + Found adjustments
21 ==> Gzipping contents
22 ==> Uploading reports
23     url: https://codecov.io
24     query: branch=master&commit=eb67f2a869f16ce1a02d4903f6eec0af124300dc&build=&k
25 d_url=&name=&tag=&slug=vfarcic%2Fgo-demo-6&service=&flags=&pr=&job=
26     -> PingingCodecov
27 https://codecov.io/upload/v4?package=bash-8a28df4&token=4384f439-9da1-4be3-af60-e
28 aed67bc8&branch=master&commit=eb67f2a869f16ce1a02d4903f6eec0af124300dc&build=&bui
29 url=&name=&tag=&slug=vfarcic%2Fgo-demo-6&service=&flags=&pr=&job=
30     -> Uploading
31     -> View reports at https://codecov.io/github/vfarcic/go-demo-6/commit/eb67f2a
32 f16ce1a02d4903f6eec0af124300dc
33 ...

```

As you can see, the coverage report was uploaded to Codecov for evaluation, and we got a link where we can see the result. Feel free to visit it. We won't be using it in the exercises since there is a better way to see the results. I'll explain it soon. For now, there is an important issue we need to fix.

We added the Codecov token directly to the pipeline. As you can imagine, that is very insecure. There must be a way to provide the token without storing it in Git. Fortunately, Jenkins X pipelines have a solution. We can define an environment variable that will get the value from a Kubernetes secret. So, our next step is to create the secret.

```

1 kubectl create secret \
2   generic codecov \
3   --from-literal=token=$CODECOV_TOKEN

```

Now we can update the pipeline. Please execute the command that follows.

```

1 echo "buildPack: go
2 pipelineConfig:
3   env:
4     # This was modified
5     - name: CODECOV_TOKEN
6       valueFrom:
7         secretKeyRef:
8           key: token
9           name: codecov
10    pipelines:
11      pullRequest:
12        build:
13          preSteps:
14            - name: unit-tests
15              command: make unittest
16            - name: code-coverage
17              command: codecov.sh
18            agent:
19              image: vfarcic/codecov
20      promote:
21        steps:
22          - name: rollout

```

```

23      command: |
24      NS=\`echo jx-\$REPO_OWNER-go-demo-6-\$BRANCH_NAME | tr '[[:upper:]]' '['
25 wer:]`\
26      sleep 15
27      kubectl -n \$NS rollout status deployment preview-preview --timeout 3
28      - name: functional-tests
29      command: ADDRESS=\`jx get preview --current 2>&1\` make functest
30 " | tee jenkins-x.yml

```

This time, instead of creating an `env` with a value, we used `valuefrom` with reference to the Kubernetes secret `codecov` and the key `token`.

Let's see whether our updated pipeline works correctly.

```

1 jx step syntax validate pipeline
2
3 git add .
4
5 git commit -m "Code coverage secret"
6
7 git push
8
9 jx get activities \
10   --filter go-demo-6/\$BRANCH \
11   --watch

```

We validated the pipeline syntax, pushed the change to GitHub, and started watching the activities related to the pull request. The output should be the same as before since we did not change any of the steps. What matters is that all the steps of the newly executed activity should be successful.

Please stop watching the activity by pressing `ctrl+c`.

Now that we are securely calculating code coverage, we can take a look at the pull request. If the integration was successful, we should see Codecov entries.

Please click on the `Preview` link from the last activity to open the pull request in your favorite browser.

You should see a comment in the pull request similar to the screenshot that follows.

@@	Coverage	Diff	@@
##	master	#103	+/- ##
=====			
Coverage	?	75.55%	=====
=====			
Files	?	1	
Lines	?	90	
Branches	?	0	
=====			
Hits	?	68	
Misses	?	22	
Partials	?	0	

[Continue to review full report at Codecov.](#)

Legend - [Click here to learn more](#)  
 $\Delta$  = absolute <relative> (impact),  $\emptyset$  = not affected, ? = missing data  
Powered by [Codecov](#). Last update [bac38c2...eb67f2a](#). Read the [comment docs](#).

**Figure 15-1: Codecov report in GitHub**

Don't be alarmed by the warning (if you see it). Codecov could not compare pull request coverage with the one from the master branch because we did not yet merge anything to master since we started using Codecov. That'll be fixed by itself when we merge with PR.

Additionally, you should see Codecov activity in the “checks” section of the pull request.

Since delaying is not a good practice, let's merge the pull request right away. That way we'll give Codecov something to compare future pull request coverage. The next challenge will require that we work with the master branch anyway.

Please click *Merge pull request* followed by the *Confirm merge* button. Click the *Delete branch* button.

All that's left, before we move on, is to check out the master branch locally, to pull the latest version of the code from GitHub, and to delete the local copy of the `better-pipeline` branch.

```
1 git checkout master
2
3 git pull
4
5 git branch -d better-pipeline
```

Off we go to the next challenge.

## Overriding Pipelines, Stages, And Steps And Implementing Loops

Our pipeline is currently building a Linux binary of our application before adding it to a container image. But what if we'd like to distribute the application also as executables for different operating systems? We could provide that same binary, but that would work only for Linux users since that is the architecture it is currently built for. We might want to extend the reach to Windows and macOS users as well, and that would mean that we'd need to build two additional binaries. How could we do that?

Since our pipeline is already building a Linux executable through a step inherited from the build pack, we can add two additional steps that would build for the other two operating systems. But that approach would result in *go-demo-6* binary for Linux, and our new steps would, let's say, build *go-demo-6\_Windows* and *go-demo-6\_darwin*. That, however, would result in "strange" naming. In that context, it would make much more sense to have *go-demo-6\_linux* instead of *go-demo-6*. We could add yet another step that would rename it, but then we'd be adding unnecessary complexity to the pipeline that would make those reading it wonder what we're doing. We could build the Linux executable again, but that would result in duplication of the steps.

What might be a better solution is to remove the build step inherited from the build pack and add those that build the three binaries in its place. That would be a more optimum solution. One step removed, and three steps added. But those steps would be almost the same. The only difference would be an argument that defines each OS. We can do better than repeating almost the same step. Instead of having three steps, one for building a binary for each operating system, we'll create a loop that will

iterate through values that represent operating systems and execute a step that builds the correct binary.

All that might be too much to swallow at once, so we'll break it into two tasks. First, we'll try to figure out how to remove a step from the inherited build pack pipeline. If we're successful, we'll put the loop of steps in its place.

Let's get going.

We can use the `overrides` instruction to remove or replace any inherited element. We'll start with the simplest version of the instruction and improve it over time.

Please execute the command that follows to create a new version of `jenkins-x.yml`.

```
1 echo "buildPack: go
2 pipelineConfig:
3   env:
4     - name: CODECOV_TOKEN
5       valueFrom:
6         secretKeyRef:
7           key: token
8           name: codecov
9   pipelines:
10     pullRequest:
11       build:
12         preSteps:
13           - name: unit-tests
14             command: make unittest
15           - name: code-coverage
16             command: codecov.sh
17             agent:
18               image: vfarcic/codecov
19   promote:
20     steps:
21       - name: rollout
22         command: |
23           NS=\`echo jx-\$REPO_OWNER-go-demo-6-\$BRANCH_NAME | tr '[:upper:]' '[wer:]\``'
24           sleep 15
25           kubectl -n \$NS rollout status deployment preview-preview --timeout 3
26       - name: functional-tests
27         command: ADDRESS=\`jx get preview --current 2>&1\` make functest
28   # This is new
29   overrides:
30     - pipeline: release
31   " | tee jenkins-x.yml
```

All we did was to add two lines at the end of the pipeline. We specified that we want to override the `release` pipeline.

Just as with the previous examples, we'll validate the syntax, push the changes to GitHub, and observe the result by watching the activities.

```
1 jx step syntax validate pipeline
2
3 git add .
4
5 git commit -m "Multi-architecture"
6
7 git push
8
9 jx get activities \
10   --filter go-demo-6/master \
11   --watch
```

The output of the last command, limited to the relevant parts, is as follows.

```
1 ...
2 vfarcic/go-demo-6/master #3
3   meta pipeline
4     Credential Initializer Bsggw
5     Working Dir Initializer 5n6mx
6     Place Tools
7     Git Source Meta Vfarcic Go Demo 6 Master R ...
8 com/vfarcic/go-demo-6.git
9     Git Merge
10    Merge Pull Refs
11    Create Effective Pipeline
12    Create Tekton Crds
13    from build pack
14      Credential Initializer Fw774
15      Working Dir Initializer S7292
16      Place Tools
17      Git Source Vfarcic Go Demo 6 Master Releas ...
18 com/vfarcic/go-demo-6.git
19     Git Merge
20     Setup Jx Git Credentials
```

Judging from the output of the latest activity, the number of steps dropped drastically. That's the expected behavior since we told Jenkins X to override the release pipeline with "nothing". We did not specify replacement steps that should be executed instead of those inherited from the build pack. So, the only steps executed are those related to Git since they are universal and not tied to any specific pipeline.

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

In our case, overriding the whole `release` pipeline might be too much. We do not have a problem with all of the inherited steps, but only with the `build` stage inside the `release` pipeline. So, we'll override only that one.

Since we are about to modify the pipeline yet again, we might want to add the `rollout` command to the `release` pipeline as well. It'll notify us if a

release cannot be rolled out.

Off we go.

```
1 echo "buildPack: go
2 pipelineConfig:
3   env:
4     - name: CODECOV_TOKEN
5       valueFrom:
6         secretKeyRef:
7           key: token
8           name: codecov
9   pipelines:
10     pullRequest:
11       build:
12         preSteps:
13           - name: unit-tests
14             command: make unittest
15           - name: code-coverage
16             command: codecov.sh
17             agent:
18               image: vfarcic/codecov
19   promote:
20     steps:
21       - name: rollout
22         command: |
23           NS=\`echo jx-\$REPO_OWNER-go-demo-6-\$BRANCH_NAME | tr '[:upper:]' '[wer:]'\```
24         sleep 15
25         kubectl -n \$NS rollout status deployment preview-preview --timeout 3
26       - name: functional-tests
27         command: ADDRESS=\`jx get preview --current 2>&1\` make functest
28   overrides:
29     - pipeline: release
30       # This is new
31       stage: build
32       # This is new
33     release:
34       promote:
35       steps:
36         - name: rollout
37           command: |
38             sleep 30
39             kubectl -n jx-staging rollout status deployment jx-go-demo-6 --timeou
40           " | tee jenkins-x.yml
```

We added the `stage: build` instruction to the existing override of the `release` pipeline. We also added the `rollout` command as yet another step in the `promote` stage of the `release` pipeline.

You probably know what comes next. We'll validate the pipeline syntax, push the changes to GitHub, and observe the activities hoping that they will tell us whether the change was successful or not.

```
1 jx step syntax validate pipeline
2
3 git add .
4
5 git commit -m "Multi-architecture"
```

```

6
7 git push
8
9 jx get activities \
10   --filter go-demo-6/master \
11   --watch

```

The output, limited to the latest build, is as follows.

```

1 ...
2 vfarcic/go-demo-6/master #5           3m46s 2m45s Succeeded Version:
3 0.446
4   meta pipeline                         3m46s  21s Succeeded
5     Credential Initializer L6kh9        3m46s  0s Succeeded
6     Working Dir Initializer Khkf6       3m46s  0s Succeeded
7     Place Tools                         3m46s  1s Succeeded
8     Git Source Meta Vfarcic Go Demo 6 Master R ... 3m45s  5s Succeeded https://
9 hub.com/vfarcic/go-demo-6.git
10    Git Merge                           3m40s  1s Succeeded
11    Merge Pull Refs                    3m39s  0s Succeeded
12    Create Effective Pipeline          3m39s  4s Succeeded
13    Create Tekton Crds                3m35s  10s Succeeded
14   from build pack                   3m23s 2m22s Succeeded
15     Credential Initializer 5cw8t      3m23s  0s Succeeded
16     Working Dir Initializer D99p2     3m23s  1s Succeeded
17     Place Tools                      3m22s  1s Succeeded
18     Git Source Vfarcic Go Demo 6 Master Releas ... 3m21s  6s Succeeded https://
19 hub.com/vfarcic/go-demo-6.git
20    Git Merge                           3m15s  0s Succeeded
21    Setup Jx Git Credentials          3m15s  0s Succeeded
22    Promote Changelog                 3m15s  8s Succeeded
23    Promote Helm Release              3m7s   18s Succeeded
24    Promote Jx Promote                2m49s 1m32s Succeeded
25    Promote Rollout                  1m17s  16s Succeeded
26   Promote: staging                 2m43s 1m26s Succeeded
27     PullRequest                     2m43s 1m26s Succeeded  PullReq
28 t: ...
29     Update                          1m17s  0s Succeeded
30     Promoted                        1m17s  0s Succeeded  Applica
31 n ...

```

The first thing we can note is that the number of steps in the activity is closer to what we're used to. Now that we are not overriding the whole pipeline but only the `build` stage, almost all the steps inherited from the build pack are there. Only those related to the `build` stage are gone, simply because we limited the scope of the `overrides` instruction.

Another notable difference is that the `Promote Rollout` step took too long to execute until it eventually failed. That's also to be expected. We removed all the steps from the `build` stage, so our binary was not created, and the container image was not built. Jenkins X did execute `promote` steps that are deploying the new release, but Kubernetes is bound to fail to pull the new image.

That demonstrated the importance of executing `rollout`, no matter whether we run tests afterward. Without it, the pipeline would finish successfully since we are not running tests against the staging environment. Before we added the `rollout` step, the promotion was the last action executed as part of the pipeline.

Please stop watching the activities by pressing `ctrl+c`.

We are getting closer to our goal. We just need to figure out how to override a specific step with the new one that will build binaries for all operating systems. But, how are we going to override a particular step if we do not know which one it is? We could find all the steps of the pipeline by visiting the repositories that host build packs. But that would be tedious. We'd need to go to a few repositories, check the source code of the related pipelines, and combine the result with the one we're rewriting right now. There must be a better way to get an insight into the pipeline related to `go-demo-6`.

Before we move on and try to figure out how to retrieve the full definition of the pipeline, we'll revert the current version to the state before we started “playing” with `overrides`. You'll see the reason for such a revert soon.

```
1 echo "buildPack: go
2 pipelineConfig:
3   env:
4     - name: CODECOV_TOKEN
5       valueFrom:
6         secretKeyRef:
7           key: token
8           name: codecov
9   pipelines:
10    pullRequest:
11      build:
12        preSteps:
13          - name: unit-tests
14            command: make unittest
15          - name: code-coverage
16            command: codecov.sh
17            agent:
18              image: vfarcic/codecov
19    promote:
20      steps:
21        - name: rollout
22          command: |
23            NS=\`echo jx-\$REPO_OWNER-go-demo-6-\$BRANCH_NAME | tr '[:upper:]' '[wer:]'\``#
24          sleep 15
25          kubectl -n \$NS rollout status deployment preview-preview --timeout 3
27        - name: functional-tests
28          command: ADDRESS=\`jx get preview --current 2>&1\` make functest
29 # Removed overrides
```

```

30     release:
31         promote:
32             steps:
33                 - name: rollout
34                     command: |
35                         sleep 30
36                         kubectl -n jx-staging rollout status deployment jx-go-demo-6 --timeout
37 " | tee jenkins-x.yml

```

Now that we are back to where we were before we discovered `overrides`, we can learn about yet another command.

```
1 jx step syntax effective
```

The output is the “effective” version of our pipeline. You can think of it as a merge of our pipeline combined with those it extends (e.g., from build packs). It is the same final version of the YAML pipeline Jenkins X would use as a blueprint for creating Tekton resources.

The reason we’re outputting the effective pipeline lies in our need to find the name of the step currently used to build the Linux binary of the application. If we find its name, we will be able to override it.

The output, limited to the relevant parts, is as follows.

```

1 buildPack: go
2 pipelineConfig:
3 ...
4 pipelines:
5 ...
6   release:
7     pipeline:
8       ...
9       stages:
10      - agent:
11          image: go
12          name: from-build-pack
13          steps:
14            ...
15            - command: make build
16              dir: /workspace/source
17              image: go
18              name: build-make-build
19            ...

```

We know that the step we’re looking for is somewhere inside the `release` pipeline, so that should limit the scope. If we take a look at the steps inside, we can see that one of them executes the command `make build`. That’s the one we should remove or, to be more precise, `override`.

You’ll notice that the names of the steps are different in the effective version of the pipeline. For example, the `rollout` step we created earlier is

now called `promote-rollout`. In the effective version of the pipelines, the step names are always prefixed with the stage. As a result, when we see the activities retrieved from Tekton pipeline runs, we see the two (stage and step) combined.

There's one more explanation I promised to deliver. Why did we revert the pipeline to the version before we added overrides? If we didn't, we would not find the step we were looking for. The whole `build` stage from the `release` pipeline would be gone since we had it overridden to nothing.

Now, let's get back to our mission. We know that the step we want to override in the effective version of the pipeline is named `build-make-build`. Since we know that the names are prefixed with the stage, we can deduce that the stage is `build` and the name of the step is `make-build`.

Now that it's clear what to override, let's talk about loops.

We can tell Jenkins X to loop between values and execute a step or a set of steps in each iteration. An example syntax could be as follows.

```
1 - loop:
2   variable: COLOR
3   values:
4     - yellow
5     - red
6     - blue
7     - purple
8     - green
9   steps:
10    - command: echo "The color is $COLOR"
```

If we'd have that loop inside our pipeline, it would execute a single step five time, once for each of the `values` of the `loop`. What we put inside the `steps` section is up to us, and the only important thing to note is that `steps` in the `loop` use the same syntax as the `steps` anywhere else (e.g., in one of the stages).

Now, let's see whether we can combine `overrides` with `loop` to accomplish our goal of building a binary for each of the “big” three operating systems.

Please execute the command that follows to update `jenkins-x.yml` with the new version of the pipeline.

```
1 echo "buildPack: go
2 pipelineConfig:
```

```

3   env:
4     - name: CODECOV_TOKEN
5       valueFrom:
6         secretKeyRef:
7           key: token
8           name: codecov
9   pipelines:
10    pullRequest:
11      build:
12        preSteps:
13          - name: unit-tests
14            command: make unittest
15          - name: code-coverage
16            command: codecov.sh
17            agent:
18              image: vfarcic/codecov
19    promote:
20      steps:
21        - name: rollout
22          command: |
23            NS=\`echo jx-\$REPO_OWNER-go-demo-6-\$BRANCH_NAME | tr '[:upper:]' '[wer:]\```
24      sleep 30
25      kubectl -n \$NS rollout status deployment preview-preview --timeout 3
26    - name: functional-tests
27      command: ADDRESS=\`jx get preview --current 2>&1\` make functest
28  overrides:
29    - pipeline: release
30      # This is new
31      stage: build
32      name: make-build
33      steps:
34        - loop:
35          variable: GOOS
36          values:
37            - darwin
38            - linux
39            - windows
40          steps:
41            - name: build
42              command: CGO_ENABLED=0 GOOS=\${GOOS} GOARCH=amd64 go build -o bin/go-
43 o-6_\${GOOS} main.go
44 release:
45   promote:
46     steps:
47       - name: rollout
48         command: |
49         sleep 15
50         kubectl -n jx-staging rollout status deployment jx-go-demo-6 --timeou
51 " | tee jenkins-x.yml

```

This time we are overriding the step `make-build` in the `build` stage of the `release` pipeline. The “old” step will be replaced with a `loop` that iterates over the values that represent operating systems. Each iteration of the loop contains the `GOOS` variable with a different value and executes the `command` that uses it to customize how we build the binary. The end result should be `go-demo-6_` executable with the unique suffix that tells us where it is meant to be used (e.g., `linux`, `darwin`, or `windows`)s.



If you're new to Go, the compiler uses environment variable `GOOS` to determine the target operating system for a build.

Next, we'll validate the pipeline and confirm that we did not introduce a typo incompatible with the supported syntax.

```
1 jx step syntax validate pipeline
```

There's one more thing we should fix. In the past, our pipeline was building the `go-demo-6` binary, and now we changed that to `go-demo-6_linux`, `go-demo-6_darwin`, and `go-demo-6_windows`. Intuition would tell us that we might need to change the reference to the new binary in Dockerfile, so let's take a quick look at it.

```
1 cat Dockerfile
```

The output is as follows.

```
1 FROM scratch
2 EXPOSE 8080
3 ENTRYPOINT ["/go-demo-6"]
4 COPY ./bin/ /
```

The last line will copy all the files from the `bin/` directory. That would introduce at least two problems. First of all, there is no need to have all three binaries inside container images we're building. That would make them bigger for no good reason. The second issue with the way binaries are copied is the `ENTRYPOINT`. It expects `/go-demo-6`, instead of `go-demo-6_linux` that we are building now. Fortunately, the fix to both of the issues is straightforward. We can change the Dockerfile `COPY` instruction so that only `go-demo-6_linux` is copied and that it is renamed to `go-demo-6` during the process. That will help us avoid copying unnecessary files and will still fulfill the `ENTRYPOINT` requirement.

```
1 cat Dockerfile \
2   | sed -e \
3     's@/bin/ @/bin/go-demo-6_linux /go-demo-6@g' \
4   | tee Dockerfile
```

Now we're ready to push the change to GitHub and observe the new activity that will be triggered by that action.

```

1 git add .
2
3 git commit -m "Multi-architecture"
4
5 git push
6
7 jx get activities \
8   --filter go-demo-6/master \
9   --watch

```

The output, limited to the latest build, is as follows.

```

1 ...
2 vfarcic/go-demo-6/master #6           5m32s 5m18s Succeeded Version:
3 0.447
4   meta pipeline                         5m32s  24s Succeeded
5     Credential Initializer Pg5cf        5m32s  0s Succeeded
6     Working Dir Initializer Lzpdb       5m32s  2s Succeeded
7     Place Tools                         5m30s  1s Succeeded
8     Git Source Meta Vfarcic Go Demo 6 Master R ... 5m29s  4s Succeeded https://
9 hub.com/vfarcic/go-demo-6.git
10    Git Merge                           5m25s  1s Succeeded
11    Merge Pull Refs                    5m24s  0s Succeeded
12    Create Effective Pipeline          5m24s  4s Succeeded
13    Create Tekton Crds                5m20s  12s Succeeded
14   from build pack                   5m6s  4m52s Succeeded
15     Credential Initializer P5wrz      5m6s  0s Succeeded
16     Working Dir Initializer Frrq2     5m6s  0s Succeeded
17     Place Tools                      5m6s  1s Succeeded
18     Git Source Vfarcic Go Demo 6 Master Releas ... 5m5s  9s Succeeded https://
19 hub.com/vfarcic/go-demo-6.git
20    Git Merge                           4m56s  1s Succeeded
21    Setup Jx Git Credentials          4m55s  0s Succeeded
22    Build1                            4m55s  42s Succeeded
23    Build2                            4m13s  16s Succeeded
24    Build3                            3m57s  33s Succeeded
25    Build Container Build            3m24s  5s Succeeded
26    Build Post Build                 3m19s  0s Succeeded
27    Promote Changelog                3m19s  7s Succeeded
28    Promote Helm Release             3m12s  16s Succeeded
29    Promote Jx Promote               2m56s  1m31s Succeeded
30    Promote Rollout                  1m25s  1m11s Succeeded
31   Promote: staging                 2m50s  1m25s Succeeded
32   PullRequest                      2m50s  1m24s Succeeded PullReq
33 t: ...f157af83f254d90df5e9b0c4bb8ddb81e1016871
34   Update                            1m25s  0s Succeeded
35   Promoted                          1m25s  0s Succeeded Applica
36 n is at: ...

```

We can make a few observations. The Build Make Build step is now gone, so the override worked correctly. We have Build1, Build2, and Build3 in its place. Those are the three steps created as a result of having the loop with three iterations. Those are the steps that are building windows, linux, and darwin binaries. Finally, we can observe that the Promote Rollout step is now shown as succeeded, thus providing a clear indication that the new building process (steps) worked correctly. Otherwise, the new release could not roll out, and that step would fail.

Please stop watching the activities by pressing *ctrl+c*.

Before we move on, I must confess that I would not make the same implementation as the one we just explored. I'd rather change the `build` target in Makefile. That way, there would be no need for any change to the pipeline. The build pack step would continue building by executing that Makefile target so there would be no need to override anything, and there would certainly be no need for a loop. Now, before you start throwing stones at me, I must also state that `overrides` and `loop` can come in handy in some other scenarios. I had to come up with an example that would introduce you to `overrides` and `loop`, and that ended up being the need to cross-compile binaries, even if it could be accomplished in an easier and a better way. Remember, the “real” goal was to learn those constructs, and not how to cross-compile with Go.

## Pipelines Without Buildpacks

While the idea behind build packs is to cover a wide range of use cases, we might easily be in a situation when what we want to accomplish is fundamentally different from any of the build packs. In such cases, it probably does not make sense to have a pipeline based on a build pack. Instead, we can tell Jenkins X that our pipeline is not based on any build pack.

Please execute the command that follows to create a pipeline without a build pack.

```
1 echo "buildPack: none
2 pipelineConfig:
3   pipelines:
4     release:
5       pipeline:
6         agent:
7           image: go
8         stages:
9           - name: nothing
10          steps:
11            - name: silly
12              command: echo \"This is a silly pipeline\" \
13 | tee jenkins-x.yml
```

In this context, the only line that matters is the first one that instructs Jenkins X not to use any `buildPack` by setting the value to `none`. The rest of the pipeline contains a silly example with constructs that we already explored.

Now, let's push the change and confirm that none of the build packs is used by observing the activities.

```
1 git add .
2
3 git commit -m "Without buildpack"
4
5 git push
6
7 jx get activities \
8   --filter go-demo-6/master \
9   --watch
```

The output of the last command, limited to the newest activity, is as follows.

```
1 ...
2 vfarcic/go-demo-6/master #7          34s 28s Succeeded
3   meta pipeline                      34s 20s Succeeded
4     Credential Initializer Jnghb    34s 0s Succeeded
5     Working Dir Initializer H4bg2  34s 1s Succeeded
6     Place Tools                      33s 1s Succeeded
7     Git Source Meta Vfarcic Go Demo 6 Master R ... 32s 5s Succeeded https://gith
8 com/vfarcic/go-demo-6.git
9       Git Merge                      27s 1s Succeeded
10      Merge Pull Refs               26s 1s Succeeded
11      Create Effective Pipeline    25s 3s Succeeded
12      Create Tekton Crds          22s 8s Succeeded
13      nothing                      13s 7s Succeeded
14     Credential Initializer 26fg8  13s 0s Succeeded
15     Working Dir Initializer Fz7zz 13s 1s Succeeded
16     Place Tools                  12s 1s Succeeded
17     Git Source Vfarcic Go Demo 6 Master Releas ... 11s 4s Succeeded https://gith
18 com/vfarcic/go-demo-6.git
19       Git Merge                  7s 1s Succeeded
20       Silly                      6s 0s Succeeded
```

We can see that all the steps we normally get from a build pack are gone. We are left only with the generic Git-related steps and the `silly` one we defined in our pipeline.

Please stop watching the activities by pressing `ctrl+c`.

## Exploring The Syntax Schema

I bet that you wondered how I knew that there are instructions like `overrides` and `loop`? I could have consulted the documentation in [jenkins-x.io](https://jenkins-x.io), but it is not always up-to-date. I could have consulted the code. As a matter of fact, most of what I know about pipelines comes from reviewing the code of the project. I did that not only because I like reading code, but also because that was the only reliable way to find out all the instructions we can specify. Fortunately, things got a bit simpler since then, and the

community added the means to consult the full syntax schema with a single command.

```
1 jx step syntax schema
```

The output is too long to be presented in a book, so I'll choose a single construct (`loop`) and show only the output related to it.

```
1 {
2   "$schema": "http://json-schema.org/draft-04/schema#",
3   "$ref": "#/definitions/ProjectConfig",
4   "definitions": {
5     ...
6     "Loop": {
7       "properties": {
8         "steps": {
9           "items": {
10            "$ref": "#/definitions/Step"
11          },
12          "type": "array"
13        },
14        "values": {
15          "items": {
16            "type": "string"
17          },
18          "type": "array"
19        },
20        "variable": {
21          "type": "string"
22        }
23      },
24      "additionalProperties": false,
25      "type": "object"
26    },
27    ...
28    "Step": {
29      "properties": {
30        ...
31        "loop": {
32          "$schema": "http://json-schema.org/draft-04/schema#",
33          "$ref": "#/definitions/Loop"
34        },
35        ...
36      }
37    }
38  }
39}
```

We can see that `Loop` is one of the definitions and that it can contain `steps`, `values`, and `variables`. The array of steps includes a reference to the `Step` definition. If we scroll down to that definition, we can see that, among others, it can have a loop inside. All in all, a step can contain a loop, and a loop can include steps inside it.

We won't go into all the definitions we can use inside pipelines. That would be too much and could easily fill an entire book alone. We will probably explore a couple of other schema definitions, and it's up to you to go through those we'll skip. For now, I'll use this opportunity to introduce you to a few other potentially useful commands.

We can add `--buildpack` argument if we want to find out the schema that is used with build pack pipelines.

```
1 jx step syntax schema --buildpack
```

Application pipeline and build pack schemas are very similar, but there are still a few differences. Use one schema or the other depending on whether you're planning to modify pipeline of your application or of a build pack.

Since we are talking about build packs, it's worth noting that we can validate those already used by our cluster with the command that follows.

```
1 jx step syntax validate buildpacks
```

If all the build packs are valid, we'll see the `SUCCESS` status in the output of each.

## What Now?

If you are using static Jenkins X, you should consider relying on build packs as much as possible. Serverless Jenkins X is the future, and most of the effort (new features, bug fixes, etc.) will be focused around it. On the other hand, static Jenkins X is in "maintenance mode". That does not mean that you should use it. There are quite a few reasons why static Jenkins X might be a better option for you. We won't go into those reasons now. I am mentioning all this because you will move to serverless Jenkins X at some point and you do not want to spend your precious time rewriting your Jenkinsfiles into the `jenkins-x.yml` format. If most of your pipelines is in build packs, you can easily switch from static to serverless Jenkins X. All you'd have to do is re-import your project and let Jenkins X convert the pipeline into the correct format.



We explored only a fraction of the Jenkins X pipeline syntax. Please consult [Jenkins X Pipeline Syntax Reference](#) for more info.

Before we leave, we'll restore the master to the `extension-model-cd`. Our `jenkins-x.yml` became too big for future examples so we'll go back to the much simpler one we had at the beginning of this chapter. I will assume

that you understood the constructs we used and that you will extend that knowledge by exploring the pipeline schema. If we'd keep adding everything we learn to our *go-demo-6* pipeline, we'd soon need multiple pages only to list jenkins-x.yml content.

```
1 git checkout extension-tekton
2
3 git merge -s ours master --no-edit
4
5 git checkout master
6
7 git merge extension-tekton
8
9 git push
```

Now you need to decide whether to continue using the cluster or to destroy it. If you choose to destroy it or to uninstall Jenkins X, you'll find the instructions at the bottom of the Gist you chose at the beginning of this chapter.

If you destroyed the cluster or you uninstalled Jenkins X, please remove the repositories and the local files we created. You can use the commands that follow for that.



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

```
1 cd ..
2
3 GH_USER=[...]
4
5 hub delete -y \
6   $GH_USER/environment-jx-rocks-staging
7
8 hub delete -y \
9   $GH_USER/environment-jx-rocks-production
10
11 rm -rf ~/.jx/environments/$GH_USER/environment-jx-rocks-*
```

# Using Jenkins X To Define And Run Serverless Deployments



At the time of this writing (December 2019), the examples in this chapter work only in a GKE cluster. Feel free to monitor [the issue 4668](#) for more info.

We already saw how we could run the serverless flavor of Jenkins X. That helped with many things, with better resource utilization and scalability being only a few of the benefits. Can we do something similar with our applications? Can we scale them to zero when no one is using them? Can we scale them up when the number of concurrent requests increases? Can we make our applications serverless?

Let's start from the beginning and discuss serverless computing.

## What is Serverless Computing?

To understand serverless computing, we need to understand the challenges we are facing with more “traditional” types of deployments of our applications.

A long time ago, most of us were deploying our apps directly to servers. We had to decide the size (memory and CPU) of the nodes where our applications would run, we had to create those servers, and we had to maintain them. The situation improved with the emergence of cloud computing. We still had to do all those things, but now those tasks were much easier due to the simplicity of the APIs and services cloud vendors gave us. Suddenly, we had (a perception of) infinite resources, and all we had to do is run a command, and a few minutes later, the servers (VMs) we needed would materialize. Things become much easier and faster. However, that did not remove the tasks of creating and maintaining servers. Instead, that made them more straightforward. Concepts like immutability become mainstream. As a result, we got much-needed

reliability, we reduced drastically lean time, and we started to rip the benefits of elasticity.

Still, some important questions were left unanswered. Should we keep our servers running even when our applications are not serving any requests? If we shouldn't, how can we ensure that they are readily available when we do need them? Who should be responsible for the maintenance of those servers? Is it our infrastructure department, is it our cloud provider, or can we build a system that will do that for us without human intervention?

Things changed with the emergence of containers and schedulers. After a few years of uncertainty created by having too many options on the table, the situation stabilized around Kubernetes that became the de-facto standard. At roughly the same time, in parallel with the rise of the popularity of containers and schedulers, solutions for serverless computing concepts started to materialize. Those solutions were not related to each other or, to be more precise, they were not during the first few years. Kubernetes provided us with means to run microservices as well as more traditional types of applications, while serverless focused on running functions (often only a few lines of code).

The name serverless is misleading by giving the impression that they are no servers involved. They are certainly still there, but the concept and the solutions implementing them allow us (users) to ignore their existence. The major cloud providers (AWS, Microsoft Azure, and Google) all came up with solutions for serverless computing. Developers could focus on writing functions with a few additional lines of code specific to the serverless computing vendor we choose. Everything else required for running and scaling those functions become transparent.

But not everything is excellent in the serverless world. The number of use-cases that can be fulfilled with functions (as opposed to applications) is limited. Even when we do have enough use-cases to make serverless computing a worthwhile effort, a more significant concern is lurking just around the corner. We are likely going to be locked into a vendor, given that none of them implements any type of industry standard. No matter whether we choose AWS Lambda, Azure Functions, or Google Cloud Functions, the code we write will not be portable from one vendor to another. That does not mean that there are no serverless frameworks that are not tied to a specific cloud provider. There are, but we'd need to maintain them ourselves, be it on-prem or inside clusters running in a

public cloud, and that removes one of the most essential benefits of serverless concepts.

That's where Kubernetes comes into play.

## **Serverless Deployments In Kubernetes**

At this point, I must make an assumption that you, dear reader, might disagree with. Most of the companies will run at least some (if not all) of their applications in Kubernetes. It is becoming (or it already is) a standard API that will be used by (almost) everyone. Why is that assumption important? If I am right, then (almost) everyone will have a Kubernetes cluster. Everyone will spend time maintaining it, and everyone will have some level of in-house knowledge of how it works. If that assumption is correct, it stands to reason that Kubernetes would be the best choice for a platform to run serverless applications as well. As an added bonus, that would avoid vendor lock-in since Kubernetes can run (almost) anywhere.

Kubernetes-based serverless computing would provide quite a few other benefits. We could be free to write our applications in any language, instead of being limited by those supported by function-as-a-service solutions offered by cloud vendors. Also, we would not be limited to writing only functions. A microservice or even a monolith could run as a serverless application. We just need to find a solution to make that happen. After all, proprietary cloud-specific serverless solutions use containers (of sorts) as well, and the standard mechanism for running containers is Kubernetes.

There is an increasing number of Kubernetes platforms that allow us to run serverless applications. We won't go into all of those but fast-track the conversation by me stating that Knative is likely going to become the de-facto standard how to deploy serverless load to Kubernetes. Or, maybe, it already is the most widely accepted standard by the time you read this.

[Knative](#) is an open-source project that delivers components used to build and run serverless applications on Kubernetes. We can use it to scale-to-zero, to autoscale, for in-cluster builds, and as an eventing framework for applications on Kubernetes. The part of the project we're interested in right now is its ability to convert our applications into serverless deployments, and that means auto-scaling down until zero, and up to whatever an application needs. That should allow us both to save resources

(memory and CPU) when our applications are idle, as well as to scale them fast when traffic increases.

Now that we discussed what is serverless and that I made an outlandish statement that Kubernetes is the platform where your serverless applications should be running, let's talk about types of scenarios that are a good fit for serverless deployments.

## **Which Types Of Applications Should Run As Serverless?**

Initially, the idea was to have only functions running as serverless loads. Those would be single-purpose pieces of code that contain only a small number of lines of code. A typical example of a serverless application would be an image processing function that responds to a single request and can run for a limited period. Restrictions like the size of applications (functions) and their maximum duration are imposed by implementations of serverless computing in cloud providers. But, if we adopt Kubernetes as the platform to run serverless deployments, those restrictions might not be valid anymore. We can say that any application that can be packaged into a container image can run as a serverless deployment in Kubernetes. That, however, does not mean that any container is as good of a candidate as any other. The smaller the application or, to be more precise, the faster its boot-up time, the better the candidate for serverless deployments.

However, things are not as straight forward as they may seem. Not being a good candidate does not mean that one should not compete at all. Knative, like many other serverless frameworks, does allow us to fine-tune configurations. We can, for example, specify with Knative that there should never be less than one replica of an application. That would solve the problem of slow boot-up while still maintaining some of the benefits of serverless deployments. In such a case, there would always be at least one replica to handle requests, while we would benefit from having the elasticity of serverless providers.

The size and the boot-up time are not the only criteria we can use to decide whether an application should be serverless. We might want to consider traffic as well. If, for example, our app has high traffic and it receives requests throughout the whole day, we might never need to scale it down to zero replicas. Similarly, our application might not be designed in a way that every request is processed by a different replica. After all, most of the

apps can handle a vast number of requests by a single replica. In such cases, serverless computing implemented by cloud vendors and based on function-as-a-service might not be the right choice. But, as we already discussed, there are other serverless platforms, and those based on Kubernetes do not follow those rules. Since we can run any container as a serverless, any type of application can be deployed as such, and that means that a single replica can handle as many requests as the design of the app allows. Also, Knative and other platforms can be configured to have a minimum number of replicas, so they might be well suited even for the applications with a mostly constant flow of traffic since every application does (or should) need scaling sooner or later. The only way to avoid that need is to overprovision applications and give them as much memory and CPU as their peak loads require.

All in all, if it can run in a container, it can be converted into a serverless deployment, as long as we understand that smaller applications with faster boot-up times are better candidates than others. However, boot-up time is not the only rule, nor it is the most important one. If there is a rule we should follow when deciding whether to run an application as serverless, it is related to the state. Or, to be more precise, the lack of it. If an application is stateless, it might be the right candidate for serverless computing.

Now, let us imagine that you have an application that is not the right candidate to be serverless. Does that mean that we cannot rip any benefits from frameworks like Knative? We can, since there is still the question of deployments to different environments.

Typically, we have permanent and temporary environments. The examples of the former would be staging and production. If we do not want our application to be serverless in production, we will probably not want it to be any different in staging. Otherwise, the behavior would be different, and we could not say that we tested precisely the same behavior as the one we expect to run in production. So, in most cases, if an application should not be serverless in production, it should not be serverless in any other permanent environment. But, that does not mean that it shouldn't be serverless in temporary environments.

Let's take an environment in which we deploy an application as a result of making a pull request as an example. It would be a temporary environment since we'd remove it the moment that pull request is closed. Its time span

is relatively short. It could exist for a few minutes, but sometimes that could be days or even weeks. It all depends on how fast we are in closing pull requests.

Nevertheless, there is a high chance that the application deployed in such a temporary environment will have low traffic. We would typically run a set of automated tests when the pull request is created or when we make changes to it. That would certainly result in a traffic spike. But, after that, the traffic would be much lower and most of the time non-existent. We might open the application to have a look at it, we might run some manual tests, and then we would wait for the pull request to be approved or for someone to push additional changes if some issues or inconsistencies were found. That means that the deployment in question would be unused most of the time. Still, if it would be a “traditional” deployment, it would occupy resources for no particular reason. That might even discourage us from making temporary environments due to high costs.

Given that deployments based on pull requests are not used for final validations before deploying to production (that’s what permanent environments are for), we do not need to insist that they are the same as production. On the other hand, the applications in such environments are mostly unused. Those two facts lead us to conclude that temporary (often pull-request based) environments are a great candidate for serverless computing, no matter the deployment type we use in permanent environments (e.g., staging and production).

Now that we saw some of the use cases for serverless computing, there is still an important one that we did not discuss.

## Why Do We Need Jenkins X To Be Serverless?

There are quite a few problems with the traditional Jenkins. Most of us already know them, so I’ll repeat them only briefly. Jenkins (without X) does not scale, it is not fault-tolerant, its resource usage is heavy, it is slow, it is not API-driven, and so on. In other words, it was not designed yesterday, but when those things were not as important as they are today.



Jenkins had to go away for Jenkins X to take its place.

Initially, Jenkins X had a stripped-down version of Jenkins but, since the release 2, not a single line of the traditional Jenkins is left in Jenkins X. Now it is fully serverless thanks to Tekton and a lot of custom code written from scratch to support the need for a modern Kubernetes-based solution.

Excluding a very thin layer that mostly acts as an API gateway, Jenkins X is fully serverless. Nothing runs when there are no builds, and it scales to accommodate any load. That might be the best example of serverless computing we can have.

Continuous integration and continuous delivery pipeline runs are temporary by their nature. When we make a change to a Git repository, it notifies the cluster, and a set of processes are spun. Each Git webhook request results in a pipeline run that builds, validates, and deploys a new release and, once those processes are finished, it disappears from the system. Nothing is executing when there are no pipeline runs, and we can have as many of them in parallel as we need. It is elastic and resource-efficient, and the heavy lifting is done by Tekton.



Continuous integration and continuous delivery tools are probably one of the best examples of a use-case that fits well in serverless computing concepts.

## What Is Tekton And How Does It Fit Jenkins X?

Those of you using serverless Jenkins X already experienced Knative, of sorts. Tekton is a spin-off project of Knative, and it is the essential component in Jenkins X. It is in charge of creating pipeline runs (a special type of Pods) when needed and destroying them when finished. Thanks to Tekton, the total footprint of serverless Jenkins X is very small when idle. Similarly, it allows the solution to scale to almost any size when that is needed.

Tekton is designed only for “special” types of processes, mostly those associated with continuous integration and continuous delivery pipelines. It is not, however, suited for applications designed to handle requests, and those are most of the applications we are developing. So, why am I talking about Tekton if it does not allow us to run our applications as serverless? The answer lies in Tekton’s father.

Tekton is a Knative spin-off. It was forked from it in hopes of providing better CI/CD capabilities. Or, to be more precise, Tekton was born out of the [Knative Build](#) component, which is now considered deprecated. Nevertheless, Knative continues being the most promising way to run serverless applications in Kubernetes. It is the father of Tekton, which we've been using for a while now, given that it is an integral part of serverless Jenkins X.

Now, I could walk you through the details of Knative definitions, but that would be out of the scope of this subject. It's about Jenkins X, not about Knative and other platforms for running serverless applications. But, my unwillingness to show you the ups and downs of Knative does not mean that we cannot use it. As a matter of fact, Jenkins X already provides means to select whether we want to create a quickstart or import an existing project that will be deployed as a serverless application using Knative. We just need to let Jenkins X know that's what we want, and it'll do the heavy lifting of creating the definition (YAML file) that we need.

So, Jenkins X is an excellent example of both a set of serverless applications that constitute the solution, as well as a tool that allows us to convert our existing applications into serverless deployments. All we have to do to accomplish the latter is to express that as our desire. Jenkins X will do all the heavy lifting of creating the correct definitions for our applications as well as to move them through their life-cycles.

## **Creating A Kubernetes Cluster With Jenkins X And Importing The Application**

If you kept the cluster from the previous chapter, you can skip this section. Otherwise, we'll need to create a new Jenkins X cluster.



All the commands from this chapter are available in the [16-serverless-apps.sh](#) Gist.

For your convenience, the Gists from the previous chapter are available below, as well.

- Create a new serverless **GKE** cluster: [gke-jx-serverless.sh](#)

Now we are ready to work on creating the first serverless application using Knative.

## Installing Gloo and Knative

We could visit Knative documentation and follow the instructions to install it and configure it. Then we could reconfigure Jenkins X to use it. But we won't do any of that, because Jenkins X already comes with a method to install and integrate Knative. To be more precise, Jenkins X allows us to install Gloo addon, which, in turn, will install Knative.

[Gloo](#) is a Kubernetes ingress controller, and API gateway. The main reason for using it in our context is because of its ability to route requests to applications managed and autoscaled by Knative. The alternative to Gloo would be Istio, which, even though it's very popular, is too heavy and complex.

Now that we know the “elevator pitch” for Gloo, we can proceed by installing the `gloooctl` CLI.

Please follow the [Install command line tool \(CLI\)](#) instructions.

Now we can use `gloooctl` to install Knative.

```
1 gloooctl install knative \
2     --install-knative-version=0.9.0
```

The process installed Gloo and Knative in our cluster.

There's one more thing missing for us to be able to run serverless applications using Knative. We need to configure it to use a domain (in this case `nip.io`). So, the first step is to get the IP of the Knative service. However, the command differs depending on whether you're using EKS or some other Kubernetes flavor.



Please run the command that follows only if you are **NOT** using **EKS** (e.g., GKE, AKS, etc.).

```
1 KNATIVE_IP=$(kubectl \
2     --namespace gloo-system \
```

```
3     get service knative-external-proxy \
4     --output jsonpath=".status.loadBalancer.ingress[0].ip")
```



Please run the commands that follow only if you are using **EKS**.

```
1 KNATIVE_HOST=$(kubectl \
2   --namespace gloo-system \
3   get service knative-external-proxy \
4   --output jsonpath=".status.loadBalancer.ingress[0].hostname")
5
6 export KNATIVE_IP=$(dig +short $KNATIVE_HOST \
7   | tail -n 1)"
```

To be on the safe side, we'll output the retrieved IP.

```
1 echo $KNATIVE_IP
```

If the output is an IP, everything is working smoothly so far. If it's empty, the load balancer was probably not yet created. If that's the case, wait for a few moments and try it again.

Now we can change Knative configuration.

```
1 echo "apiVersion: v1
2 kind: ConfigMap
3 metadata:
4   name: config-domain
5   namespace: knative-serving
6 data:
7   $KNATIVE_IP.nip.io: \"\"\" \
8     | kubectl apply --filename -"
```

We used the IP of the LoadBalancer Service that was created during the Knative installation as a `nip.io` address in the Knative configuration. From now on, all applications deployed using Knative will be exposed using that address.

Let's take a closer look at what we got by exploring the Namespaces.

```
1 kubectl get namespaces
```

The output is as follows.

```
1 NAME          STATUS AGE
2 default       Active 67m
3 gloo-system   Active 2m1s
4 jx            Active 66m
```

```
5 jx-production Active 59m
6 jx-staging Active 59m
7 knative-serving Active 117s
8 kube-public Active 67m
9 kube-system Active 67m
```

We can see that we got two new Namespaces. As you can probably guess, `gloo-system` contains Gloo components, while Knative runs in `knative-serving`. Keep in mind that we did not get all the Knative components, but only `serving`, which is in charge of running Pods as serverless loads.

Now, I could go into detail and explain the function of every Pod, service, CRD, and other components running in `gloo-system` and `knative-serving` Namespaces. But I feel that would be a waste of time. You can get that information yourself by exploring Kubernetes resources running in those Namespaces, or by going through the official documentation. What matters, for now, is that we got everything Jenkins X needs to convert your applications into serverless deployments.

We’re almost done with the setup. Knative is installed in our cluster, but we still need to tell Jenkins X to use it as a default deployment mechanism. We can do that with the command that follows.

```
1 jx edit deploy \
2   --team \
3   --kind knative \
4   --batch-mode
```

From this moment on, all new projects will be serverless, unless we say otherwise. If you choose to change your mind, please re-run the same command, with the `default` kind instead of `knative`.

Let’s create a new project and check it out.

## Creating A New Serverless Application Project

Jenkins X does its best to be easy for everyone and not to introduce unnecessary complexity. True to that goal, there is nothing “special” users need to do to create a new project with serverless deployments. There is no additional command, nor there are any extra arguments. The `jx edit deploy` command already told Jenkins X that we want all new projects to be serverless by default, so all there is for us to do is to create a new quick start.

```
1 jx create quickstart \
2   --filter golang-http \
3   --project-name jx-knative \
4   --batch-mode
```

As you can see, that command was no different than any other quick start we created earlier. We needed a project with a unique name, so the only (irrelevant) change is that this one is called `jx-knative`.

If you look at the output, there is nothing new there either. If someone else changed the team's deployment kind, you would not even know that a quick start will end with the first release running in the staging environment in the serverless fashion.

There is one difference, though, and we need to enter the project directory to find it.

```
1 cd jx-knative
```

Now, there is only one value that matters, and it is located in `values.yaml`.

```
1 cat charts/jx-knative/values.yaml
```

The output, limited to the relevant parts, is as follows.

```
1 ...
2 # enable this flag to use knative serve to deploy the app
3 knativeDeploy: true
4 ...
```

As you can see, the `knativeDeploy` variable is set to `true`. All the past projects, at least those created after May 2019, had that value set to `false`, simply because we did not have the Gloo addon installed, and our deployment setting was set to `default` instead of `knative`. But, now that we changed that, `knativeDeploy` will be set to `true` for all the new projects unless we change the deployment setting again.

Now, you might be thinking to yourself that a Helm variable does not mean much by itself unless it is used. If that's what's passing through your head, you are right. It is only a variable, and we are yet to discover the reason for its existence.

Let's take a look at what we have in the Chart's `templates` directory.

```
1 ls -1 charts/jx-knative/templates
```

The output is as follows.

```
1 NOTES.txt
2 _helpers.tpl
3 deployment.yaml
4 ksVC.yaml
5 service.yaml
```

We are already familiar with `deployment.yaml` and `service.yaml` files, but we might have missed a crucial detail. So, let's take a look at what's inside one of them.

```
1 cat charts/jx-knative/templates/deployment.yaml
```

The output, limited to the top and the bottom parts, is as follows.

```
1 {{- if .Values.knativeDeploy }}
2 {{- else }}
3 ...
4 {{- end }}
```

We have the `{{- if .Values.knativeDeploy }}` instruction that immediately continues into `{{- else }}`, while the whole definition of the deployment is between `{{- else }}` and `{{- end }}`. While that might look strange at the first look, it actually means that the Deployment resource should be created only if `knativeDeploy` is set to `false`. If you take a look at the `service.yaml` file, you'll notice the same pattern. In both cases, the resources are created only if we did not select to use Knative deployments. That brings us to the `ksvc.yaml` file.

```
1 cat charts/jx-knative/templates/ksvc.yaml
```

The output is as follows.

```
1 {{- if .Values.knativeDeploy }}
2 apiVersion: serving.knative.dev/v1alpha1
3 kind: Service
4 metadata:
5 {{- if .Values.service.name }}
6   name: {{ .Values.service.name }}
7 {{- else }}
8   name: {{ template "fullname" . }}
9 {{- end }}
10  labels:
11    chart: "{{ .Chart.Name }}-{{ .Chart.Version | replace "+" "_" }}"
12 spec:
13   runLatest:
14     configuration:
15       revisionTemplate:
16         spec:
17           container:
18             image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
```

```

19         imagePullPolicy: {{ .Values.image.pullPolicy }}
20     env:
21 {{- range $pkey, $pval := .Values.env --}}
22         - name: {{ $pkey }}
23         value: {{ quote $pval }}
24 {{- end --}}
25     livenessProbe:
26         httpGet:
27             path: {{ .Values.probePath }}
28             initialDelaySeconds: {{ .Values.livenessProbe.initialDelaySeconds }}
29             periodSeconds: {{ .Values.livenessProbe.periodSeconds }}
30             successThreshold: {{ .Values.livenessProbe.successThreshold }}
31             timeoutSeconds: {{ .Values.livenessProbe.timeoutSeconds }}
32     readinessProbe:
33         httpGet:
34             path: {{ .Values.probePath }}
35             periodSeconds: {{ .Values.readinessProbe.periodSeconds }}
36             successThreshold: {{ .Values.readinessProbe.successThreshold }}
37             timeoutSeconds: {{ .Values.readinessProbe.timeoutSeconds }}
38     resources:
39 {{ toYaml .Values.resources | indent 14 }}
40 {{- end --}}

```

To begin with, you can see that the conditional logic is reversed. The resource defined in that file will be created only if the `knativeDeploy` variable is set to `true`.

We won't go into details of the specification. I'll only say that it is similar to what we'd define as a Pod specification, and leave you to explore [Knative Serving API spec](#) on your own. Where Knative definition differs significantly from what we're used to when, let's say, we work with Deployments and StatefulSets, is that we do not need to specify many of the things. There is no need for creating a Deployment, that defines a ReplicaSet, that defines Pod templates. There is no definition of a Service associated with the Pods. Knative will create all the objects required to convert our Pods into a scalable solution accessible to our users.

We can think of the Knative definition as being more developer-friendly than other Kubernetes resources. It dramatically simplifies things by making some assumptions. All the Kubernetes resources we're used to seeing (e.g., Deployment, ReplicaSet, Service) will still be created together with quite a few others. The significant difference is not only in what will be running in Kubernetes but also in how we define what we need. By focusing only on what really matters, Knative removes clutter from YAML files we usually tend to create.

Now, let's see whether the activity of the pipeline run initiated by pushing the initial commit to the newly created repository is finished.

```
1 jx get activities \
2   --filter jx-knative \
3   --watch
```

Unless you are the fastest reader on earth, the pipeline run should have finished, and you'll notice that there is no difference in the steps. It is the same no matter whether we are using serverless or any other type of deployment. So, feel free to stop the activity by pressing *ctrl+c*, and we'll take a look at the Pods and see whether that shows anything interesting.

Before we take a look at the Pods of the new application deployed to the staging environment, we'll confirm that the latest run of the staging environment pipeline is finished as well.

```
1 jx get activities \
2   --filter environment-jx-rocks-staging/master \
3   --watch
```

Feel free to press *ctrl+c* when the staging environment pipeline run is finished.

Now we can have a look at the Pods running as part of our serverless application.

```
1 kubectl \
2   --namespace jx-staging \
3   get pods \
4   --selector serving.knative.dev/service=jx-knative
```

The output is as follows.

```
1 NAME           READY STATUS    RESTARTS AGE
2 jx-knative-... 2/2   Running  0          84s
```



If the output states that `no resources were found`, enough time passed without any traffic, and the application was scaled to zero replicas. We'll see a similar effect and comment on it a few more times. Just keep in mind that the next command that describes the Pod will not work if the Pod was already removed.

The Pod is there, as we expected. The strange thing is the number of containers. There are two, even though our application needs only one.

Let's describe the Pod and see what we'll get.

```
1 kubectl \
2   --namespace jx-staging \
3   describe pod \
4   --selector serving.knative.dev/service=jx-knative
```

The output, limited to the relevant parts, is as follows.

```
1 ...
2 Containers:
3 ...
4   queue-proxy:
5 ...
6     Image: gcr.io/knative-releases/github.com/knative/serving/cmd/queue@sha256:...
7 ...
```

The `queue-proxy` container was “injected” into the Pod. It serves as a proxy responsible for request queue parameters, and it reports metrics to the Autoscaler. In other words, requests are reaching our application through this container. Later on, when we explore scaling our Knative-based applications, that container will be the one responsible for providing metrics used to make scaling-related decisions.

Let’s see which other resources were created for us.

```
1 kubectl \
2   --namespace jx-staging \
3   get all
```

The output, limited to the relevant parts, is as follows.

```
1 ...
2 service/jx-knative ...
3 service/jx-knative-svtns-service ...
4 ...
5 deployment.apps/jx-knative-...
6 ...
7 replicaset.apps/jx-knative-...
8 ...
9 podautoscaler.autoscaling.internal.knative.dev/jx-knative-...
10 ...
11 image.caching.internal.knative.dev/jx-knative-...
12 ...
13 clusteringress.networking.internal.knative.dev/route-...
14 ...
15 route.serving.knative.dev/jx-knative ...
16 ...
17 service.serving.knative.dev/jx-knative ...
18 ...
19 configuration.serving.knative.dev/jx-knative ...
20 ...
21 revision.serving.knative.dev/jx-knative-...
```

As you can see, quite a few resources were created from a single YAML definition with a (`serving.knative.dev`) Service. There are some core

Kubernetes resources we are likely already familiar with, like Deployment, ReplicaSet, Pod, Service. Even if that would be all we've got, we could already conclude that Knative service simplifies things since it would take us approximately double the lines in YAML to define the same resources (Deployment and Service, the rest was created by those) ourselves. But we got so much more. There are seven or more resources created from Knative specific Custom Resource Definitions (CRDs). Their responsibilities differ. One (`podautoscaler.autoscaling`) is in charge of scaling based on the number of requests or other metrics, the other (`image.caching`) of caching of the image so that boot-up time is faster, a few are making sure that networking is working as expected, and so on and so forth. We'll get more familiar with those features later.

There is one inconvenience, though. As of today (July 7), `get applications` does not report Knative-based applications correctly.

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

The output is as follows.

```
1 APPLICATION STAGING PODS URL
2 knative      svtns
```

The serverless application is not reported correctly by Jenkins X. Hopefully, that will be fixed soon. Until then, feel free to monitor the progress through the [issue 4635](#).

Knative defines its own Service that, just like those available in the Kubernetes core, can be queried to get the domain through which we can access the application. We can query it just as we would query the “normal” Service, the main difference being that it is called `ksvc`, instead of `svc`. We'll use it to retrieve the address through which we can access and, therefore, test whether the newly deployed serverless application works as expected.

```
1 ADDR=$(kubectl \
2   --namespace jx-staging \
3   get ksvc jx-knative \
4   --output jsonpath=".status.url")
5
6 echo $ADDR
```

The output should be similar to the one that follows.

```
1 jx-knative.jx-staging.35.243.171.144.nip.io
```

As you can see, the pattern is the same no matter whether it is a “normal” or a Knative service. Jenkins X is making sure that the URLTemplate we explored in the [Changing URL Patterns](#) subchapter is applied no matter the type of the Service or the Ingress used to route external requests to the application. In this case, it is the default one that combines the name of the service (`jx-knative`) with the environment (`jx-staging`) and the cluster domain (`35.243.171.144.nip.io`).

Now comes the moment of truth. Is our application working? Can we access it?

```
1 curl "$ADDR"
```

The good news is that we did get the `Hello` greeting as the output, so the application is working. But that might have been the slowest response you ever saw from such a simple application. Why did it take so long? The answer to that question lies in the scaling nature of serverless applications. Since no one sent a request to the app before, there was no need for it to run any replica, and Knative scaled it down to zero a few minutes after it was deployed. The moment we sent the first request, Knative detected it and initiated scaling that, after a while, resulted in one replica running inside the cluster. As a result, we received the familiar greeting, only after the image is pulled, the Pod was started, and the application inside it was initiated. Don’t worry about that “slowness” since it manifests itself only initially before Knative creates the cache. You’ll see soon that the boot-up time will be very fast from now on.

So, let’s take a look at that “famous” Pod that was created out of thin air.

```
1 kubectl \
2   --namespace jx-staging \
3   get pods \
4   --selector serving.knative.dev/service=jx-knative
```

The output is as follows.

```
1 NAME          READY STATUS  RESTARTS AGE
2 jx-knative-... 2/2   Running 0        24s
```

We can see a single Pod created a short while ago. Now, let’s observe what we’ll get with a little bit of patience.

Please wait for seven minutes or more before executing the command that follows.

```
1 kubectl --namespace jx-staging \
2     get pods \
3     --selector serving.knative.dev/service=jx-knative
```

The output shows that no resources were found. The Pod is gone. No one was using our application, so Knative removed it to save resources. It scaled it down to zero replicas.

If you're anything like me, you must be wondering about the configuration. What are the parameters governing Knative scaling decisions? Can they be fine-tuned?

The configuration that governs scaling is stored in the `config-autoscaler` ConfigMap.

```
1 kubectl --namespace knative-serving \
2     describe configmap config-autoscaler
```

The output is a well-documented configuration example that explains what we'd need to do to change any aspect of Knative's scaling logic. It is too big to be presented in a book, so I'll leave it to you to explore it.

In a nutshell, Knative's scaling algorithm is based on the average number of concurrent requests. By default, it will try to target a hundred parallel requests served by a single Pod. That would mean that if there are three hundred concurrent requests, the system should scale to three replicas so that each can handle a hundred.

Now, the calculation for the number of Pods is not as simple as the number of concurrent requests divided by a hundred (or whatever we defined the `container-concurrency-target-default` variable). The Knative scaler calculated the average number of parallel requests over a sixty seconds window, so it takes a minute for the system to stabilize at the desired level of concurrency. There is also a six seconds window that might make the system enter into the panic mode if, during that period, the number of requests is more than double the target concurrency.

I'll let you go through the documentation and explore the details. What matters, for now, is that the system, as it is now, should scale the number of Pods if we send it more than a hundred parallel requests.

Before we test Knative's scaling capabilities, we'll check whether the application scaled down to zero replicas.

```

1 kubectl \
2   --namespace jx-staging \
3   get pods \
4   --selector serving.knative.dev/service=jx-knative

```

If the output states that no resources were found, the Pods are gone, and we can proceed. Otherwise, wait for a while longer and repeat the previous command.

We ensured that no Pods are running only to simplify the “experiment” that follows. When nothing is running, the calculation is as simple as the number of concurrent requests divided by the target concurrency equals the number of replicas. Otherwise, the calculation would be more complicated than that, and our “experiment” would need to be more elaborated. We won’t go into those details since I’m sure that you can gather such info from the Knative’s documentation. Instead, we’ll perform a simple experiment and check what happens when nothing is running.

So, we want to see what result of sending hundreds of parallel requests to the application. We’ll use [Siege](#) for that. It is a small and simple tool that allows us to stress test a single URL. It does that by sending parallel requests to a specific address.

Since I want to save you from installing yet-another-tool, we’ll run Siege inside a Pod with a container based on the [yokogawa/siege](#) image. Now, we’re interested in finding out how Knative deployments scale based on the number of requests, so we’ll also need to execute `kubectl get pod` command to see how many Pods were created. But, since Knative scales both up and down, we’ll need to be fast. We have to make sure that the Pods are retrieved as soon as the siege is finished. We’ll accomplish that by concatenating the two commands into one.

```

1 kubectl run siege \
2   --image yokogawa/siege \
3   --generator "run-pod/v1" \
4   -it --rm \
5   -- --concurrent 300 --time 20S \
6   "$ADDR" \
7   && kubectl \
8   --namespace jx-staging \
9   get pods \
10  --selector serving.knative.dev/service=jx-knative

```

We executed three hundred concurrent requests (`-c 300`) for twenty seconds (`-t 20S`). Immediately after that we retrieved the Pods related to the `jx-knative` Deployment. The combined output is as follows.

```

1 If you don't see a command prompt, try pressing enter.
2
3 Lifting the server siege...      done.
4
5 Transactions:          4920 hits
6 Availability:           100.00 %
7 Elapsed time:            19.74 secs
8 Data transferred:        0.20 MB
9 Response time:           1.16 secs
10 Transaction rate:       249.24 trans/sec
11 Throughput:              0.01 MB/sec
12 Concurrency:             289.50
13 Successful transactions: 4920
14 Failed transactions:     0
15 Longest transaction:    6.25
16 Shortest transaction:   0.14
17
18 FILE: /var/log/siege.log
19 You can disable this annoying message by editing
20 the .siegerc file in your home directory; change
21 the directive 'show-logfile' to false.
22 Session ended, resume using 'kubectl attach siege -c siege -i -t' command when th
23 od is running
24 pod "siege" deleted
25 NAME                  READY STATUS  RESTARTS AGE
26 jx-knative-cv152-deployment-... 2/2   Running 0      18s
27 jx-knative-cv152-deployment-... 2/2   Running 0      20s
28 jx-knative-cv152-deployment-... 2/2   Running 0      18s

```

The `siege` output shows us that it successfully executed 4920 requests within 19.74 seconds, and all that was done with the concurrency of almost three hundred.

What is more interesting is that we got three Pods of the `jx-knative` application. If we go back to the values in the ConfigMap `config-autoscaler`, we'll see that Knative tries to maintain one replica for every hundred concurrent requests. Since we sent almost triple that number of concurrent requests, we got three Pods. Later on, we'll see what Knative does when that concurrency changes. For now, we'll focus on “fine-tuning” Knative’s configuration specific to our application.

```

1 cat charts/jx-knative/templates/ksvc.yaml \
2   | sed -e \
3     's@revisionTemplate:@revisionTemplate:\\
4       metadata:\\
5         annotations:\\
6           autoscaling.knative.dev/target: "3"\\
7           autoscaling.knative.dev/maxScale: "5"@g' \
8   | tee charts/jx-knative/templates/ksvc.yaml

```

We modified the `ksvc.yaml` file by adding a few annotations. Specifically, we set the `target` to 3 and `maxScale` to 5. The former should result in Knative scaling our application with every three concurrent requests. The latter, on the other hand, will prevent the system from having more than 5

replicas. As a result, we'll have better-defined parameters that will be used to decide when to scale, but we'll also be protected from the danger of “getting more than we are willing to have.”

Now, let's push the changes to the GitHub repository and confirm that the pipeline run that will be triggered by that will complete successfully.

```
1 git add .
2
3 git commit -m "Added Knative target"
4
5 git push --set-upstream origin master
6
7 jx get activities \
8     --filter jx-knative \
9     --watch
```

Feel free to press the *ctrl+c* key to stop watching the activities when the run is finished.

As before, we'll also confirm that the new release was deployed to the staging environment by monitoring the activities of the `environment-jx-rocks-staging` pipeline runs.

```
1 jx get activities \
2     --filter environment-jx-rocks-staging/master \
3     --watch
```

Please press *ctrl+c* to cancel the watcher once the new activity is finished.

Finally, the last step we'll execute before putting the application under siege is to double-check that it is still reachable by sending a single request.

```
1 curl "$ADDR/"
```

You should see the familiar message, and now we're ready to put the app under siege. Just as before, we'll concatenate the command that will output the Pods related to the `jx-knative` app.

```
1 kubectl run siege \
2     --image yokogawa/siege \
3     --generator "run-pod/v1" \
4     -it --rm \
5     -- --concurrent 400 --time 60S \
6     "$ADDR" \
7     && kubectl \
8     --namespace jx-staging \
9     get pods \
10    --selector serving.knative.dev/service=jx-knative
```

We sent a stream of four hundred (`-c 400`) concurrent requests over one minute (`-t 60s`). The output is as follows.

```
1 If you don't see a command prompt, try pressing enter.
2
3 Lifting the server siege...      done.
4
5 Transactions:          19078 hits
6 Availability:          100.00 %
7 Elapsed time:           59.63 secs
8 Data transferred:       0.78 MB
9 Response time:          1.04 secs
10 Transaction rate:      319.94 trans/sec
11 Throughput:            0.01 MB/sec
12 Concurrency:           332.52
13 Successful transactions: 19078
14 Failed transactions:   0
15 Longest transaction:   8.29
16 Shortest transaction:  0.01
17
18 FILE: /var/log/siege.log
19 You can disable this annoying message by editing
20 the .siegerc file in your home directory; change
21 the directive 'show-logfile' to false.
22 Session ended, resume using 'kubectl attach siege -c siege -i -t' command when th
23 od is running
24 pod "siege" deleted
25 NAME          READY STATUS RESTARTS AGE
26 jx-knative-... 2/2   Running 0        58s
27 jx-knative-... 2/2   Running 0        58s
28 jx-knative-... 2/2   Running 0        61s
29 jx-knative-... 2/2   Running 0        58s
30 jx-knative-... 2/2   Running 0        58s
```

If we'd focus only on the `target` annotation we set to 3, we would expect to have over one hundred Pods, one for every three concurrent requests. But, we also set the `maxScale` annotation to 5. As a result, only five Pods were created. Knative started scaling the application to accommodate three requests per Pod rule, but it capped at five to match the `maxScale` annotation.

Now, let's see what happens a while later. Please execute the command that follows a minute (but not much more) after than the previous command.

```
1 kubectl \
2   --namespace jx-staging \
3   get pods \
4   --selector serving.knative.dev/service=jx-knative
```

The output is as follows.

```
1 NAME          READY STATUS RESTARTS AGE
2 jx-knative-... 2/2   Running 0        2m32s
```

As we can see, Knative scaled-down the application to one replica shortly after the burst of requests stopped. It intentionally did not scale down to zero right away to avoid potentially slow boot-up time. Now, that will change soon as well, so let's see what happens after another short pause.

Please wait for some five to ten minutes more before executing the command that follows.

```
1 kubectl \
2   --namespace jx-staging \
3   get pods \
4   --selector serving.knative.dev/service=jx-knative
```

This time the output states that no resources were found. Knative observed that no traffic was coming to the application for some time and decided that it should scale down the application to zero replicas. As you already saw, that will change again as soon as we send additional requests to the application. For now, we'll focus on one more annotation.

In some cases, we might not want to allow Knative to scale to zero replicas. Our application's boot-time might be too long, and we might not want to risk our users waiting for too long. Now, that would not happen often since such situations would occur only if there is no traffic for a prolonged time. Still, some applications might take seconds or even minutes to boot up. I'll skip a discussion in which I would try to convince you that you should not have such applications or that, if you do, you should redesign them or even throw them to thrash and start over. Instead, I'll just assume that you do have an app that is slow to boot up but that you still see the benefits in adopting Knative for, let's say, scaling up. So, how do we prevent it from scaling to zero replicas, and yet allow it to scale to any other number?

Let's give it a try with the command that follows.

```
1 cat charts/jx-knative/templates/ksvc.yaml \
2   | sed -e \
3     's@autoscaling.knative.dev/target: "3"@autoscaling.knative.dev/target: "3" \
4       autoscaling.knative.dev/minScale: "1"@g' \
5   | tee charts/jx-knative/templates/ksvc.yaml
```

We used `sed` to add `minScale` annotation set to 1. You can probably guess how it will behave, but we'll still double-check that everything works as expected.

Before we proceed, please note that we used only a few annotations and that Knative offers much more fine-tuning. As an example, we could tell Knative to use HorizontalPodAutoscaler for scaling decisions. I'll leave it up to you too to check out the project's documentation, and we'll get back to our task of preventing Knative from scaling our application to zero replicas.

```
1 git add .
2
3 git commit -m "Added Knative minScale"
4
5 git push
6
7 jx get activities \
8   --filter jx-knative \
9   --watch
```

We pushed changes to the GitHub repository, and we started watching the `jx-native` activity created as a result of making changes to our source code.

Feel free to stop watching the activities (press `ctrl+c`) once you confirm that the newly started pipeline run completed successfully.

Just as before, we'll also confirm that the pipeline run of the staging environment completed successfully as well.

```
1 jx get activities \
2   --filter environment-jx-rocks-staging/master \
3   --watch
```

Just as a few moments ago, press `ctrl+c` when the new pipeline run is finished.

Now, let's see how many Pods we have.

```
1 kubectl \
2   --namespace jx-staging \
3   get pods \
4   --selector serving.knative.dev/service=jx-knative
```

The output should show that only one Pod is running. However, that might not be definitive proof that, from now on, Knative will never scale our app to zero replicas. To confirm that “for real”, please wait for, let's say, ten minutes, before retrieving the Pods again.

```
1 kubectl \
2   --namespace jx-staging \
```

```
3     get pods \
4     --selector serving.knative.dev/service=jx-knative
```

The output is still the same single Pod we saw earlier. That proves that our application's minimum number of replicas is maintained at 1, even if it does not receive any requests for a prolonged time.

There's probably no need to give you instructions on how to check whether the application scales up if we start sending it a sufficient number of concurrent requests. Similarly, we should know by now that it will also scale down if the number of simultaneous requests decreases. Everything is the same as it was, except that the minimum number of replicas is now 1 (it is zero by default).

This last exercise of adding the `minScale` annotation converted our application from serverless to a microservice or whichever other architecture our app had.

## Using Serverless Deployments With Pull Requests

So far, all the changes we made were pushed to a branch. That was on purpose since one of my goals was to show you the benefits of using serverless deployments with pull requests.

The percentage of the apps running as serverless inevitably varies. Some might have all the stateless applications running as serverless, while others might have none. Between those two extremes can be all shades of gray. While I cannot predict how many apps you should run as serverless deployments, what I can guess with a high level of certainty is that you'll use Knative much more in temporary environments like those created for pull requests than in permanent ones like staging and production. The reason for such a bold statement lies in the differences in purpose for running applications.

An application running in a preview environment is to validate (automatically and/or manually) that a change we'd like to merge to the master branch is most likely a good one and that it is relatively safe to merge it with production code. However, the validations we're running against a release in a preview environment are often not the final ones. A preview environment is often not the same as production. It might differ in size, it might not be integrated with all the other applications, it might not have all the data we need, and so on and so forth. That's why we have the

staging and other non-production permanent environments. Their primary purpose is often to provide a production-like environment where we can run the final set of validations before promoting to production. If we'd do that for each preview environment, our resource usage would probably go through the roof.

Imagine having hundreds or even thousands of open pull requests and that each is deployed to a preview environment. Now, imagine also that each pull request is a full-blown system. How much CPU and memory would that require? That was a rhetorical question I do not expect you to answer with a precise number. Saying that the answer is “a lot” or “much more than I can afford” should suffice. For that reason, our preview environments used with pull requests usually contain only the application in question. We might add one or two essential applications it integrates with, but we seldom deploy the full system there. Instead, we use mocks and stubs to test applications in those environments. That, by itself, should save a lot of resources, while still maintaining the advantages of preview environments. Nevertheless, we can do better than that.

Preview environments are one of the best use-cases for serverless loads. We might choose to run our application as serverless in production, or we might decide not to do so. The decision will depend on many factors, user experience being one of the most important ones. If our application scales to zero replicas due to lack of traffic, when a request does come, it might take a while until the process is in the newly spun replica is fully initialized. It should be obvious why forcing our users to wait for more than a second or two before receiving the first response is a bad idea. They are impatient and are likely to go somewhere else if they are unhappy with us. Also, our production traffic is likely going to be more demanding and less volatile than the one exercised over deployments in preview environments.

When a preview environment is created as a result of creating a pull request, our application is deployed to a unique Namespace. Typically, we would run some automated tests right after the deployment. But what happens after that? The answer is “mostly nothing”. A while later (a minute, an hour, a day, or even a week), someone might open the deployment associated with the pull request and do some manual validations. Those might result in the need for new issues or validations. As a result, that pull request is likely going to be idle for a while, before it

is used again. So, we have mostly unused deployments wasting our memory and CPU.

Given that we established how preview environments represent a colossal waste in resources, we can easily conclude that deployments initiated by pull request are one of the best candidates for serverless computing. But, that would also assume that we do not mind to wait for a while until an application is scaled from zero to one or more replicas. In my experience, that is (almost) never a problem. We cannot put users of our production releases and pull request reviewers on the same level. It should not be a problem if a person who decides to review a pull request and validate the release candidate manually has to wait for a second or even a minute until the application is scaled up. That loss in time is more than justified with much better usage of resources. Before and after the review, our app would use no resources unless it has dependencies (e.g., database) that cannot be converted into serverless deployments. We can gain a lot, even in those cases, when only some of the deployments are serverless. A hundred percent of deployments in preview environments running as serverless is better than, let's say, fifty percent. Still, fifty percent is better than nothing.



Databases in preview or even in permanent environments can be serverless as well. As long as their state is safely stored in a network drive, that should be able to continue operating when scaled from zero to one or more replicas. Nevertheless, databases tend to be slow to boot, especially when having a lot of data. Even though they could be serverless, they are probably not the right candidate. The fact that we can do something does not mean that we should.

Now, let's create a pull request and see how the `jx-knative` behaves.

```
1 git checkout -b serverless
2
3 echo "A silly change" | tee README.md
4
5 git add .
6
7 git commit -m "Made a silly change"
8
9 git push --set-upstream origin serverless
10
11 jx create pullrequest \
12   --title "A silly change" \
13   --body "What I can say?" \
14   --batch-mode
```

We created a branch, we made “a silly change”, and we created a new pull request.

Next, since Jenkins X treats pull requests as yet-another-branch, we’ll store the name of that branch in an environment variable.



Please replace [...] with `PR-[PR_ID]` (e.g., PR-109). You can extract the ID from the last segment of the pull request address.

```
1 BRANCH=[...] # e.g., `PR-1`
```

Now, let’s double-check that the pull request was processed successfully.

```
1 jx get activities \
2   --filter jx-knative/$BRANCH \
3   --watch
```

Feel free to press *ctrl+c* to stop watching the activities when all the steps in the pipeline run were executed correctly.

To see what was deployed to the preview environment, we need to discover the Namespace Jenkins X created for us. Fortunately, it uses a predictable naming convention so we can reconstruct the Namespace name easily using the base Namespace, GitHub user, application name, and the branch.



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

```
1 GH_USER=[...]
2
3 PR_NAMESPACE=$( \
4   echo jx-$GH_USER-jx-knative-$BRANCH \
5   | tr '[:upper:]' '[:lower:]')
6
7 echo $PR_NAMESPACE
```

In my case, the output of the last command was `jx-vfarcic-jx-knative-pr-1`. Yours should be different but still follow the same logic.

Now we can take a look at the Pods running in the preview Namespace.

```
1 kubectl --namespace $PR_NAMESPACE \
2   get pods
```

The output is as follows.

```
1 NAME                      READY STATUS RESTARTS AGE
2 jx-knative-ccql1-deployment... 2/2   Running 0          109s
```

We can see two interesting details just by observing the number of containers in those Pods.

To be on the safe side, we'll confirm that the application deployed in the preview environment is indeed working as expected. To do that, we need to construct the auto-assigned address through which we can access the application.

```
1 PR_ADDR=$(kubectl \
2   --namespace $PR_NAMESPACE \
3   get ksvc jx-knative \
4   --output jsonpath=".status.url")
5
6 echo $PR_ADDR
```

The output should be similar to the one that follows.

```
1 jx-knative.jx-vfarcic-jx-knative-pr-115.34.73.141.184.nip.io
```



Please note that we did not have to “discover” the address. We could have gone to the GitHub pull request screen and clicked the *here* link. We'd need to add `/demo/hello` to the address, but that could still be easier than what we did. Still, I am “freak” about automation and doing everything from a terminal screen, and I have the right to force you to do things my way, at least while you're following the exercises I prepared.

Now comes the moment of truth.

```
1 curl "$PR_ADDR"
```

The output should be already familiar `hello` from message. If, by the time we sent the request, there was already a replica, it was simply forwarded there. If there wasn't, Knative created one.

## Limiting Serverless Deployments To Pull Requests

I already explained that running applications as serverless deployments in preview (pull request) environments is highly beneficial. As a result, you might have the impression that an application must be serverless in all environments. That is certainly not the case. We can, for example, choose to run some applications as serverless deployments in preview environment and run them as “normal” apps in those that are permanent.

To be fair, we could have more complicated arrangements with, for example, running a serverless application in the staging environment but non-serverless in production. However, that would mean that we do not test what we’re deploying to production. After all, serverless applications do not behave in the same way as other types of deployments.

Now, you could argue that preview environments are used for testing, so they should be the same as production. While it is true that testing is the primary function of preview environments, they are not used for the final round of testing. By their nature, preview environments are more lightweight and do not contain the whole system, but only the parts required to (partly) validate pull requests. The “real” or “final” testing is performed in the staging environment if we are performing continuous delivery, or in production, if we are practicing continuous deployment. The latter option would require some form of progressive delivery, which we might explore later. For now, I’ll assume that you are following the continuous delivery model and, therefore, staging environment (or whatever you call it) is the one that should be production-like.

All in all, we’ll explore how to make an application serverless only in preview environments, and continue being whatever it was before in permanent ones.

Since our *jx-knative* application is already serverless by default, we’ll go with the least possible effort and leave it as-is, but disable `knativeDeploy` in values for the staging in production environments.

So, our first order of business to is clone the staging environment repository.

```
1 cd ..
2
3 rm -rf environment-jx-rocks-staging
4
5 git clone https://github.com/$GH_USER/environment-jx-rocks-staging.git
6
7 cd environment-jx-rocks-staging
```

We removed a local copy of the staging repository just in case there is a left-over from one of the previous chapters, we cloned the repo, and we entered inside the local copy.

Now, changing the way an application is deployed to a specific repository is as easy as changing the value of the `knativeDeploy` variable. But, since an environment defines all the applications running in it, we need to specify for which one we're changing the value. Or, to be more precise, since all the apps in an environment are defined as dependencies in `requirements.yaml`, we need to prefix the value with the alias of the dependency. In our case, we have `jx-knative` and potentially a few other applications. We want to ensure that `jx-knative` is not running as serverless in the staging environment.

```
1 echo "jx-knative:
2   knativeDeploy: false" \
3     | tee -a env/values.yaml
```

Now that we added the `jx-knaative.knativeDeploy` variable set to `false`, we can push the changes and let Jenkins X do the job for us.

```
1 git add .
2
3 git commit -m "Removed Knative"
4
5 git pull
6
7 git push
```

Now, even though that push will trigger a new deployment, that will not recreate the required Ingress resource, so we'll need to make a (trivial) change to the application as well. That should result in the new deployment with everything we need for our *jx-knative* application to behave in the staging environment as if it is a “normal” application (not serverless).

```
1 cd ../jx-knative
2
3 git checkout master
4
5 echo "jx-knative rocks" \
6   | tee README.md
7
8 git add .
9
10 git commit -m "Removed Knative"
11
12 git pull
```

```
13  
14 git push
```

Just as before, we'll check the activities of the project pipeline to confirm that it executed successfully.

```
1 jx get activities \  
2   --filter jx-knative/master \  
3   --watch
```

Feel free to stop watching the activities with *ctrl+c*, so that we double-check that the activity triggered by making changes to the staging environment repository is finished as well.

```
1 jx get activities \  
2   --filter environment-jx-rocks-staging/master \  
3   --watch
```

You know what to do now. Press *ctrl+c* when the newly spun activity is finished.

Now, let's check whether the application was indeed deployed as non-serverless to staging.

```
1 kubectl \  
2   --namespace jx-staging \  
3   get pods
```

The output is as follows.

```
1 NAME           READY STATUS RESTARTS AGE  
2 jx-jx-knative-... 1/1   Running 0        78s
```

It's hard to see whether an application is serverless or not by looking at the Pods, so we'll check all the resources related to the `jx-knative` app.

```
1 kubectl \  
2   --namespace jx-staging \  
3   get all \  
4   | grep jx-knative
```

The output is as follows.

```
1 pod/jx-jx-knative-66df89fb74-mgg68  1/1   Running  0          3m29s  
2 service/jx-knative ClusterIP 10.31.254.208 <none>  80/TCP    3m29s  
3 deployment.apps/jx-jx-knative 1/1     1          1          8m18s  
4 replicaset.apps/jx-jx-knative-5cbd9d4799 0      0          0          8m18s  
5 replicaset.apps/jx-jx-knative-66df89fb74 1      1          1          3m30s  
6 release.jenkins.io/jx-knative-0.0.4 jx-knative v0.0.4   https://github.com/vf  
7 cic/jx-knative
```

As you can see, we have a bunch of resources. What matters, in this case, is not that much what we do have, but rather what we don't. There is not `ksvc`. Instead, we got the “standard” resources like a Service, a Deployment, ReplicaSets, etc.

Since you know that I have a paranoid nature, you won't be surprised that we'll double-check whether the application works by sending a request and observing the output.

```
1 ADDR=$(kubectl \
2   --namespace jx-staging \
3   get ing jx-knative \
4   --output jsonpath=".spec.rules[0].host") \
5 
6 echo $ADDR
7 
8 curl "http://$ADDR"
```

If you got the familiar message, the application works and is back to its non-serverless form.

Right now, our preview (pull requests) and production environments feature serverless deployments of *jx-knative*, while staging is back to the “normal” deployment. We should make a similar change to the production repository, but I will not provide instructions for that since they are the same as for the staging environment. Think of that as a (trivial) challenge that you should complete alone.

## What Now?

As you saw, converting our applications into serverless deployments with Knative is trivial. The same can be said for all the projects we start from now on. Jenkins X buildpacks already contain everything we need, and the only action on our part is either to change the `knativeDeploy` variable or to use the `jx edit deploy` command to make Knative deployments default for all new projects.

Now you need to decide whether to continue using the cluster or to destroy it. If you choose to destroy it or to uninstall Jenkins X, you'll find the instructions at the bottom of the Gist you chose at the beginning of this chapter.

If you destroyed the cluster or you uninstalled Jenkins X, please remove the repositories and the local files we created. You can use the commands

that follow for that.



Please replace [...] with your GitHub user.

```
1 cd ..
2
3 GH_USER=[...]
4
5 hub delete -y $GH_USER/environment-jx-rocks-staging
6
7 hub delete -y $GH_USER/environment-jx-rocks-production
8
9 hub delete -y $GH_USER/jx-knative
10
11 rm -rf ~/.jx/environments/$GH_USER/environment-jx-rocks-*
12
13 rm -rf jx-knative
14
15 rm -rf environment-jx-rocks-staging
```

Finally, you might be planning to move into the next chapter right away. If that's the case, there are no cleanup actions to do. Just keep reading.

# Choosing The Right Deployment Strategy

**Carlos Sanchez** co-authored this chapter.

So far, we performed many deployments of our releases. All those created from master branches were deployed to the staging environment, and a few reached production through manual promotions. On top of those, we deployed quite a few releases to preview environments. Nevertheless, except for serverless deployments with Knative, we did not have a say in how an application is deployed. We just assumed that the default method employed by Jenkins X is the correct one. As it happens, the default deployment process used by Jenkins X happens to be the default or, to be more precise, the most commonly used deployment process in Kubernetes. However, that does not necessarily mean that the default strategy is the right one for all our applications.

For many people, deploying applications is transparent or even irrelevant. If you are a developer, you might be focused on writing code and allowing magic to happen. By magic, I mean letting other people and departments figure out how to deploy your code. Similarly, you might be oblivious to deployments. You might be a tester, or you might have some other role not directly related to system administration, operations, or infrastructure. Now, I doubt that you are one of the oblivious. The chances are that you would not be even reading this if that's the case. If, against all bets, you do belong to the deployment-is-not-my-thing group, the only thing I can say is that you are wrong.

Generally speaking, there are two types of teams. The vast majority of us is still working in groups based on types of tasks and parts of application lifecycles. If you're wondering whether that's the type of the team you work in, ask yourself whether you are in development, testing, operations, or some other department? Is your team focused on a fraction of a lifecycle of an application? Are you handing your work to someone else? When you finish writing code, do you give it to the testing department to validate it? When you need to test a live application, are you giving it to operations to deploy it to an environment? Or, to formulate the question on a higher level, are you (your team) in charge only of a part of the lifecycle of your

application? If the answer to any of those question is “yes”, you are NOT working in a self-sufficient team. Now, I’m not going to tell you why that is wrong, nor I’m here to judge you. Instead, I’m only going to state that there is a high probability that you do not know in detail how your application is deployed. As a result, you don’t know how to architecture it properly, you don’t know how to test it well, and so on. That, of course, is not true if you are dedicated only to operations. But, in that case, you might not be aware of the architecture of the application. You might know how the application is deployed, but you might not know whether that is the optimum way to go.

On the other hand, you might be indeed working in a self-sufficient team. Your team might be fully responsible for each aspect of the application lifecycle, from requirements until it is running in production. If that’s the case, your definition of done is likely defined as “it’s running in production and nothing exploded.” Being in a self-sufficient team has a distinct advantage of everyone being aware of every aspect of the application lifecycle. You know the architecture, you can read the code, you understand the tests, and you are aware of how it is deployed. That is not to say that you are an expert in all those and other areas. No one can know everything in depth, but everyone can have enough high-level knowledge of everything while being specialized in something.

Why am I rambling about team organizations? The answer is simple. Deployment strategies affect everyone, no matter whether we are focused only on a single aspect of the application lifecycle or we are in full control. The way we deploy affects the architecture, testing, monitoring, and many other aspects. And not only that, but we can say that architecture, testing, and monitoring affect the way we deploy. All those things are closely related and affect each other in ways that might not be obvious on the first look.

We already learned many of the things Jenkins X does out-of-the-box and quite a few others that could be useful to customize it to behave as we want. But, so far, we mostly ignored deployment strategies. Excluding our brief exploration of serverless deployments with Knative, we always assumed that the application should be deployed using whichever strategy was defined in a build pack. Not only that, but we did not even question whether the types of resources defined in our Helm charts are the right ones. We’ll fill at least one of those holes next.

The time has come to discuss different deployment strategies and answer a couple of questions. Is your application stateful or stateless? Does its architecture permit scaling? How do you roll back? How do you scale up and down? Do you need your application to run always? Should you use Kubernetes Deployments instead of, let's say, StatefulSets? Those are only a few of the questions you need to answer to choose the right deployment mechanism. But, answers to those questions will not serve much unless we are familiar with some of the most commonly used deployment strategies. Not only that knowledge will help us choose which one to pick, but they might even influence the architecture of our applications.

## What Do We Expect From Deployments?

Before we dive into some of the deployment strategies, we might want to set some expectations that will guide us through our choices. But, before we do that, let's try to define what a deployment is.

Traditionally, a deployment is a process through which we would install new applications into our servers or update those that are already running with new releases. That was, more or less, what we were doing from the beginning of the history of our industry, and that is in its essence what we're doing today. But, as we evolved, our requirements were changing as well. Today, say that all we expect is for our releases to run is an understatement. Today we want so much more, and we have technology that can help us fulfill those desires. So, what does "much more" mean today?

Depending on who you speak with, you will get a different list of "desires". So, mine might not be all-encompassing and include every single thing than anyone might need. What follows is what I believe is essential, and what I observed that the companies I worked typically put emphasis. Without further ado, the requirements, excluding the obvious that applications should be running inside the cluster, are as follows.

Applications should be **fault-tolerant**. If an instance of the application dies, it should be brought back up. If a node where an application is running dies, the application should be moved to a healthy node. Even if a whole data center goes down, the system should be able to move the applications that were running there into a healthy one. An alternative would be to recreate the failed nodes or even whole data centers with precisely the same apps that were running there before the outage.

However, that is too slow and, frankly speaking, we moved away from that concept the moment we adopted schedulers. That does not mean that failed nodes and failed data centers should not recuperate, but rather that we should not wait for infrastructure to get back to normal. Instead, we should run failed applications (no matter the cause) on healthy nodes as long as there is enough available capacity.

Fault tolerance might be the most crucial requirement of all. If our application is not running, our users cannot use it. That results in dissatisfaction, loss of profit, churn, and quite a few other adverse outcomes. Still, we will not use fault tolerance as a criterion because Kubernetes makes (almost) everything fault-tolerant. As long as it has enough available capacity, our applications will run. So, even that is an essential requirement, it is off the table because we are fulfilling it no matter the deployment strategy we choose. That is not to say that there is no chance for an application not to recuperate from a failure but instead that Kubernetes provides a reasonable guarantee of fault tolerance. If things do go terribly wrong, we are likely going to have to do some manual actions no matter which deployment strategy we choose.

Long story short, fault-tolerance is a given with Kubernetes, and there's no need to think about it in terms of deployment strategies.

The next in line is **high availability**, and that a trickier one.

Being fault-tolerant means that the system will recuperate from failure, not that there will be no downtime. If our application goes down, a few moments later, it will be up-and-running again. Still, those few moments can result in downtime. Depending on many factors, “few moments” can be translated to milliseconds, seconds, minutes, hours, or even days. It is certainly not the same whether our application is unavailable during milliseconds as opposed to hours. Still, for the sake of brevity, we'll assume that any downtime is bad and look at things as black and white. Either there is, or there isn't downtime. Or, to be more precise, either there is a considerable downtime, or there isn't. What changed over time is what “considerable” means. In the past, having 99% availability was a worthy goal for many. Today, that figure is unacceptable. Today we are talking about how many nines there are after the decimal. For some, 99.99% uptime is acceptable. For others, that could be 99.99999%.

Now, you might say: “my business is important; therefore, I want 100% uptime.” If anyone says that to you, feel free to respond with: “you have no idea what you’re talking about.” Hundred percent uptime is impossible, assuming that by that we mean “real” uptime, and not “my application runs all the time.”

Making sure that our application is always running is not that hard. Making sure that not a single request is ever lost or, in other words, that our users perceive our application as being always available, is impossible. By the nature of HTTP, some requests will fail. Even if that never happens (as it will), network might go down, storage might fail, or some other thing might happen. Any of those is bound to produce at least one request without a response or with a 4xx or 5xx message.

All in all, high-availability means that our applications are responding to our users most of the time. By “most”, we mean at least 99.99%. Even that is a very pessimistic number that would result in one failure for each ten thousand requests.

What are the common causes of unavailability? We already discussed those that tend to be the first associations (hardware and software failures). However, those are often not the primary causes of unavailability. You might have missed something in your tests, and that might cause a malfunction. More often than not, those are not failures caused by “obvious” bugs but rather by those that manifest themselves a while after a new release is deployed. I will not tell you that you should make sure that there are no bugs because that is impossible). Instead, I’ll tell you that you should focus on detecting those that sneak into production. It’s as important to try to avoid bugs as to minimize their effect to as few users as possible. So, our next requirement will be that our deployments should reduce the number of users affected by bugs. We’ll call it **progressive rollout**. Don’t worry if you never heard that term. We’ll explain it in more depth later.

Progressive rollout, as you’ll see later, does allow us to abort upgrades or, to be more precise, not to proceed with them, if something goes wrong. But that might not be enough. We might need not only to abort deployment of a new release but also to roll back what we had before. So, we’ll add **rollback** as yet another requirement.

We'll probably find more requirements directly or indirectly related to high-availability or, to inverse it, to unavailability. For now, we'll leave those aside, and move to yet another vital aspect. We should strive to make our applications **responsive**, and there are many ways to accomplish that. We can design our apps in a certain way, we can avoid congestions and memory leaks, and we can do many other things. However, right now, that's not the focus. We're interested in things that are directly or indirectly related to deployments. With such a limited scope, scalability is the key to responsiveness. If we need more replicas of our application, it should scale up. Similarly, if we do not need as many, it should scale down and free the resources for some other processes if cost savings are not a good enough reason.

Finally, we'll add one more requirement. It would be nice if our applications do not use more resources than it is necessary. We can say that scalability provides that (it can scale up and down) but we might want to take it a step further and say that our applications should not use (almost) any resources when they are not in use. We can call that "nothing when idle" or, use a more commonly used term, serverless.

I'll use this as yet another opportunity to express my disgust with that term given that it implies that there are no servers involved. But, since it is a commonly used one, we'll stick with it. After all, it's still better than calling it function-as-a-service since that is just as misleading as serverless, and it occupies more characters (it is a longer word). However, serverless is not the real goal. What matters is that our solution is **cost-effective**, so that will be our last requirement.

Are those all the requirements we care for. They certainly aren't. But, this text cannot contain an infinite number of words, and we need to focus on something. Those, in my experience, are the most important ones, so we'll stick with them, at least for now.

Another thing we might need to note is that those requirements or, to be more precise, that those features are all interconnected. More often than not, one cannot be accomplished without the other or, in some other cases, one facilitates the other and makes it easier to accomplish.

Another thing worth noting is that we'll focus only on automation. For example, I know perfectly well that anything can be rolled back through human intervention. I know that we can extend our pipelines with post-

deployment tests followed with a rollback step in case they fail. As a matter of fact, anything can be done with enough time and manpower. But that's not what matters in this discussion. We'll ignore humans and focus only on the things that can be automated and be an integral part of deployment processes. I don't want you to scale your applications. I want the system to do it for you. I don't want you to roll back in case of a failure. I want the system to do that for you. I don't want you to waste your brain capacity on such trivial tasks. I wish you to spend your time on things that matter and leave the rest to machines.

After all that, we can summarize our requirements or features by saying that we'd like deployments to result in applications that are running and are:

- fault-tolerant
- highly available
- responsive
- rolling out progressively
- rolling back in case of a failure
- cost-effective

We'll remove *fault tolerance* from the future discussions since Kubernetes provides that out-of-the-box. As for the rest, we are yet to see whether we can accomplish them all and, if we can, whether a single deployment strategy will give us all those benefits.

There is a strong chance that there is no solution that will provide all those features. Even if we do find such a solution, the chances are that it might not be appropriate for your applications and their architecture. We'll worry about that later. For now, we'll explore some of the commonly used deployment strategies and see which of those requirements they fulfill.

Just as in any other chapter, we'll explore the subject in more depth through practical examples. For that, we need a working Jenkins X cluster as well as an application that we'll use it as a guinea pig on which we'll experiment with different deployment strategies.

## **Creating A Kubernetes Cluster With Jenkins X And Creating A Sample Application**

If you kept the cluster from the previous chapter, you can skip this section only if you were doubting my choice of VM sizes and make the nodes bigger than what I suggested. Otherwise, we'll need to create a new Jenkins X cluster.

We've been using the same cluster specifications for a while now. No matter the hosting vendor you chose in the past, if you created the cluster using my instructions, it is based on nodes with only 2 available CPUs or even less. We'll need more. Even if your cluster is set to autoscale, increasing the number of nodes will not help since one of the Istio components we'll use requires at least 2 CPUs available. Remember, even if you do have nodes with 2 CPUs, some computing power is reserved for system-level processes or Kubernetes daemons, so a 2 CPUs node does not result in 2 CPUs available.

We'll need to create a cluster with bigger nodes. The gists listed below will do just that. Those related to AKS, EKS, and GKE are now having nodes with 4 CPUs. If you are using your own cluster hosted somewhere else, the Gists are the same, and I will assume that the nodes have more than 2 available CPUs.

On top of all that, if you are using GKE, the gist now contains the command that installs **Gloo** which we explored in the previous chapter, and it sets the team deployment type to `knative`.



All the commands from this chapter are available in the [17-progressive-delivery.sh](#) Gist.

The new Gists are as follows.

- Create a new serverless **GKE** cluster with Gloo: [gke-jx-serverless-gloo.sh](#)
- Create a new serverless **EKS** cluster: [eks-jx-serverless.sh](#)
- Create a new serverless **AKS** cluster: [aks-jx-serverless.sh](#)
- Use an **existing** serverless cluster: [install-serverless.sh](#)

Now that we have a cluster with Jenkins X, we'll create a sample application.

```
1 jx create quickstart \
2   --filter golang-http \
3   --project-name jx-progressive \
4   --batch-mode
```

Now we can start exploring deployment strategies, with serverless being the first in line.

## Using Serverless Strategy With Gloo And Knative (GKE only)

Judging by the name of this section, you might be wondering why do we start with serverless deployments. The honest answer is that I did not try to put the deployment strategies in any order. We're starting with serverless simply because that is the one we used in the previous chapter. So, we'll start with what we have right now, at least for those who are running Jenkins X in GKE.

Another question you might be asking is why do we cover serverless with Knative in here given that we already discussed it in the previous chapter. The answer to that question lies in completeness. Serverless deployments are one of the essential options we have when choosing the strategy, and this chapter could not be complete without it. If you did go through the previous chapter, consider this one a refresher with a potential to find out something new. If nothing else, you'll get a better understanding of the flow of events with Knative as well as to see a few diagrams. In any case, the rest of the strategies will build on top of this one. On the other hand, you might be impatient and bored with repetition. If that's the case, feel free to skip this section altogether.



At the time of this writing (September 2019), serverless deployments with Knative work out-of-the-box only in GKE ([issue 4668](#)). That does not mean that Knative does not work in other Kubernetes flavors, but rather that Jenkins X installation of Knative works only in GKE. I encourage you to set up Knative yourself and follow along in your Kubernetes flavor. If you cannot run Knative, I still suggest you stick around even if you cannot run the examples. I'll do my best to be brief and to make the examples clear even for those not running them.

Instead of discussing the pros and cons first, we'll start each strategy with an example. We'll observe the results, and, based on that, comment on their advantages and disadvantages as well as the scenarios when they

might be a good fit. In that spirit, let's create a serverless deployment first and see what we'll get.

Let's go into the project directory and take a quick look at the definition that makes the application serverless.

```
1 cd jx-progressive  
2  
3 cat charts/jx-progressive/templates/ksvc.yaml
```

We won't go into details of Knative specification. It was briefly explained in the [Using Jenkins X To Define And Run Serverless Deployments](#) chapter and details can be found in the [official docs](#). What matters in the context of the current discussion is that the YAML you see in front of you defined a serverless deployment using Knative.

By now, if you created a new cluster, the application we imported should be up-and-running. But, to be on the safe side, we'll confirm that by taking a quick look at the *jx-progressive* activities.



There's no need to inspect the activities to confirm whether the build is finished if you are reusing the cluster from the previous chapter. The application we deployed previously should still be running.

```
1 jx get activities \  
2   --filter jx-progressive \  
3   --watch
```

Once you confirm that the build is finished press *ctrl+c* to stop watching the activities.

As you probably already know, the activity of an application does not wait until the release is promoted to the staging environment. So, we'll confirm that the build initiated by changes to the *environment-jx-rocks-staging* repository is finished as well.

```
1 jx get activities \  
2   --filter environment-jx-rocks-staging/master \  
3   --watch
```

Just as before, feel free to press *ctrl+c* once you confirm that the build was finished.

Finally, the last verification we'll do is to confirm that the Pods are running.

```
1 kubectl --namespace jx-staging \
2     get pods
```

The output is as follows.

```
1 NAME                  READY STATUS RESTARTS AGE
2 jx-jx-progressive-... 1/1   Running 0        45s
```

In your case, `jx-progressive` deployment might not be there. If that's the case, it's been a while since you used the application and Knative made the decision to scale it to zero replicas. We'll go through a scaling-to-zero example later. For now, imagine that you do have that Pod running.

On the first look, everything looks “normal”. It as if the application was deployed like any other. The only “strange” thing we can observe by looking at the Pods is the name of the one created through the `jx-progressive` Deployment and that it contains two containers instead of one. We'll ignore the “naming strangeness” and focus on the latter observation.

Knative injected a container into our Pod. It contains `queue-proxy` that, as the name suggests, serves as a proxy responsible for request queue parameters. It also reports metrics to the Autoscaler through which it might scale up or down depending on the number of different parameters. Requests are not going directly to our application but through this container.

Now, let's confirm that the application is indeed accessible from outside the cluster.

```
1 STAGING_ADDR=$(kubectl \
2     --namespace jx-staging \
3     get ksvc jx-progressive \
4     --output jsonpath=".status.url")
5
6 curl "$STAGING_ADDR"
```

We retrieved the address through which we can reach the application running in the staging environment, and we used `curl` to send a request. The output should be `hello, PR!` which is the message we defined in one of the previous chapters.

So far, the significant difference when compared with “normal” Kubernetes deployments is that the access to the application is not controlled through Ingress any more. Instead, it goes through a new resource type abbreviated as `ksvc` (short for Knative Service). Apart from that, everything else seems to be the same, except if we left the application unused for a while. If that’s the case, we still got the same output, but there was a slight delay between sending the request and receiving the response. The reason for such a delay lies in Knative’s scaling capabilities. It saw that the application is not used and scaled it to zero replicas. But, the moment we sent a request, it noticed that zero replicas is not the desired state and scaled it back to one replica. All in all, the request entered into a gateway (in our case served by Gloo Envoy) and waited there until a new replica was created and initialized, unless one was already running. After that, it forwarded the request to it, and the rest is the “standard” process of our application responding and that response being forwarded to us (back to `curl`).

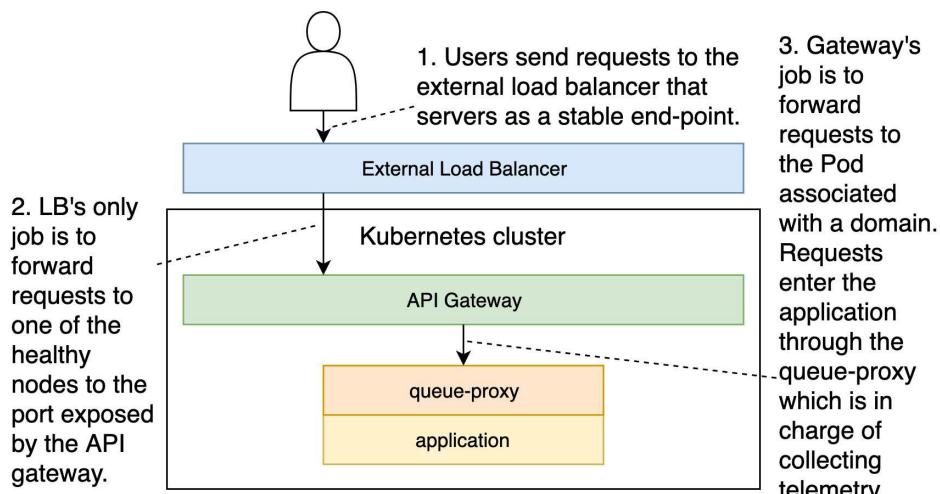


Figure 17-1: The flow of a request with API gateway

I cannot be sure whether your serverless deployment indeed scaled to zero or it didn’t. So, we’ll use a bit of patience to validate that it does indeed scale to nothing after a bit of inactivity. All we have to do is wait for five to ten minutes. Get a coffee or some snack.

```
1 kubectl \
2   --namespace jx-staging \
3   get pods
```

Assuming that sufficient time passed, the output should be as follows state that no resources were found in the namespace, unless you have other

applications there. The application is now gone. If we ignore other resources and focus only on Pods, it seems like the application is wiped out completely. That is true in terms that nothing application-specific is running. All that's left are a few Knative definitions and the common resources used for all applications (not specific to *jx-progressive*).



If you still see the *jx-progressive* Pod, all I can say is that you are impatient and you did not wait long enough. If that's what happened, wait for a while longer and repeat the `get pods` command.

Using telemetry collected from all the Pods deployed as Knative applications, Gloo detected that no requests were sent to *jx-progressive* for a while and decided that the time has come to scale it down. It sent a notification to Knative that executed a series of actions which resulted in our application being scaled to zero replicas.

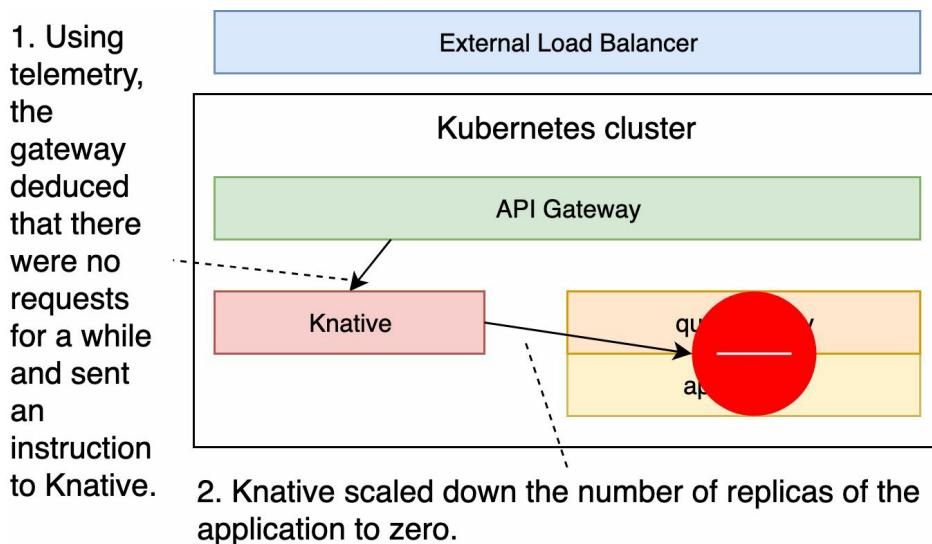


Figure 17-2: Knative's ability to scale to zero replicas

Bear in mind that the actual process is more complicated than that and that there are quite a few other components involved. Nevertheless, for the sake of brevity, the simplistic view we presented should suffice. I'll leave it up to you to go deeper into Gloo and Knative or accept it as magic. In either case, our application was successfully scaled to zero replicas. We started saving resources that could be better used by other applications and save us some costs in the process.

If you never used serverless deployments and if you never worked with Knative, you might think that your users would not be able to access it anymore since the application is not running. Or you might think that it will be scaled up once requests start coming in, but you might be scared that you'll lose those sent before the new replica starts running. Or you might have read the previous chapter and know that those fears are unfounded. In any case, we'll put that to the test by sending three hundred concurrent requests for twenty seconds.

```
1 kubectl run siege \
2   --image yokogawa/siege \
3   --generator "run-pod/v1" \
4   -it --rm \
5   -- --concurrent 300 --time 20S \
6   "$STAGING_ADDR" \
7   && kubectl \
8   --namespace jx-staging \
9   get pods
```

We won't go into details about Siege. Read the previous chapter if you want to know more. What matters is that we finished sending a lot of requests and that the previous command retrieved the Pods in the staging namespace. That output is as follows.

```
1 ...
2 NAME                               READY STATUS    RESTARTS AGE
3 jx-progressive-dzghl-deployment-... 2/2  Running  0          19s
4 jx-progressive-dzghl-deployment-... 2/2  Running  0          16s
5 jx-progressive-dzghl-deployment-... 2/2  Running  0          18s
6 jx-progressive-dzghl-deployment-... 2/2  Running  0          16s
```

Our application is up-and-running again. A few moments ago, the application was not running, and now it is. Not only that, but it was scaled to three replicas to accommodate the high number of concurrent requests.

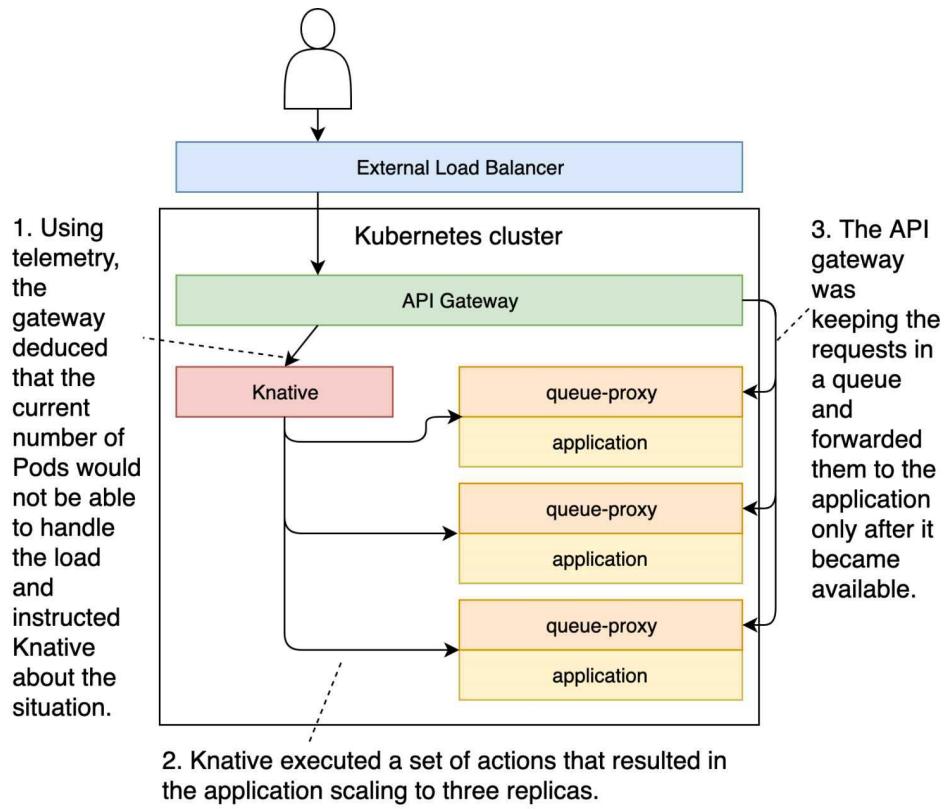


Figure 17-3: Knative’s ability to scale from zero to multiple replicas

What did we learn from serverless deployments in the context of our quest to find one that fits our needs the best?

High availability is easy in Kubernetes, as long as our applications are designed with that in mind. What that means is that our apps should be scalable and should not contain state. If they cannot be scaled, they cannot be highly available. When a replica fails (note that I did not say *if* but *when*), no matter how fast Kubernetes will reschedule it somewhere else, there will be downtime, unless other replicas take over its load. If there are no other replicas, we are bound to have downtime both due to failures but also whenever we deploy a new release. So, scalability (running more than one replica) is the prerequisite for high availability. At least, that’s what logic might make us think.

In the case of serverless deployments with Knative, not having replicas that can respond to user requests is not an issue, at least not from the high availability point of view. While in a “normal” situation, the requests would fail to receive a response, in our case, they were queued in the gateway and forwarded after the application is up-and-running. So, even if the application is scaled to zero replicas (if nothing is running), we are still

highly available. The major downside is in potential delays between receiving the first requests and until the first replica of the application is responsive.

The problem we might have with serverless deployments, at least when used in Kubernetes, is responsiveness. If we keep the default settings, our application will scale to zero if there are no incoming requests. As a result, when someone does send a request to our app, it might take longer than usual until the response is received. That could be a couple of milliseconds, a few seconds, or much longer. It all depends on the size of the container image, whether it is already cached on the node where the Pod is scheduled, the amount of time the application needs to initialize, and quite a few other criteria. If we do things right, that delay can be short. Still, any delay reduces the responsiveness of our application, no matter how short or long it is. What we need to do is compare the pros and cons. The results will differ from one app to another.

Let's take the static Jenkins as an example. In many organizations, it is under heavy usage throughout working hours, and with low or no usage at nights. We can say that half of the day it is not used. What that means is that we are paying double to our hosting vendor. We could have shut it down overnight and potentially remove a node from the cluster due to decreased resource usage. Even if the price is not an issue, surely those resources reserved by inactive Jenkins could be better used by some other processes. Shutting down the application would be an improvement, but it would also produce potentially very adverse effects.

What if someone is working overnight and pushes a change to Git. A webhook would fire trying to notify Jenkins that it should run a build. But, such webhook would fail if there is no Jenkins to handle the request. A build would never be executed. Unless we set up a policy that says “you are never allowed to work after 6 pm, even if the whole system crashed”, having a non-responsive system is unacceptable.

Another issue would be to figure out when is our system not in use. If we continue using the “traditional” Jenkins as an example, we could say that it should shut-down at 9 pm. If our official working hours end at 6 pm, that will provide three hours margin for those who do stay in the office longer. But, that would still be a suboptimal solution. During much of those three hours, Jenkins would not be used, and it would continue wasting resources.

On the other hand, there is still no guarantee that no one will ever push a change after 9 pm.

Knative solves those and quite a few other problems. Instead of shutting down our applications at predefined hours and hoping that no one is using them while they are unavailable, we can let Knative (together with Gloo or Istio) monitor requests. It would scale down if a certain period of inactivity passed. On the other hand, it would scale back up if a request is sent to it. Such requests would not be lost but queued until the application becomes available again.

All in all, I cannot say that Knative might result in non-responsiveness. What I can say is that it might produce slower responses in some cases (between having none and having some replicas). Such periodical slower responsiveness might produce less negative effect than the good it brings. Is it such a bad thing if static Jenkins takes an additional ten seconds to start building something after a whole night of inactivity? Even a minute or two of delay is not a big deal. On the other hand, in that particular case, the upside outweighs the downsides. Still, there are even better examples of the advantages of serverless deployments than Jenkins.

Preview environments might be the best example of wasted resources. Every time we create a pull request, a release is deployed into a temporary environment. That, by itself, is not a waste. The benefits of being able to test and review an application before merging it to master outweigh the fact that most of the time we are not using those applications. Nevertheless, we can do better. Just as we explained in the previous chapter, we can use Knative to deploy to preview environments, no matter whether we use it for permanent environments like staging and production. After all, preview environments are not meant to provide a place to test something before promoting it to production (staging does that). Instead, they provide us with relative certainty that what we'll merge to the master branch is likely code that works well.

If the response delay caused by scaling up from zero replicas is unacceptable in certain situations, we can still configure Knative to have one or more replicas as a minimum. In such a case, we'd still benefit from Knative capabilities. For example, the metrics it uses to decide when to scale might be easier or better than those provided by HorizontalPodAutoscaler (HPA). Nevertheless, the result of having Knative deployment with a minimum number of replicas above zero is

similar to the one we'd have with using HPA. So, we'll ignore such situations since our applications would not be serverless. That is not to say that Knative is not useful if it doesn't scale to zero. What it means is that we'll treat those situations separately and stick to serverless features in this section.

What's next in our list of deployment requirements?

Even though we did not demonstrate it through examples, serverless deployments with Knative do not produce downtime when deploying new releases. During the process, all new requests are handled by the new release. At the same time, the old ones are still available to process all those requests that were initiated before the new deployment started rolling out. Similarly, if we have health checks, it will stop the rollout if they fail. In that aspect, we can say that rollout is progressive.

On the other hand, it is not “true” progressive rollout but similar to those we get with rolling updates. Knative, by itself, cannot choose whether to continue progressing with a deployment based on arbitrary metrics. Similarly, it cannot roll back automatically if predefined criteria are met. Just like rolling updates, it will stop the rollout if health checks fail, and not much more. If those health checks fail with the first replica, even though there is no rollback, all the requests will continue being served with the old release. Still, there are too many ifs in those statements. We can only say that serverless deployments with Knative (without additional tooling) partially fulfills the progressive rollout requirement and that they are incapable of automated rollbacks.

Finally, the last requirement is that our deployment strategy should be cost-effective. Serverless deployments, at least those implemented with Knative, are probably the most cost-effective deployments we can have. Unlike vendor-specific serverless implementations like AWS Lambda, Azure Functions, and Google Cloud’s serverless platform, we are in (almost) full control. We can define how many requests are served by a single replica. We control the size of our applications given that anything that can run in a container can be serverless (but is not necessarily a good candidate). We control which metrics are used to make decisions and what are the thresholds. Truth be told, that is likely more complicated than using vendor-specific serverless implementations. It's up to us to decide whether additional complications with Knative outweigh the benefits it brings. I'll leave such a decision in your hands.

So, what did we conclude? Do serverless deployments with Knative fulfill all our requirements? The answer to that question is a resounding “no”. No deployment strategy is perfect. Serverless deployments provide **huge benefits** with **high-availability** and **cost-effectiveness**. They are **relatively responsive** and offer a certain level of progressive rollouts. The major drawback is the **lack of automated rollbacks**.

Requirement	Fullfilled
High-availability	Fully
Responsiveness	Partly
Progressive rollout	Partly
Rollback	Not
Cost-effectiveness	Fully

Please note that we used Gloo in conjunction with Knative to perform serverless deployments. We could have used Istio instead of Gloo. Similarly, we could have used OpenFaaS instead of Knative. Or we could have opted for something completely different. There are many different solutions we could assemble to make our applications serverless. Still, the goal was not to compare them all and choose the best one. Instead, we explored serverless deployments in general as one possible strategy we could employ. I do believe that Knative is the most promising one, but we are still in early stages with serverless in general and especially in Kubernetes. It would be impossible to be sure of what will prevail. Similarly, for many engineers, Istio would be the service mesh of choice due to its high popularity. I chose Gloo mostly because of its simplicity and its small footprint. For those of you who prefer Istio, all I can say is that we will use it for different purposes later on in this chapter.

Finally, I decided to present only one serverless implementation mostly because it would take much more than a single chapter to compare all those that are popular. The same can be said for service mesh (Gloo). Both are fascinating subjects that I might explore in the next book. But, at this moment I cannot make that promise because I do not plan a new book before the one I’m writing (this one) is finished.

What matters is that we’re finished with a very high-level exploration of the pros and cons of using serverless deployments and now we can move

into the next one. But, before we do that, we'll revert our chart to the good old Kubernetes Deployment.

```
1 jx edit deploy \
2   --kind default \
3   --batch-mode
4
5 cat charts/jx-progressive/values.yaml \
6   | grep knative
```

We edited the deployment strategy by setting it to `default` (it was `knative` so far). Also, we output the `knative` variable to confirm that it is now set to `false`.

The last thing we'll do is go out of the local copy of the *jx-progressive* directory. That way we'll be in the same place as those who could not follow the examples because their cluster cannot yet run Knative or those who were too lazy to set it up.

```
1 cd ..
```

## Using Recreate Strategy With Standard Kubernetes Deployments

*A long time ago in a galaxy far, far away,* most of the applications were deployed with what today we call the *recreate* strategy. We'll discuss it shortly. For now, we'll focus on implementing it and observing the outcome.

By default, Kubernetes Deployments use the `RollingUpdate` strategy. If we do not specify any, that's the one that is implied. We'll get to that one later. For now, what we need to do is add the `strategy` into the `deployment.yaml` file that defines the Deployment.

```
1 cd jx-progressive
2
3 cat charts/jx-progressive/values.yaml \
4   | sed -e \
5     's@replicaCount: 1@replicaCount: 3@g' \
6   | tee charts/jx-progressive/values.yaml
7
8 cat charts/jx-progressive/templates/deployment.yaml \
9   | sed -e \
10    's@ replicas:@  strategy:@\n11      type: Recreate@\n12      replicas:@g' \
13   | tee charts/jx-progressive/templates/deployment.yaml
```

We entered the local copy of the *jx-progressive* repository, and we used a bit of `sed` magic to increase the number of replicas in `values.yaml` and to add the `strategy` entry just above `replicas` in `deployment.yaml`. If you are not a `sed` ninja, that command might have been confusing, so let's output the file and see what we got.

```
1 cat charts/jx-progressive/templates/deployment.yaml
```

The output, limited to the relevant section, is as follows.

```
1 ...
2 spec:
3   strategy:
4     type: Recreate
5 ...
```

Now that we changed our deployment strategy to `recreate`, all that's left is to push it to GitHub, wait until it is deployed, and observe the outcome. Right?

```
1 git add .
2
3 git commit -m "Recreate strategy"
4
5 git push --set-upstream origin master
6
7 jx get activities \
8   --filter jx-progressive/master \
9   --watch
```

We pushed the changes, and we started watching the activities. Please press `ctrl+c` to cancel the watcher once you confirm that the newly launched build is finished.

If you're using serverless Jenkins X, the build of an application does not wait for the activity associated with automatic promotion to finish. So, we'll confirm whether that is done as well.



Please execute the command that follows only if you are using **serverless Jenkins X**.

```
1 jx get activities \
2   --filter environment-jx-rocks-staging/master \
3   --watch
```

You know what needs to be done. Press `ctrl+c` when the build is finished.

Let's take a look at the Pods we got.

```
1 kubectl --namespace jx-staging \
2   get pods
```

The output is as follows

```
1 NAME          READY STATUS RESTARTS AGE
2 jx-jx-progressive-... 1/1   Running 0      2m
3 jx-jx-progressive-... 1/1   Running 0      2m
4 jx-jx-progressive-... 1/1   Running 0      2m
```

There's nothing new here. Judging by the look of the Pods, if we did not change the strategy to Recreate, you would probably think that it is still the default one. The only difference we can notice is in the description of the Deployment, so let's output it.

```
1 kubectl --namespace jx-staging \
2   describe deployment jx-jx-progressive
```

The output, limited to the relevant part, is as follows.

```
1 ...
2 StrategyType:      Recreate
3 ...
4 Events:
5   Type    Reason           Age   From            Message
6   ----  -----  ----  ----
7 ...
8   Normal  ScalingReplicaSet 20s  deployment-controller  Scaled up replica set jx-prc
9   essive-589c47878f to 3
```

Judging by the output, we can confirm that the `StrategyType` is now `Recreate`. That's not a surprise. What is more interesting is the last entry in the `Events` section. It scaled replicas of the new release to three. Why is that a surprise? Isn't that the logical action when deploying the first release with the new strategy? Well... It is indeed logical for the first release so we'll have to create and deploy another to see what's really going on.

If you had Knative deployment running before, there is a small nuisance we need to fix. Ingress is missing, and I can prove that.

```
1 kubectl --namespace jx-staging \
2   get ing
```

The output claims that no resources were found.



Non-Knative users will have Ingress running and will not have to execute the workaround we are about to do. Feel free to skip the few commands that follow. Alternatively, you can run them as well. No harm will be done. Just remember that their purpose is to create the missing Ingress that is already running and that there will be no visible effect.

What happened? Why isn't there Ingress when we saw it countless times before in previous exercises?

Jenkins X creates Ingress resources automatically unless we tell it otherwise. You know that already. What you might not know is that there is a bug (undocumented feature) that prevents Ingress from being created the first time we change the deployment type from Knative to plain-old Kubernetes Deployments. That happens only when we switch and not in consecutive deployments of new releases. So, all we have to do is deploy a new release, and Jenkins X will pick it up correctly and create the missing Ingress resource the second time. Without it we won't be able to access the application from outside the cluster. So, all we have to do is make a trivial change and push it to GitHub. That will trigger yet another pipeline activity that will result in creation of a new release and its deployment to the staging environment.

```
1 echo "something" | tee README.md
2
3 git add .
4
5 git commit -m "Recreate strategy"
6
7 git push
```

We made a silly change, we pushed it to GitHub, and that triggered yet another build. All we have to do is wait. Or, even better, we can watch the activities of *jx-progressive* and the staging environment pipelines to confirm that everything was executed correctly. I'll skip showing you the `jx get activities` commands given that I'm sure you already know them by heart.

Assuming that you were patient enough and waited until the new release is deployed, now we can confirm that the Ingress was indeed created.

```
1 kubectl \
2   --namespace jx-staging \
3   get ing
```

The output is as follows.

```
1 NAME           HOSTS          ADDRESS      PORTS
2 jx-progressive jx-progressive.jx-staging.35.196.143.33.nip.io 35.196.143.33 80
3 56s
```

That's the same output that those that did not run Knative before saw after the first release. Now we are all on the same page.

All in all, the application is now running in staging, and it was deployed using the `recreate` strategy.

Next, we'll make yet another simple change to the code. This time we'll change the output message of the application. That will allow us to easily see how it behaves before and after the new release is deployed.

```
1 cat main.go | sed -e \
2   "s@example@recreate@g" \
3   | tee main.go
4
5 git add .
6
7 git commit -m "Recreate strategy"
8
9 git push
```

We changed the message. As a result, our current release is outputting `Hello from: Jenkins X golang http example`, while the new release, once it's deployed, will return `Hello from: Jenkins X golang http recreate`.

Now we need to be **very fast** and start sending requests to our application before the new release is rolled out. If you're unsure why we need to do that, it will become evident in a few moments.

Please open a **second terminal** window.

Given that **EKS** requires access key ID and secret access key as authentication, we'll need to declare a few environment variables in the new terminal session. Those are the same ones we used to create the cluster, so you shouldn't have any trouble recreating them.



Please execute the commands that follow **only** if your cluster is running in **EKS**. You'll have to replace the first [...] with your access key ID, and the second with the secret access key.

```
1 export AWS_ACCESS_KEY_ID=[...]
2
3 export AWS_SECRET_ACCESS_KEY=[...]
4
5 export AWS_DEFAULT_REGION=us-east-1
```

Let's find out the address of our application running in staging.

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

The output should be similar to the one that follows.

```
1 APPLICATION      STAGING PODS URL
2 jx-progressive 0.0.4    3/3  http://jx-progressive.jx-staging.35.196.143.33.nip.io
```

Copy the `jx-progressive` URL and paste it instead of [...] in the command that follows.

```
1 STAGING_ADDR=[...]
```

That's it. Now we can start bombing our application with requests.

```
1 while true
2 do
3   curl "$STAGING_ADDR"
4   sleep 0.2
5 done
```

We created an infinite loop inside which we're sending requests to the application running in staging. To avoid burning your laptop, we also added a short delay of 0.2 seconds.

If you were fast enough, the output should consist of an endless list of `Hello from: Jenkins X golang http example` messages. If that's what you're getting, it means that the deployment of the new release did not yet start. In such a case, all we have to do is wait.

At one moment, `Hello from: Jenkins X golang http example` messages will turn into 5xx responses. Our application is down. If this were a "real world" situation, our users would experience an outage. Some

of them might even be so disappointed that they would choose not to stick around to see whether we'll recuperate and instead switch to a competing product. I know that I, at least, have a very low tolerance threshold. If something does not work and I do not have a strong dependency on it, I move somewhere else almost instantly. If I'm committed to a service or an application, my tolerance might be a bit more forgiving, but it is not indefinite. I might forgive you one outage. I might even forgive two. But, the third time I cannot consume something, I will start considering an alternative. Then again, that's me, and your users might be more forgiving. Still, even if you do have loyal customers, downtime is not a good thing, and we should avoid it.

While you were reading the previous paragraph, the message probably changed again. Now it should be an endless loop of `Hello from: Jenkins X golang http recreate`. Our application recuperated and is now operational again. It's showing us the output from the new release. If we could erase from our memory the 5xx messages, that would be awesome.

All in all, the output, limited to the relevant parts, should be as follows.

```
1 ...
2 Hello from: Jenkins X golang http example
3 Hello from: Jenkins X golang http example
4 ...
5 <html>
6 <head><title>502 Bad Gateway</title></head>
7 <body>
8 <center><h1>502 Bad Gateway</h1></center>
9 <hr><center>openresty/1.15.8.1</center>
10 </body>
11 </html>
12 <html>
13 <head><title>503 Service Temporarily Unavailable</title></head>
14 <body>
15 <center><h1>503 Service Temporarily Unavailable</h1></center>
16 <hr><center>openresty/1.15.8.1</center>
17 </body>
18 </html>
19 ...
20 Hello from: Jenkins X golang http recreate
21 Hello from: Jenkins X golang http recreate
22 ...
```

If all you ever saw was only the loop of `Hello from: Jenkins X golang http recreate`, all I can say is that you were too slow. If that's the case, you'll have to trust me that there were some nasty messages in between the old and the new release.

That was enough looping for now. Please press `ctrl+c` to stop it and give your laptop a rest. Leave the second terminal open and go **back to the first one**.

What happened was neither pretty nor desirable. Even if you are not familiar with the `RollingUpdate` strategy (the default one for Kubernetes Deployments), you already experienced it countless times before. You probably did not see those `5xx` messages in the previous exercises, and that might make you wonder why did we switch to `Recreate`. Why would anyone want it? The answer to that question is that no one desires such outcomes, but many are having them anyway. I'll explain soon why we want to use the `Recreate` strategy even though it produces downtime. To answer why would anyone want something like that, we'll first explore why was the outage created in the first place.

When we deployed the second release using the `Recreate` strategy, Kubernetes first shut down all the instances of the old release. Only when they all ceased to work, it deployed the new release in its place. The downtime we experienced existed between the time the old release was shut down, and the time the new one became fully operational. The downtime lasted only for a couple of seconds, but that's because our application (`go-demo-6`) boots up very fast. Some other apps might be much slower, and the downtime would be much longer. It's not uncommon for the downtime in such cases to take minutes and sometimes even hours.

Alternatively, you might not have seen a single `5xx` error. If that's the case, you were fortunate because that means that the old release was shut down and the new was up-and-running within 200 milliseconds (iteration between two requests in the loop)<sup>9</sup>. If that's what happened to you, rest assured that it is highly unlikely it'll happen again. You just experienced once in a lifetime event. As you can probably guess, we cannot rely on users being that lucky.

We can think of the `Recreate` strategy as a “big bang”. There is no transition period, there are no rolling updates, nor there are any other “modern” deployment practices. The old release is shut down, and the new one is put in its place. It's simple and straightforward, but it results in inevitable downtime.

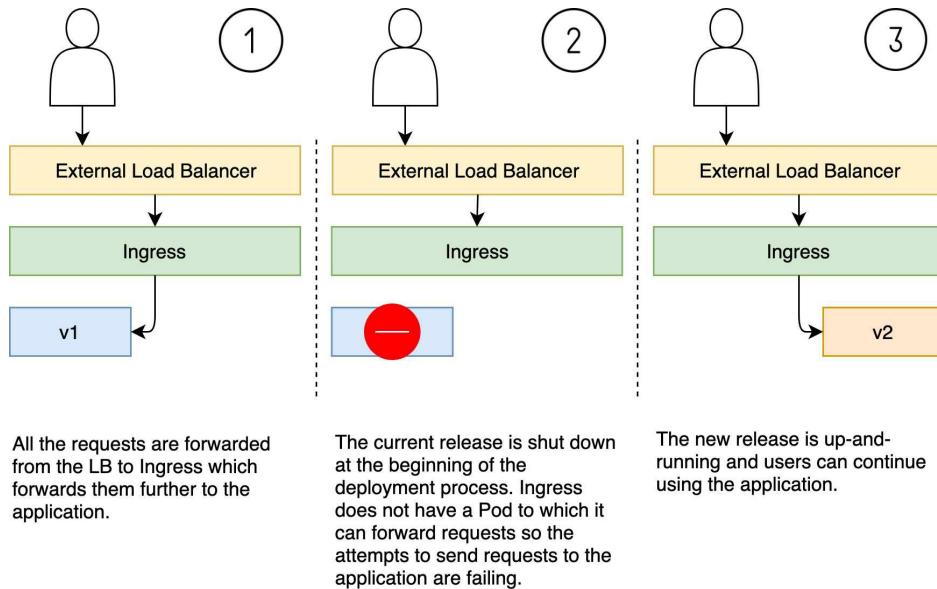


Figure 17-4: The recreate deployments strategy

Still, the initial question stands. Who would ever want to use the Recreate strategy? The answer is not that much who wants it, but rather who must use it.

Let's take another look at static Jenkins. It is a stateful application that cannot scale. So, replacing one replica at a time as a way to avoid downtime is out of the question. When applications cannot scale, there is no way we could ever accomplish deployments without downtime. Two replicas are a minimum. Otherwise, if only one replica is allowed to run at any given moment, we have to shut it down to make room for the other replica (the one from the new release). So, when there is no scaling, there is no high availability. Downtime, at least related to new releases, is unavoidable.

Why can static Jenkins not scale? There can be many answers to that question, but the main culprit is its state. It is a stateful application unable to share that state across multiple instances. Even if you deploy various Jenkins instances, they will operate independently from each other. Each would have a different state and manage different pipelines. That, dear reader, is not scaling. Having multiple independent instances of an application is not replication. For an application to be scalable, each replica needs to work together with others and share the load. As a rule of thumb, for an application to be scalable, it needs to be stateless (e.g., *go-demo-6*) or to be able to replicate state across all replicas (e.g., MongoDB). Jenkins does not fulfill either of the two criteria and, therefore, it cannot

scale. Each instance has its separate file storage where it keeps the state unique to that instance. The best we can do with static Jenkins is to give an instance to each team. That solves quite a few Jenkins-specific problems, but it does not make it scalable. As a result, it is impossible to upgrade Jenkins without downtime.

Upgrades are not the only source of downtime with unscalable applications. If we have only one replica, Kubernetes will recreate it when it fails. But that will also result in downtime. As a matter of fact, failure and upgrades of single-replica applications are more or less the same processes. In both cases, the only replica is shut down, and a new one is put in its place. There is downtime between those two actions.

All that might lead you to conclude that only single-replica applications should use the `Recreate` strategy. That's not true. There are many other reasons while the “big bang” deployment strategy should be applied. We won't have time to discuss all. Instead, I'll mention only one more example.

The only way to avoid downtime when upgrading applications is to run multiple replicas and start replacing them one by one or in batches. It does not matter much how many replicas we shut down and replace with those of the new release. We should be able to avoid downtime as long as there is at least one replica running. So, we are likely going to run new releases in parallel with the old ones, at least for a while. We'll go through that scenario soon. For now, trust me when I say that running multiple releases of an application in parallel is unavoidable if we are to perform deployments without downtime. That means that our releases must be backward compatible, that our applications need to version APIs, and that clients need to take that versioning into account when working with our apps. Backward compatibility is usually the main stumbling block that prevents teams from applying zero-downtime deployments. It extends everywhere. Database schemas, APIs, clients, and many other components need to be backward compatible.

All in all, inability to scale, statefulness, lack of backward compatibility, and quite a few other things might prevent us from running two releases in parallel. As a result, we are forced to use the `Recreate` strategy or something similar.

So, the real question is not whether anyone wants to use the `Recreate` strategy, but rather who is forced to apply it due to the problems usually related to the architecture of an application. If you have a stateful application, the chances are that you have to use that strategy. Similarly, if your application cannot scale, you are probably forced to use it as well.

Given that deployment with the `Recreate` strategy inevitably produces downtime, most teams tend to have less frequent releases. The impact of, let's say, one minute of downtime is not that big if we produce it only a couple of times a year. But, if we would increase the release frequency, that negative impact would increase as well. Having downtime a couple of times a year is much better than once a month, which is still better than if we'd have it once a day. High-velocity iterations are out of the question. We cannot deploy releases frequently if we experience downtime each time we do that. In other words, zero-downtime deployments are a prerequisite for high-frequency releases to production. Given that the `Recreate` strategy does produce downtime, it stands to reason that it fosters less frequent releases to production as a way to reduce the impact of downtime.

Before we proceed, it might be important to note that there was no particular reason to use the `Recreate` deployment strategy. The *jx-progressive* application is scalable, stateless, and it is designed to be backward compatible. Any other strategy would be better suited given that zero-downtime deployment is probably the most important requirement we can have. We used the `Recreate` strategy only to demonstrate how that deployment type works and to be consistent with other examples in this chapter.

Now that we saw how the `Recreate` strategy works, let's see which requirements it fulfills, and which it fails to address. As you can probably guess, what follows is not going to be a pretty picture.

When there is downtime, there is no high-availability. One excludes the other, so we failed with that one.

Is our application responsive? If we used an application that is more appropriate for that type of deployment, we would probably discover that it would not be responsive or that it would be using more resources than it needs. Likely we'd experience both side effects.

If we go back to static Jenkins as an excellent example for the `Recreate` deployment strategy, we would quickly discover that it is expensive to run it. Now, I do not mean expensive in terms of licensing costs but rather in resource usage. We'd need to set it up to always use memory and CPU required for its peak load. We'd probably take a look at metrics and try to figure out how much memory and CPU it uses when the most concurrent builds are running. Then, we'd increase those values to be on the safe side and set them as requested resources. That would mean that we'd use more CPU and memory than what is required for the peak load, even if most of the time we need much less. In the case of some other applications, we'd let them scale up and down and, in that way, balance the load while using only the resources they need. But, if that would be possible with Jenkins, we would not use the `Recreate` strategy. Instead, we'd have to waste resources to be on the safe side, knowing that it can handle any load. That's very costly. The alternative would be to be cheaper and give it fewer resources than the peak load. However, in that case, it would not be responsive given that the builds at the peak load would need to be queued. Or, even worse, it would just bleed out and fail under a lot of pressure. In any case, a typical application used with the `Recreate` deployment strategy is often unresponsive, or it is expensive. More often than not, it is both.

The only thing left is to see whether the `Recreate` strategy allows progressive rollout and automated rollbacks. In both cases, the answer is a resounding no. Given that most of the time only one replica is allowed to run, progressive rollout is impossible. On the other hand, there is no mechanism to roll back in case of a failure. That is not to say that it is not possible to do that, but that it is not incorporated into the deployment process itself. We'd need to modify our pipelines to accomplish that. Given that we're focused only on deployments, we can say that rollbacks are not available.

What's the score? Does the `Recreate` strategy fulfill all our requirements? The answer to that question is a huge “no”. Did we manage to get at least one of the requirements? The answer is still no. “Big bang” deployments **do not provide high-availability**. They are **not cost-effective**. They are **rarely responsive**. There is **no possibility to perform progressive rollouts**, and they come with **no automated rollbacks**.

The summary of the fulfillment of our requirements for the `Recreate` deployment strategy is as follows.

<b>Requirement</b>	<b>Fulfilled</b>
High-availability	Not
Responsiveness	Not
Progressive rollout	Not
Rollback	Not
Cost-effectiveness	Not

As you can see, that was a very depressing outcome. Still, the architecture of our applications often forces us to apply it. We need to learn to live with it, at least until the day we are allowed to redesign those applications or throw them to thrash and start over.

I hope that you never worked with such applications. If you didn't, you are either very young, or you always worked in awesome companies. I, for example, spent most of my career with applications that had to be put down for hours every time we deploy a new release. I had to come to the office during weekends because that's then the least number of people were using our applications. I had to spend hours or even days doing deployments. I spent too many nights sleeping in the office over weekends. Luckily, we had only a few releases a year. Those days now feel like a nightmare that I never want to experience again. That might be the reason why I got interested in automation and architecture. I wanted to make sure that I replace myself with scripts.

So far, we saw two deployment strategies. We probably started with the inverted order, at least from the historical perspective. We can say that serverless deployments are one of the most advanced and modern strategies. At the same time, `Recreate` or, to use a better name, "big bang" deployments are the ghosts of the past that are still haunting us. It's no wonder that Kubernetes does not use it as a default deployment type.

From here on, the situation can only be more positive. Brace yourself for an increased level of happiness.

## **Using RollingUpdate Strategy With Standard Kubernetes Deployments**

We explored one of the only two strategies we can use with Kubernetes Deployment resource. As we saw, the non-default `Recreate` is meant to serve legacy applications that are typically stateful and often do not scale.

Next, we'll see what the Kubernetes community thinks is the default way we should deploy our software.



Please bear in mind that, both in the previous and in this section, we are focused on what Kubernetes Deployments offer. We could have just as well used StatefulSet for stateful applications or DaemonSet for those that should be running in each node of the cluster. However, even though those behave differently, they are still based on similar principles. We'll ignore those and focus only on Kubernetes Deployment resource, given that I do not want to convert this chapter into a neverending flow of rambling. Later on, we'll go yet again outside of what Kubernetes offers out-of-the-box.

Now, let's get back to the topic.

To make our Deployment use the `RollingUpdate` strategy, we can either remove the whole `strategy` entry given that is the default, or we can change the type. We'll go with the latter since the command to accomplish that is easier.

```
1 cat charts/jx-progressive/templates/deployment.yaml \
2     | sed -e \
3     's@type: Recreate@type: RollingUpdate@g' \
4     | tee charts/jx-progressive/templates/deployment.yaml
```

All we did was to change the `strategy.type` to `RollingUpdate`. You should see the full definition of the Deployment on the screen.

Next, we'll change the application's return message so that we can track the change easily from one release to the other.

```
1 cat main.go | sed -e \
2     "s@recreate@rolling update@g" \
3     | tee main.go
4
5 git add .
6
7 git commit -m "Recreate strategy"
8
9 git push
```

We made the changes and pushed them to the GitHub repository. Now, all that's left is to execute another loop. We'll keep sending requests to the application and display the output.



Please go to the **second terminal** before executing the command that follows.

```
1 while true
2 do
3     curl "$STAGING_ADDR"
4     sleep 0.2
5 done
```

The output should be a long list of Hello from: Jenkins X golang http recreate messages. After a while, when the new release is deployed, it will suddenly switch to Hello from: Jenkins X golang http rolling update!. The relevant part of the output should be as follows.

```
1 ...
2 Hello from: Jenkins X golang http recreate
3 Hello from: Jenkins X golang http recreate
4 Hello from: Jenkins X golang http rolling update!
5 Hello from: Jenkins X golang http rolling update!
6 ...
```

As you can see, this time, there was no downtime. The application switched from one release to another, or so it seems. But, if that's what happened, we would have seen some downtime, unless that switch happened exactly in those 0.2 seconds between the two requests. To understand better what happened, we'll describe the deployment and explore its events.



Please stop the loop with *ctrl+c* and return to the **first terminal**.

```
1 kubectl --namespace jx-staging \
2     describe deployment jx-jx-progressive
```

The output, limited to the events section, is as follows.

```
1 ...
2 Events:
3   Type      Reason          Age      From            Message
4   ----      -----          ----      ----
5 ...
6   Normal    ScalingReplicaSet 6m24s  deployment-controller  Scaled down replica set
7 progressive-8b5698864 to 0
8   Normal    ScalingReplicaSet 6m17s  deployment-controller  Scaled up replica set jx
9 ogressive-77b6455c87 to 3
10  Normal   ScalingReplicaSet  80s   deployment-controller  Scaled up replica set jx
```

```

11 ogressive-658f88478b to 1
12 Normal ScalingReplicaSet 80s deployment-controller Scaled down replica set
13 progressive-77b6455c87 to 2
14 Normal ScalingReplicaSet 80s deployment-controller Scaled up replica set jx
15 ogressive-658f88478b to 2
16 Normal ScalingReplicaSet 72s deployment-controller Scaled down replica set
17 progressive-77b6455c87 to 1
18 Normal ScalingReplicaSet 70s deployment-controller Scaled up replica set jx
19 ogressive-658f88478b to 3
20 Normal ScalingReplicaSet 69s deployment-controller Scaled down replica set
21 progressive-77b6455c87 to 0

```

From those events, we can see what happened to the Deployment so far. The first entry in my output (the one that happened over 6 minutes ago) we can see that it scaled one replica set to 0 and the other to 3. That was the rollout of the new release we created when we used the `Recreate` strategy. Everything was shut down before the new release was put in its place. That was the cause of downtime.

Now, with the `RollingUpdate` strategy, we can see that the system was gradually increasing replicas of one ReplicaSet (`jx-progressive-658f88478b`) and decreasing the other (`jx-progressive-77b6455c87`). As a result, instead of having “big bang” deployment, the system was gradually replacing the old release with the new one, one replica at the time. That means that there was not a single moment without one or the other release available and, during a brief period, both were running in parallel.

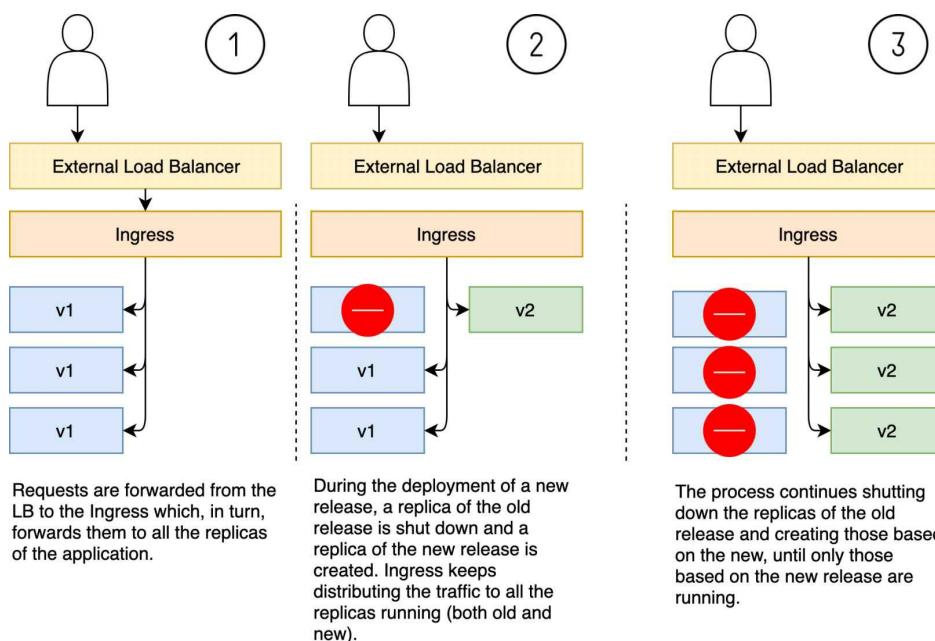


Figure 17-5: The `RollingUpdate` deployment strategy

You saw from the output of the loop that the messages switched from the old to the new release. In “real world” scenarios, you are likely going to have mixed outputs from both releases. For that reason, it is paramount that releases are backward compatible.

Let’s take a database as an example. If we updated schema before initiating the deployment of the application, we could assume that for some time both releases would use the new schema. If the change is not backward compatible, we could end up in a situation where some requests fail because the old release running in parallel with the new is incapable of operating with the new schema. If that were to happen, the result would be similar as if we used the `Recreate` strategy. Some requests would fail. Or, even worse, everything might seem to be working correctly from the end-user point of view, but we would end up with inconsistent data. That could be even worse than downtime.

There are quite a few other things that could go wrong with `RollingUpdates`, but most of them can be resolved by answering positively to two crucial questions. Is our application scalable? Are our releases backward compatible? Without scaling (multiple replicas), `RollingUpdate` is impossible, and without backward compatibility, we can expect errors caused by serving requests through multiple versions of our software.

So, what did we learn so far? Which requirements did we fulfill with the `RollingUpdate` strategy?

Our application was highly available at all times. By running multiple replicas, we are safe from downtime that could be caused by one or more of them failing. Similarly, by gradually rolling out new releases, we are avoiding downtime that we experienced with the `Recreate` strategy.

Even though we did not use `HorizontalPodAutoscaler` (HPA) in our example, we should add it to our solution. With it, we can make our application scale up and down to meet the changes in traffic. The effect would be similar as if we’d use serverless deployments (e.g., with Knative). Still, since HPA does not scale to zero replicas, it would be even more responsive given that there would be no response delay while the system is going from nothing to something (from zero replicas to whatever is needed). On the other hand, this approach comes at a higher cost. We’d have to run at least one replica even if our application is receiving no

traffic. Also, some might argue that setting up HPA might be more complicated given that Knative comes with some sensible scaling defaults. That might or might not be an issue, depending on the knowledge one has with deployments and Kubernetes in general. While with Knative HPA and quite a few other resources are implied, with Deployments and the `RollingUpdate` strategy, we do need to define it ourselves. We can say that Knative is more developer-friendly given its simpler syntax and that there is less need to change the defaults.

The only two requirements left to explore are progressive rollout and rollback.

Just as with serverless deployments, `RollingUpdate` kind of works. As you already saw, it does roll out replicas of the new release progressively, one or more at the time. However, the best we can do is make it stop the progress based on very limiting health checks. We can do much better on this front and later we'll see how.

Rollback feature does not exist with the `RollingUpdate` strategy. It can, however, stop rolling forward and that, in some cases, we might end up with only one non-functional replica of the new release. From the user's perspective, that might seem like only the old release is running. But there is no guarantee for such behavior given that in many occasions a problem might be detected after the second, third, or some other replica is rolled out. Automated rollbacks are the only requirement that wasn't fulfilled by any of the deployment strategies we employed so far. Bear in mind that, just as before, by automated rollback, I'm referring to what deployments offer us. I'm excluding situations in which you would do them inside your Jenkins X pipelines. Anything can be rolled back with a few tests and scripts executed if they fail, but that's not our current goal.

So, what did we conclude? Do rolling updates fulfill all our requirements? Just as with other deployment strategies, the answer is still "no". Still, `RollingUpdate` is much better than what we experienced with the `Recreate` strategy. Rolling updates provide **high-availability** and **responsiveness**. They are getting us **half-way towards progressive rollouts**, and they are **more or less cost-effective**. The major drawback is the **lack of automated rollbacks**.

The summary of the fulfillment of our requirements for the `RollingUpdate` deployment strategy is as follows.

<b>Requirement</b>	<b>Fulfilled</b>
High-availability	Fully
Responsiveness	Fully
Progressive rollout	Partly
Rollback	Not
Cost-effectiveness	Partly

The next in line is blue-green deployment.

## Evaluating Whether Blue-Green Deployments Are Useful

Blue-green deployment is probably the most commonly mentioned “modern” deployment strategy. It was made known by Martin Fowler.

The idea behind blue-green deployments is to run two production releases in parallel. If, for example, the current release is called “blue”, the new one would be called “green”, and vice versa. Assuming that the load balancer (LB) is currently pointing to the blue release, all we’d need to do to start redirecting users to the new one would be to change the LB to point to green.

We can describe the process through three different stages.

1. Let’s say that, right now, all the requests are forwarded to the current release. Let’s say that that’s the blue release with the version v2. We’ll also imagine that the release before it is running as green and that the version is v1. The green release lays dormant, mostly wasting resources.
2. When we decide to deploy a new release, we do it by replacing the inactive (dormant) instances. In our case, that would be green instances which are currently running v1 and will be replaced with v3.
3. When all the green instances are running the new release, all that’s left is to reconfigure the LB to forward all the requests to green instead of the blue instances. Now the blue release is dormant (unused) and will be the target of the next deployment.

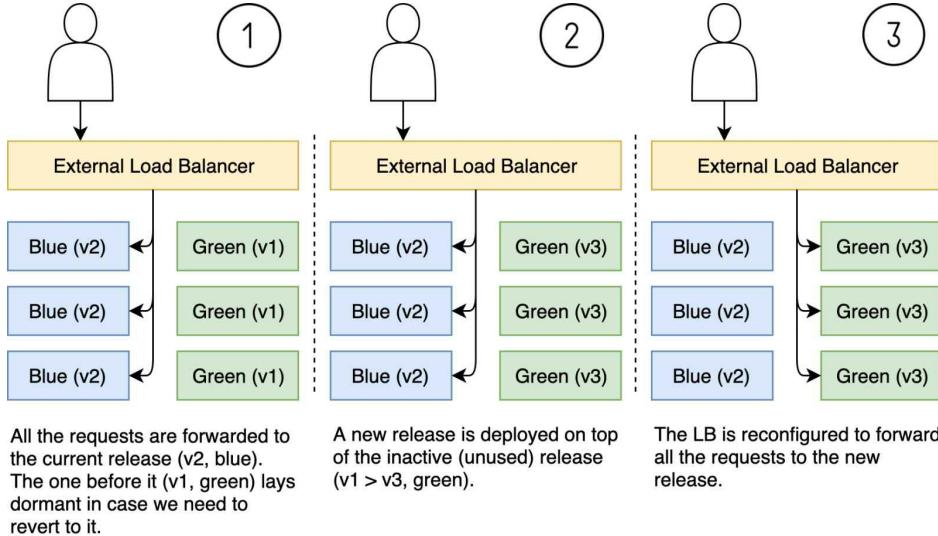


Figure 17-6: Blue-green deployment strategy

If we'd like to revert the release, all we'd have to do is change the LB to point from the active to the inactive set of instances. Or, to use different terminology, we switch it one color to another.

Blue-green deployments made a lot of sense before. If each of the replicas of our application were running in a separate VM, rolling updates would be much harder to accomplish. On top of that, rolling back (manual or automated) is indeed relatively easy with blue-green given that both releases are running in parallel. All we'd have to do is to reconfigure the LB to point to the old release.

However, we do not live in the past. We are not deploying binaries to VMs but containers to Kubernetes which schedules them inside virtual machines that constitute the cluster. Running any release is easy and fast. Rolling back containers is as easy as reconfiguring the LB.

When compared to the serverless deployments and the `RollingUpdate` strategy, blue-green does not bring anything new to the table. In all the cases, multiple replicas are running in parallel, even though that might not be that obvious with blue-green deployments.

People tend to think that switching from blue to green deployment is instant, but that is far from the truth. The moment we change the load balancer to point to the new release, both are being accessed by users, at least for a while. The requests initiated before the switch will continue being processed by the old release, and the new ones will be handled by

the new. In that aspect, blue-green deployments are no different from the `RollingUpdate` strategy. The significant difference is in the cost.

Let's imagine that we have five replicas of our application. If we're using rolling updates, during the deployment process, we will have six, unless we configure it differently. A replica of the new release is created so we have six replicas. After that, a replica of the old release is destroyed, so we're back to five. And so on, the process keeps alternating between five and six replicas, until the whole new release is rolled out and the old one is destroyed. With blue-green deployments, the number of replicas is duplicated. If we keep with five replicas as the example, during the deployment process, we'd have ten (five of the old and five of the new). As you can imagine, the increase in resource consumption is much lower if we increase the number of replicas by one than if we double them. Now, you can say that the increase is not that big given that it lasts only for the duration of the deployment process, but that would not necessarily be true.

One of the cornerstones of blue-green strategy is the ability to roll back by reconfiguring the load balancer to point to the old release. For that, we need the old release to be always up-and-running, and thus have the whole infrastructure requirements doubled permanently. Now, I do not believe that's the reason good enough today. Replacing a container based on one image with a container based on another is almost instant. Instead of running two releases in parallel, we can just as easily and rapidly roll forward to the old release. Today, running two releases in parallel forever and ever is just a waste of resources for no good reason.

Such a waste of resources (for no good reason) is even more evident if we're dealing with a large scale. Imagine that your application requires hundreds of CPU and hundreds of gigabytes of memory. Do you want to double that knowing that rolling updates give you all the same benefits without such a high cost associated with it?

Frankly, I think that blue-green was a short blink in the history of deployments. They were instrumental, and they provided the base from which others like rolling updates and canaries were born. Both are much better implementations of the same objectives. Nevertheless, blue-green is so popular that there is a high chance that you will not listen to me and that you want it anyways. I will not show you how to do "strict" blue-green deployments. Instead, I will argue that you were already doing a variation of it through quite a few chapters. I will assume that you want to deploy to

production what you already tested so no new builds of binaries should be involved. I will also expect that you do understand that there is no need to keep the old release running so that you can roll back to it. Finally, I will assume that you do not want to have any downtime during the deployment process. With all that in mind, let's do a variation of blue-green deployments without actually employing any new strategy or using any additional tools.

Now, let's take a look at what's running in the staging environment.

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

The output is as follows.

```
1 APPLICATION      STAGING PODS URL
2 jx-progressive  0.0.7   3/3  http://jx-progressive.jx-staging...
```

For now, we'll assume that whatever is running in production is our blue release and that staging is green. At this moment, you can say that both releases should be running in production to qualify for blue-green deployments. If that's what's going through your brain, remember that "staging" is just the name. It is running in the same cluster as production unless you choose to run Jenkins X environments in different clusters. The only thing that makes the release in staging different from production (apart from different Namespaces) is that it might not be using the production database or to be connected with other applications in production, but to their equivalents in staging. Even that would be an exaggeration since you are (and should be) running in staging the same setup as production. The only difference should be that one has production candidates while the other is the "real" production. If that bothers you, you can easily change the configuration so that an application in staging is using the database in production, be connected with other applications in production, and whatever else you have there.

With the differences between production and staging out of the way, we can say that the application running in staging is the candidate to be deployed in production. We can just as easily call one blue (production) and the other one green (staging).

Now, what comes next will be a slight deviation behind the idea of blue-green deployments. Instead of changing the load balancer (in our case

Ingress) to point to the staging release (green), we'll promote it to production.



Please replace [...] with the version from the previous output.

```
1 VERSION=[...]
2
3 jx promote jx-progressive \
4   --version $VERSION \
5   --env production \
6   --batch-mode
```

After a while, the promotion will end, and the new (green) release will be running in production. All that's left, if you're running serverless Jenkins X, is to confirm that by watching the activities associated with the production repository.

```
1 jx get activities \
2   --filter environment-jx-rocks-production/master \
3   --watch
```

Please press *ctrl+c* to stop watching the activity once you confirm that the build initiated by the previous activity's push of the changes to the master branch.

Now we can take a look at the applications running in production.

```
1 jx get applications --env production
```

The output should not be a surprise since you already saw the promotion process quite a few times before. It shows that the release version we have in staging is now running in production and that it is accessible through a specific address.

Finally, we'll confirm that the release is indeed running correctly by sending a request.



Please replace [...] with the address of the release in production. You can get it from the output of the previous command.

```
1 PRODUCTION_ADDR=[...]
2
3 curl "$PRODUCTION_ADDR"
```

Surprise, surprise... The output is Hello from: Jenkins X golang http rolling update. If you got 503 Service Temporarily Unavailable, the application is not yet fully operational because you probably did not have anything running in production before the promotion. If that's the case, please wait for a few moments and re-run the `curl` command.

Was that blue-green deployment? It was, of sorts. We had a release (in staging) running in precisely the same way as if it would run in production. We had the opportunity to test it. We switched our production load from the old to the new release without downtime. The significant difference is that we used `RollingUpdate` for that, but that's not a bad thing. Quite the contrary.

What we did has many of the characteristics of blue-green deployments. On the other hand, we did not strictly follow the blue-green deployment process. We didn't because I believe that it is silly. Kubernetes opened quite a few new possibilities that make blue-green deployments obsolete, inefficient, and wasteful.

Did that make you mad? Are you disappointed that I bashed blue-green deployments? Did you hope to see examples of the “real” blue-green process? If that’s what’s going through your mind, the only thing I can say is to stick with me for a while longer. We’re about to explore progressive delivery and the tools we’ll explore can just as well be used to perform blue-green deployments. By the end of this chapter, all you’ll have to do is read a bit of documentation and change a few things in a YAML file. You’ll have “real” blue-green deployment. However, by the time you finish reading this chapter, especially what’s coming next, the chances are that you will discard blue-green as well.

Given that we did not execute “strict” blue-green process and that what we used is `RollingUpdate` combined with promotion from one environment to another, we will not discuss the pros and cons of the blue-green strategy. We will not have the table that evaluates which requirements we fulfilled. Instead, we’ll jump into progressive delivery as a way to try to address progressive rollout and automated rollbacks. Those are the only two requirements we did not yet obtain fully.

## About The World We Lived In

The necessity to test new releases before deploying them to production is as old as our industry. Over time, we developed elaborate processes aimed at ensuring that our releases are ready for production. We were testing them locally and deploying them to testing environments where we would test them more. When we were comfortable with the quality, we were deploying those releases to integration and pre-production environments for the final round of validations. Typically, the closer we were getting to releasing something to production, the more similar our environments were to production. That was a lengthy process that would last for months, sometimes even years.

Why are we moving our releases through different environments (e.g., servers or clusters)? The answer lies in the difficulties in maintaining production-like environments.

In the past, it took a lot of effort to manage environments, and the more they looked like production, the more work they required. Later on, we adopted configuration management tools like CFEngine, Chef, Puppet, Ansible, and quite a few others. They simplified the management of our environments. Still, we kept the practice of moving our software from one to another as if it was an abandoned child moving from one foster family to another. The main reason why configuration management tools did not solve many problems lies in a misunderstanding of the root cause of the problem. What made the management of environments challenging is not that we had many of them, nor that production-like clusters are complicated. Instead, the issue was in mutability. No matter how much effort we put in maintaining the state of our clusters, differences would pile up over time. We could not say that one environment is genuinely the same as the other. Without that guarantee, we could not claim that what was tested in one environment would work in another. The risk of experiencing failure after deploying to production was still too high.

Over time, we adopted immutability. We learned that things shouldn't be modified at runtime, but instead created anew whenever we need to update something. We started creating VM images that contained new releases and applying rolling updates that would gradually replace the old. But that was slow. It takes time to create a new VM image, and it takes time to instantiate them. There were many other problems with them, but this is neither time nor place to explore them all. What matters is that

immutability applied to the VM level brought quite a few improvements, but also some challenges. Our environments became stable, and it was easy to have as many production-like environments as we needed, even though that approach revealed quite a few other issues.

Then came containers that took immutability to the next level. They gave us the ability to say that something running in a laptop is the same as something running in a test environment that happens to behave in the same way as in production. Creating a container based on an image produces the same result no matter where it runs. That's not 100% true, but when compared to what we had in the past, containers bring us as close to repeatability as we can get today.

So, if containers provide a reasonable guarantee that a release will behave the same no matter the environment it runs in, we can safely say that if it works in staging, it should work in production. That is especially true if both environments are in the same cluster. In such a case, hardware, networking, storage, and other infrastructure components are the same, and the only difference is the Kubernetes Namespace something runs in. That should provide a reasonable guarantee that a release tested in staging should work correctly when promoted to production. Don't you agree?

Even if environments are just different Namespaces in the same cluster, and if our releases are immutable container images, there is still a reasonable chance that we will detect issues only after we promote releases to production. No matter how well our performance tests are, production load cannot be reliably replicated. No matter how good we became writing functional tests, real users are unpredictable, and that cannot be reflected in test automation. Tests look for errors we already know about, and we can't test what we don't know. I can go on and on about the differences between production and non-production environments. It all boils down to one having real users, and the other running simulations of what we think "real" people would do.

I'll assume that we agree that production with real users and non-production with I-hope-this-is-what-real-people-do type of simulations are not the same. We can conclude that the only final and definitive confirmation that a release is successful can come from observing how well it is received by "real" users while running in production. That leads us to the need to monitor our production systems and observe user behavior, error rates, response times, and a lot of other metrics. Based on

that data, we can conclude whether a new release is truly successful or not. We keep it if it is and we roll back if it isn't. Or, even better, we roll forward with improvements and bug fixes. That's where progressive delivery kicks in.

## A Short Introduction To Progressive Delivery

Progressive delivery is a term that includes a group of deployment strategies that try to avoid the pitfalls of the all-or-nothing approach. New versions being deployed do not replace existing versions but run in parallel for some time while receiving live production traffic. They are evaluated in terms of correctness and performance before the rollout is considered successful.

Progressive Delivery encompasses methodologies such as rolling updates, blue-green or canary deployments. We already used rolling updates for most of our deployments so you should be familiar with at least one flavor of progressive delivery. What is common to all of them is that monitoring and metrics are used to evaluate whether a new version is “safe” or needs to be rolled back. That's the part that our deployments were missing so far or, at least, did not do very well. Even though we did add tests that run during and after deployments to staging and production environments, they were not communicating findings to the deployment process. We did manage to have a system that can decide whether the deployment was successful or not, but we need more. We need a system that will run validations during the deployment process and let it decide whether to proceed, to halt, or to roll back. We should be able to roll out a release to a fraction of users, evaluate whether it works well and whether the users are finding it useful. If everything goes well, we should be able to continue extending the percentage of users affected by the new release. All in all, we should be able to roll out gradually, let's say ten percent at a time, run some tests and evaluate results, and, depending on the outcome, choose whether to proceed or to roll back.

To make progressive delivery easier to grasp, we should probably go through the high-level process followed for the three most commonly used flavors.

With rolling updates, not all the instances of our application are updated at the same time, but they are rolled out incrementally. If we have several replicas (containers, virtual machines, etc.) of our application, we would

update one at a time and check the metrics before updating the next. In case of issues, we would remove them from the pool and increase the number of instances running the previous version.

Blue-green deployments temporarily create a parallel duplicate set of our application with both the old and new version running at the same time. We would reconfigure a load balancer or a DNS to start routing all traffic to the new release. Both versions coexist until the new version is validated in production. In some cases, we keep the old release until it is replaced with the new. If there are problems with the new version, the load balancer or DNS is just pointed back to the previous version.

With canary deployments, new versions are deployed, and only a subset of users are directed to it using traffic rules in a load balancer or more advanced solutions like service mesh. Users of the new version can be chosen randomly as a percentage of the total users or using other criteria such as geographic location, headers, employees instead of general users, etc. The new version is evaluated in terms of correctness and performance and, if successful, more users are gradually directed to the new version. If there are issues with the new version or if it doesn't match the expected metrics, the traffic rules are updated to send all traffic back to the previous one.

Canary releases are very similar to rolling updates. In both cases, the new release is gradually rolled out to users. The significant difference is that canary deployments allow us more control over the process. They allow us to decide who sees the new release and who is using the old. They allow us to gradually extend the reach based on the outcome of validations and on evaluating metrics. There are quite a few other differences we'll explore in more detail later through practical examples. For now, what matters is that canary releases bring additional levers to continuous delivery.

**Progressive delivery makes it easier to adopt continuous delivery.** It reduces risks of new deployments by limiting the blast radius of any possible issues, known or unknown. It also provides automated ways to rollback to an existing working version. No matter how much we test a release before deploying it to production, we can never be entirely sure that it will work. Our tests could never fully simulate “real” users behavior. So, if being 100% certain that a release is valid and will be well received by our users is impossible, the best we can do is to provide a safety net in the form of gradual rollout to production that depends on

results of tests, evaluation of metrics, and observation how users receive the new release.

There are quite a few ways we can implement canary deployments, ranging from custom scripts to using ready-to-go tools that can facilitate the process. Given that we do not have time or space to evaluate all the tools we could use, we'll have to pick a combination.

We will explore how Jenkins X integrates with Flagger, Istio, and Prometheus which, when combined, can be used to facilitate canary deployments. Each will start by getting a small percentage of the traffic and analyzing metrics such as response errors and duration. If these metrics fit a predefined requirement, the deployment of the new release will continue, and more and more traffic will be forwarded to it until everything goes through the new release. If these metrics are not successful for any reason, our deployment will be rolled back and marked as a failure. To do all that, we'll start with a rapid overview of those tools. Just remember that what you'll read next is a high-level overview, not an in-depth description. A whole book can be written on Istio alone, and this chapter is already too big.

## A Quick Introduction To Istio, Prometheus, Flagger, And Grafana

[Istio](#) is a service mesh that runs on top of Kubernetes. Quickly after the project was created, it became one of the most commonly used in Kubernetes. It allows traffic management that enables us to control the flow of traffic and other advanced networking such as point to point security, policy enforcement, automated tracing, monitoring, and logging.

We could write a full book about Istio. Instead, we'll focus on the traffic shifting and metric gathering capabilities of Istio and how we can use those to enable Canary deployments.

We can configure Istio to expose metrics that can be pulled by specialized tools. [Prometheus](#) is a natural choice, given that it is the most commonly used application for pulling, storing, and querying metrics. Its format for defining metrics can be considered the de-facto standard in Kubernetes. It stores time-series data that can be queried using its query language PromQL. We just need to make sure that Istio and Prometheus are integrated.

[Flagger](#) is a project sponsored by WeaveWorks. It can use service mesh solutions like Istio, Linkerd, Gloo (which we used with Knative), and quite a few others. Together with a service mesh, we can use Flagger to automate deployments and rollback using a few different strategies. Even though the focus right now is canary deployments, you should be able to adapt the examples that follow into other strategies as well. To make things easier, Flagger even offers a Grafana dashboard to monitor the deployment progress.

[Grafana](#) provides a user interface that allows us to visualize metrics through dashboards. Just like the other tools we chose, it is probably the most commonly used among the solutions we can run inside our Kubernetes clusters.



Please note that we could have used Gloo instead of Istio, just as we did in the [Using Jenkins X To Define And Run Serverless Deployments](#) chapter. But, I thought that this would be an excellent opportunity to introduce you to Istio. Also, you should be aware that none of the tools are the focus of the book and that the main goal is to show you one possible implementation of canary deployments. Once you understand the logic behind it, you should be able to switch to whichever toolset you prefer.



This book is dedicated to continuous delivery with Jenkins X. All the other tools we use are chosen mostly to demonstrate integrations with Jenkins X. We are not providing an in-depth analysis of those tools beyond their usefulness to continuous delivery.

## Installing Istio, Prometheus, Flagger, And Grafana

We'll install all the tools we need as Jenkins X addons. They are an excellent way to install and integrate tools. However, addons might not provide you with all the options you can use to tweak those tools to your specific needs. Later on, once you adopt Jenkins X in production, you should evaluate whether you want to continue using the addons or you prefer to set up those tools in some other way. The latter might give you more freedom. For now, addons are the easiest way to set up what we need so we'll roll with them.

Let's start with Istio.

```
1 jx create addon istio
```



In some cases, the previous command may fail due to the order Helm applies CRD resources. If that happens, re-run the command again to fix the issue.



Istio is resource-heavy. It is the reason why we increased the size of the VMs that compose our cluster.

When installing Istio, a new Ingress gateway service is created. It is used for sending all the incoming traffic to services based on Istio rules (`VirtualServices`). That achieves a similar functionality as the one provided by Ingress. While Ingress has the advantage of being simple, it is also very limited in its capabilities. Istio, on the other hand, allows us to create advanced rules for incoming traffic, and that's what we'll need for canary deployments.

For now, we need to find the external IP address of the Istio Ingress gateway service. We can get it from the output of the `jx create addon istio` command. But, given that I don't like copying and pasting outputs, we'll use the commands that follow to retrieve the address and store it in environment variable `ISTIO_IP`.

Just as with Ingress, the way to retrieve the IP differs depending on whether you're using EKS or some other Kubernetes flavor.



Please run the command that follows only if you are **NOT** using EKS (e.g., GKE, AKS, etc.).

```
1 ISTIO_IP=$(kubectl \
2   --namespace istio-system \
3   get service istio-ingressgateway \
4   --output jsonpath='{.status.loadBalancer.ingress[0].ip}' )
```



Please run the commands that follow only if you are using **EKS**.

```
1 ISTIO_HOST=$(kubectl \
2   --namespace istio-system \
3   get service istio-ingressgateway \
4   --output jsonpath='{.status.loadBalancer.ingress[0].hostname}')
5
6 export ISTIO_IP=$(dig +short $ISTIO_HOST \
7   | tail -n 1)"
```

To be on the safe side, we'll output the environment variable to confirm that the IP does indeed look to be correct.

```
1 echo $ISTIO_IP
```

When we created the `istio` addon, Prometheus was installed alongside it, so the only tool left for us to add is Flagger. Later on, we'll see why I skipped Grafana.

```
1 jx create addon flagger
```

You will see from the output that Istio was enabled in the production namespace (`jx-production`, or whichever Namespace you're using). As a result, Istio will be injecting whatever it needs to the Pods running there, and Prometheus will pull traffic metrics generated by the resources in that Namespace. As you will soon see, metrics are the cornerstone of canary deployments. We can also see that it created an Istio gateway called `jx-gateway`. It will accept incoming external traffic, and we are yet to create Istio VirtualServices with the rules where to forward those requests. Flagger will do that for us later on.

Now, let's take a quick look at the Pods that were created through those two addons.

```
1 kubectl --namespace istio-system \
2   get pods
```

The output is as follows.

1 NAME	READY	STATUS	RESTARTS	AGE
2 flagger-5bdbccc7f4-...	1/1	Running	0	110s
3 flagger-grafana-...	1/1	Running	0	78s
4 istio-citadel-...	1/1	Running	0	3m22s
5 istio-galley-...	1/1	Running	0	4m46s

```

6 istio-ingressgateway-...    1/1   Running   0      4m40s
7 istio-init-crd-10-...     0/1   Completed  0      5m8s
8 istio-init-crd-11-...     0/1   Completed  0      5m7s
9 istio-pilot-...           2/2   Running   0      3m35s
10 istio-policy-...          2/2   Running   6      4m38s
11 istio-sidecar-injector-... 1/1   Running   0      3m14s
12 istio-telemetry-...       2/2   Running   6      4m38s
13 prometheus-...            1/1   Running   0      3m28s

```

We won't go into details of what each of those Pods does. I expect you to consult the documentation if you are curious. For now, we'll note that Flagger, Istio, and Prometheus Pods were created in the `istio-system` Namespace and that, by the look of it, they are all running. If any of those are in the pending state, you either need to increase the number of nodes in your cluster or none of the nodes is big enough to meet the demand of the requested resources. The former case should be solved with the Cluster Autoscaler if you have it running in your Kubernetes cluster. The latter, on the other hand, probably means that you did not follow the instructions to create a cluster with bigger VMs. In any case, the next step would be to describe the pending Pod, see the root cause of the issue, and act accordingly.

As we saw from the output of the `jx create addon flagger` command, the Namespace that holds our production releases is enabled to use Istio. Let's see how that enablement works.

```

1 kubectl describe namespace \
2     jx-production

```

The output, limited to the relevant parts, is as follows.

```

1 Name:          jx-production
2 Labels:        env=production
3             istio-injection=enabled
4             team=cd
5 ...

```

We can see that we got a new label `istio-injection=enabled`. That one tells Istio to inject the sidecar containers into our Pods. Without it, we'd need to perform additional manual steps, and you already know that's not something I like doing.

Whichever deployment strategy we use, it should be the same in all the permanent environments. Otherwise, we do not have parity between applications running in production and those running in environments meant to be used for testing (e.g., staging). The more similar, if not the

same, those environments are, the more confident we are to promote something to production. Otherwise, we can end up in a situation where someone could rightfully say that what was tested is not the same as what is being deployed to production.

So, let's take a look at the labels in the staging environment.

```
1 kubectl describe namespace \
2     jx-staging
```

The output, limited to the relevant parts, is as follows.

```
1 Name:          jx-staging
2 Labels:        env=staging
3             team=jx
4 ...
```

As we can see, the staging environment does not have the `istio-injection=enabled` label, so Istio will not inject sidecars and, as a result, it will not work there. Given that we already elaborated that staging and production should be as similar as possible, if not the same, we'll add the missing label so that Istio works in both.

```
1 kubectl label namespace jx-staging \
2     istio-injection=enabled \
3     --overwrite
```

Let's have another look at the staging environment to confirm that the label was indeed added correctly.

```
1 kubectl describe namespace \
2     jx-staging
```

The output, limited to the relevant parts, is as follows.

```
1 Name:          jx-staging
2 Labels:        env=staging
3             istio-injection=enabled
4             team=jx
5 ...
```

The `istio-injection=enabled` is there, and we can continue while knowing that whatever Istio will do in the staging environment will be the same as in production.

## Creating Canary Resources With Flagger

Let's say we want to deploy our new release only to 20% of the users and that we will monitor metrics for 30 seconds. During that period, we'll be validating whether the error rate is within a predefined threshold and whether the time it takes to process requests is within some limits. If everything seems right, we'll increase the percentage of users who can use our new release for another 20%, and continue monitoring metrics to decide whether to proceed. The process should repeat until the new release is rolled out to everyone, twenty percent at a time every thirty seconds.

Now that we have a general idea what we want to accomplish and that all the tools are set up, all that's missing is to create Flagger Canary definition. We'll create a new file `canary.yaml` in the `charts/jx-progressive/templates` directory.



There might already be a `canary.yaml` in the `buildpacks` if I (or someone else) made a pull request to add canary capabilities out-of-the-box. If that's the case, ignore it because I'll explain the one we are about to create from scratch. Later on, if canaries are indeed available in build packs, you should be able to use the knowledge from creating a canary deployment from scratch to fine-tune those available in build packs.

Please execute the command that follows to create a new Canary resource.

```
1 echo "{{- if .Values.canary.enable }}"
2 apiVersion: flagger.app/v1alpha2
3 kind: Canary
4 metadata:
5   name: {{ template \"fullname\" . }}
6 spec:
7   provider: {{.Values.canary.provider}}
8   targetRef:
9     apiVersion: apps/v1
10    kind: Deployment
11    name: {{ template \"fullname\" . }}
12   progressDeadlineSeconds: 60
13   service:
14     port: {{.Values.service.internalPort}}
15 {{- if .Values.canary.service.gateway }}
16   gateways:
17 {{ toYaml .Values.canary.service.gateway | indent 4 }}
18 {{- end }}
19 {{- if .Values.canary.service.hosts }}
20   hosts:
21 {{ toYaml .Values.canary.service.hosts | indent 4 }}
22 {{- end }}
23   canaryAnalysis:
24     interval: {{ .Values.canary.canaryAnalysis.interval }}
25     threshold: {{ .Values.canary.canaryAnalysis.threshold }}
26     maxWeight: {{ .Values.canary.canaryAnalysis.maxWeight }}
27     stepWeight: {{ .Values.canary.canaryAnalysis.stepWeight }}
28 {{- if .Values.canary.canaryAnalysis.metrics }}
```

```

29     metrics:
30 {{ toYaml .Values.canary.canaryAnalysis.metrics | indent 4 }}
31 {{- end }}
32 {{- end }}
33 " | tee charts/jx-progressive/templates/canary.yaml

```

To begin with, we enveloped the whole definition inside an `if` statement so that `Canary` is created only if the value `canary.enable` is set to `true`. That way, by default, we will not use canary deployments at all. Instead, we'll have to specify when and under which circumstances they will be created. That might spark the question “why do we go into the trouble of making sure that canary deployments are enabled only in certain cases?” Why not use them always?

By their nature, it takes much more time to execute a canary deployment than most of the other strategies. Canaries roll out progressively and are pausing periodically to allow us to validate the result on a subset of users. That increased duration can be anything from seconds to hours or even days or weeks. In some cases, we might have an important release that we want to test with our users progressively over a whole week. Such prolonged periods might be well worth the wait in production and staging, which should use the same processes. But, waiting for more than necessary to deploy a release in a preview environment is a waste. Those environments are not supposed to provide the “final stamp” before promoting to production, but rather to flush out most of the errors. After all, not using canaries in preview environments might not be the only difference. As we saw in the previous chapter, we might choose to make them serverless while keeping permanent environments using different deployment strategies. Long story short, the `if` statement allows us to decide when we'll do canary deployments. We are probably not going to employ it in all environments.

The `apiVersion`, `kind`, and `metadata` should be self-explanatory if you have minimal experience with Kubernetes and Helm templating.

The exciting entry is `spec.provider`. As the name suggests, it allows us to specify which provider we'll use. In our case, it'll be Istio, but I wanted to make sure that you can easily switch to something else like, for example, Gloo. If you are using GKE, you already have Gloo running. While exploring different solutions is welcome while learning, later on, you should probably use one or the other, not necessarily both.

The `spec.targetRef` tells Canary which Kubernetes object it should manipulate. Unlike serverless Deployments with Knative which replace Kubernetes Deployments, Canary runs on top of whichever Kubernetes resource we're using to run our software. For *jx-progressive* that's Deployment, but it could just as well be a `StatefulSet` or something else.

The next in line is `spec.progressDeadlineSeconds`. Think of it as a safety net. If the system cannot progress with the deployment for the specified period (expressed in seconds), it will initiate a rollback to the previous release.

The `spec.service` entry provides the information on how to access the application, both internally (`port`) and externally (`gateways`) as well as the hosts through which the end-users can communicate with the app.

The `spec.canaryAnalysis` entries are probably the most interesting ones. They define the analysis that should be done to decide whether to progress with the deployment or to roll back. Earlier I mentioned that the interval between progress iterations is thirty seconds. That's specified in the `interval` entry. The `threshold` defined how many failed metric checks are allowed before rolling back. The `maxWeight` sets the percentage of requests routed to the canary deployment before it gets converted to the primary. After that percentage is reached, all users will see the new release. More often than not, we do not need to wait until the process reaches 100% through smaller increments. We can say that, for example, when 50% of users are using the new release, there is no need to proceed with validations of the metrics. The system can move forward and make the new release available to everyone right away. The `stepWeight` entry defines how big roll out increments should be (e.g., 20% at a time). Finally, `metrics` can host an array of entries, one for each metric and threshold that should be evaluated continuously during the process.

As you noticed, many of the values are not hard-coded into the `Canary` definition. Instead, we defined them as Helm values. That way we can change all those that should be tweaked from `charts/jx-progressive/values.yaml`. So, let's create that file with some sample values.

```
1 echo "
2 canary:
3   enable: false
4   provider: istio
5   service:
```

```

6   hosts:
7     - jx-progressive.$ISTIO_IP.nip.io
8   gateways:
9     - jx-gateway.istio-system.svc.cluster.local
10  canaryAnalysis:
11    interval: 30s
12    threshold: 5
13    maxWeight: 70
14    stepWeight: 20
15    metrics:
16      - name: request-success-rate
17        threshold: 99
18        interval: 120s
19      - name: request-duration
20        threshold: 500
21        interval: 120s
22 " | tee -a charts/jx-progressive/values.yaml

```

We set the provider to `istio` since that is our service mesh technology of choice in this section.

Through the `service` entry, we set the external-facing `hosts` to a single address (`jx-progressive.$ISTIO_IP.nip.io`). For what we're about to do, we cannot rely on Jenkins X's ability to auto-generate addresses, so they need to be explicit. We'll keep that address as production so we'll have to set the one for staging separately later on. The `gateway` is set to `jx-gateway.istio-system.svc.cluster.local` which represents the `jx-gateway` created by Flagger.

The `canaryAnalysis` sets the interval to `30s`. So, it will progress with the rollout every half a minute. Similarly, it will roll back if it encounters failures (e.g., reaches metrics thresholds) 5 times. It will finish rollout when it reaches 70 percent of users (`maxWeight`), and it will increase the number of requests forwarded to the new release with increments of 20 percent (`stepWeight`).

Finally, it will use two metrics to validate rollouts and decide whether to proceed, to halt, or to roll back. The first metric is `request-success-rate` calculated throughout `120s`. If less than 99 percent of requests are successful (are not 5xx responses), it will be considered an error. Remember that does not necessarily mean that it will rollback right away since the `threshold` is set to 5. There must be five failures for the rollback to initiate. The second metric is `request-duration`. It is also measured throughout `120s` with the threshold of half a second (500 milliseconds). It does not take into account every request but rather the 99th percentile.

We added the `istio-injection=enabled` label to the staging and the production environment Namespaces. As a result, everything running in those Namespaces will automatically use Istio for networking.

There's only one more change missing before we can try Canary deployments in action. Do you remember that we set the `hosts` to `jx-progressive.$ISTIO_IP.nip.io`? Assuming that's the address through which our application will be exposed when running in production, we need to define a different address for staging. Otherwise, the releases running in both environments would be accessible through the same address, and that's almost certainly not something we want to do.

Apart from changing the address through which the application will be accessible when running in the staging environment, we'll also have to enable canary deployment. Remember, it is disabled by default.

The best place to add staging-specific values is the staging repository. We'll have to clone it, so let's get out of the local copy of the `jx-progressive` repo.

```
1 cd ..
```

Now we can clone the staging repository.



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

```
1 rm -rf environment-jx-rocks-staging
2
3 GH_USER=[...]
4
5 git clone \
6   https://github.com/$GH_USER/environment-jx-rocks-staging.git
7
8 cd environment-jx-rocks-staging
```

We removed the local copy of the environment repo just in case it was a leftover from the exercises in the previous chapters. After that, we cloned the repository and entered inside the local copy.

Next, we need to enable Canary deployments for the staging environment and to define the address through which `jx-progressive` be accessible.

```

1 STAGING_ADDR=staging.jx-progressive.$ISTIO_IP.nip.io
2
3 echo "jx-progressive:
4   canary:
5     enable: true
6     service:
7       hosts:
8         - $STAGING_ADDR" \
9   | tee -a env/values.yaml

```

All we did was to add a few variables associated with *jx-progressive*.

Now we can push the changes to the staging repository.

```

1 git add .
2
3 git commit \
4   -m "Added progressive deployment"
5
6 git push

```

With the staging-specific values defined, we can go back to the *jx-progressive* repository and push the changes we previously did over there.

```

1 cd ../jx-progressive
2
3 git add .
4
5 git commit \
6   -m "Added progressive deployment"
7
8 git push

```

We should not see a tangible change to the deployment process with the first release. So, all there is to do, for now, is to confirm that the activities initiated by pushing the changes to the repository were executed successfully.

```

1 jx get activities \
2   --filter jx-progressive/master \
3   --watch

```

Press *ctrl+c* to cancel the watcher once you confirm that the newly started build is finished.

```

1 jx get activities \
2   --filter environment-jx-rocks-staging/master \
3   --watch

```

Press *ctrl+c* to cancel the watcher once you confirm that the newly started build is finished.

Now we're ready with the preparations for Canary deployments, and we can finally take a closer look at the process itself as well as the benefits it brings.

## Using Canary Strategy With Flagger, Istio, And Prometheus

Before we start exploring Canary deployments, let's take a quick look at what we have so far to confirm that the first release using the new definition worked.

Is our application accessible through the Istio gateway?

```
1 curl ${STAGING_ADDR}/demo/hello
```

The output should say Hello from: Jenkins X golang http rolling update.

Now that we confirmed that the application released with the new definition is accessible through the Istio gateway, we can take a quick look at the Pods running in the staging Namespace.

```
1 kubectl \
2   --namespace jx-staging \
3   get pods
```

The output is as follows.

```
1 NAME                           READY STATUS RESTARTS AGE
2 jx-jx-progressive-primary-... 2/2   Running 0        42s
3 jx-jx-progressive-primary-... 2/2   Running 0        42s
4 jx-jx-progressive-primary-... 2/2   Running 0        42s
```

There is a change at least in the naming of the Pods belonging to *jx-progressive*. Now they contain the word `primary`. Given that we deployed only one release using `Canary`, those Pods represent the main release accessible to all the users. As you can imagine, the Pods were created by the corresponding `jx-jx-progressive-primary` ReplicaSet which, in turn, was created by the `jx-jx-progressive-primary` Deployment. As you can probably guess, there is also the `jx-jx-progressive-primary` Service that allows communication to those Pods, even though sidecar containers injected by Istio further complicate that. Later on, we'll see why all those are important.

What might matter more is the `canary` resource, so let's take a look at it.

```
1 kubectl --namespace jx-staging \
2     get canary
```

The output is as follows.

```
1 NAME           STATUS    WEIGHT  LASTTRANSITIONTIME
2 jx-jx-progressive  Initialized 0          2019-12-01T21:35:32Z
```

There's not much going on there since we have only the first Canary release running. For now, please note that `canary` can give us additional insight into the process.

You saw that we set the `Canary gateway` to `jx-gateway.istio-system.svc.cluster.local`. As a result, when we deployed the first Canary release, it created the gateway for us. We can see it by retrieving `virtualservice.networking.istio.io` resources.

```
1 kubectl --namespace jx-staging \
2     get virtualservices.networking.istio.io
```

The output is as follows.

```
1 NAME           GATEWAYS           HOSTS
2                               AGE
3 jx-jx-progressive [jx-gateway.istio-system.svc.cluster.local] [staging.jx-progress
4 e.104.196.199.98.nip.io jx-jx-progressive] 3m
```

We can see from the output that the gateway `jx-gateway.istio-system.svc.cluster.local` is handling external requests coming from `staging.jx-progressive.104.196.199.98.nip.io` as well as `jx-jx-progressive`. We'll focus on the former host and ignore the latter.

Finally, we can output Flagger logs if we want to see more details about the deployment process.

```
1 kubectl \
2     --namespace istio-system logs \
3     --selector app.kubernetes.io/name=flagger
```

I'll leave it to you to interpret those logs. Don't get stressed if you don't understand what each event means. We'll explore what's going on in the background as soon as we deploy the second release.

To see Canary deployments in action, we'll create a trivial change in the demo application by replacing `hello, rolling update!` in `main.go` to `hello, progressive!..`. Then, we will commit and merge it to master to get a new version in the staging environment.

```
1 cat main.go | sed -e \
2   "s@rolling update@progressive@g" \
3   | tee main.go
4
5 git add .
6
7 git commit \
8   -m "Added progressive deployment"
9
10 git push
```

Those were such trivial and commonly used commands that there is no need explaining them.

Just as with previous deployment strategies, now we need to be fast.

```
1 echo $STAGING_ADDR
```

Please copy the output and go to the **second terminal**.



Replace [...] in the command that follows with the copied address of *jx-progressive* in the staging environment.

```
1 STAGING_ADDR=[...]
2
3 while true
4 do
5   curl "$STAGING_ADDR"
6   sleep 0.2
7 done
```

As with the other deployment types, we initiated a loop that continuously sends requests to the application. That will allow us to see whether there is deployment-caused downtime. It will also provide us with the first insight into how canary deployments work.

Until the pipeline starts the deployment, all we're seeing is the `hello, rolling update!` message coming from the previous release. Once the first iteration of the rollout is finished, we should see both `hello, rolling update!` and `hello, progressive!` messages alternating. Since

we specified that `stepWeight` is 20, approximately twenty percent of the requests should go the new release while the rest will continue receiving the requests from the old. Thirty seconds later (the `interval` value), the balance should change. We should have reached the second iteration, with forty percent of requests coming from the new release and the rest from the old.

Based on what we can deduce so far, `Canary` deployments are behaving in a very similar way as `RollingUpdate`. The significant difference is that our rolling update examples did not specify any delay, so the process looked almost as if it was instant. If we did specify a delay in rolling updates and if we had five replicas, the output would be nearly the same.

As you might have guessed, we would not go into the trouble of setting up `Canary` deployments if their behavior is the same as with the `RollingUpdate` strategy. There's much more going on. We'll have to go back to the first terminal to see the other effects better.

Leave the loop running and go back to the **first terminal**

Let's see which Pods do we have in the staging Namespace.

```
1 kubectl --namespace jx-staging \
2   get pods
```

The output is as follows.

```
1 NAME                      READY STATUS RESTARTS AGE
2 jx-jx-progressive-...     2/2   Running 0        22s
3 jx-jx-progressive-...     2/2   Running 0        22s
4 jx-jx-progressive-...     2/2   Running 0        22s
5 jx-jx-progressive-primary-... 2/2   Running 0        9m
6 jx-jx-progressive-primary-... 2/2   Running 0        9m
7 jx-jx-progressive-primary-... 2/2   Running 0        9m
```

Assuming that the process did not yet finish, we should see that besides the `jx-jx-progressive-primary` we also got `jx-jx-progressive` (without `-primary`). If you take a closer look at the `AGE`, you should notice that all the Pods were created a while ago except `jx-progressive`. That's the new release, and we'll call it "canary Pod". Flagger has both releases running during the deployment process. Initially, all traffic was being sent to the primary Pods. But, when the deployment process was initiated, `VirtualService` started sending traffic to one or another, depending on the iteration and the `stepWeight`. To be more precise, the percentage of

requests being sent to the new release is equivalent to the iteration multiplied with `stepWeight`. Behind the scenes, Flagger is updating Istio `VirtualService` with the percentage of requests that should be sent to one group of Pods or another. It is updating `VirtualService` telling it how many requests should go to the Service associated with `primary` and how many should go to the one associated with “canary” Pods.

Given that much of the action is performed by the `VirtualService`, we’ll take a closer look at it and see whether we can gain some additional insight.



Your outputs will probably differ from mine depending on the deployment iteration (stage) you’re in right now. Follow my explanations of the outputs even if they are not the same as what you’ll see on your screen.

```
1 kubectl --namespace jx-staging \
2   get virtualservice.networking.istio.io \
3     jx-jx-progressive \
4   --output yaml
```

The output, limited to the relevant parts, is as follows.

```
1 ...
2 spec:
3   gateways:
4     - jx-gateway.istio-system.svc.cluster.local
5   hosts:
6     - staging.jx-progressive.104.196.199.98.nip.io
7     - jx-jx-progressive
8   http:
9     - route:
10       - destination:
11         host: jx-jx-progressive-primary
12         weight: 20
13       - destination:
14         host: jx-jx-progressive-canary
15         weight: 80
```

The interesting part is the `spec` section. In it, besides the `gateways` and the `hosts`, is `http` with two `routes`. The first one points to `jx-progressive-primary`, which is the old release. Currently, at least in my case, it has the weight of 40. That means that the `primary` (the old) release is currently receiving forty percent of requests. On the other hand, the rest of sixty percent is going to the `jx-progressive-canary` (the new) release. Gradually, Flagger was increasing the weight of `canary` and decreasing

the primary, thus gradually shifting more and more requests from the old to the new release. Still, so far all that looks just a “fancy” way to accomplish what rolling updates are already doing. If that thought is still passing through your head, soon you’ll see very soon that there’s so much more.

An easier and more concise way to see the progress is to retrieve the `canary` resource.

```
1 kubectl --namespace jx-staging \
2     get canary
```

The output is as follows.

```
1 NAME           STATUS    WEIGHT  LASTTRANSITIONTIME
2 jx-jx-progressive  Progressing  60      2019-08-16T23:24:03Z
```

In my case, the process is still progressing and, so far, it reached 60 percent. In your case, the `weight` is likely different, or the `status` might be succeeded. In the latter case, the process is finished successfully. All the requests are now going to the new release. The deployment rolled out fully.

If we describe that `canary` resource, we can get more insight into the process by observing the events.

```
1 kubectl --namespace jx-staging \
2     describe canary jx-jx-progressive
```

The output, limited to the `events` initiated by the latest deployment, is as follows.

```
1 ...
2 Events:
3   Type    Reason  Age   From       Message
4   ----  -----  ---  ----
5 ...
6   Normal  Synced  3m32s flagger New revision detected! Scaling up jx-jx-progressive
7 x-staging
8   Normal  Synced  3m2s  flagger Starting canary analysis for jx-jx-progressive.jx-
9 ging
10  Normal  Synced  3m2s  flagger Advance jx-progressive.jx-staging canary weight 20
11  Normal  Synced  2m32s flagger Advance jx-progressive.jx-staging canary weight 40
12  Normal  Synced  2m2s  flagger Advance jx-progressive.jx-staging canary weight 60
13  Normal  Synced  92s   flagger Advance jx-progressive.jx-staging canary weight 80
14  Normal  Synced  92s   flagger Copying jx-progressive.jx-staging template spec to
15 -progressive-primary.jx-staging
16  Normal  Synced  62s    flagger Routing all traffic to primary
17  Normal  Synced  32s    flagger Promotion completed! Scaling down jx-progressive.j
18 taging
```

If one of the last two events does not state promotion completed, please wait for a while longer for the process to finish and re-run the `kubectl describe` command.

We can see that when the deployment was initiated, Flagger detected that there is a new revision (a new release). As a result, it started scaling the application. A while later, it initiated the analysis that consists of evaluations of the metrics (`request-success-rate` and `request-duration`) against the thresholds we defined earlier. Further on, we can see that it was increasing the weight every thirty seconds until it reached 80 percent. That number is vital given that it is the first iteration with the weight equal to or above the `maxWeight` which we set to 70 percent. After that, it did not wait for another thirty seconds. Instead, it replaced the definition of the `primary` template to the one used for `canary`. From that moment on, the `primary` was updated to the new release and all the traffic is being routed to it. Finally, the last event was the message that the promotion was completed and that the `canary` (`jx-progressive.jx-staging`) was scaled down to zero replicas. The last two events happened at the same time, so in your case, their order might be reverted.

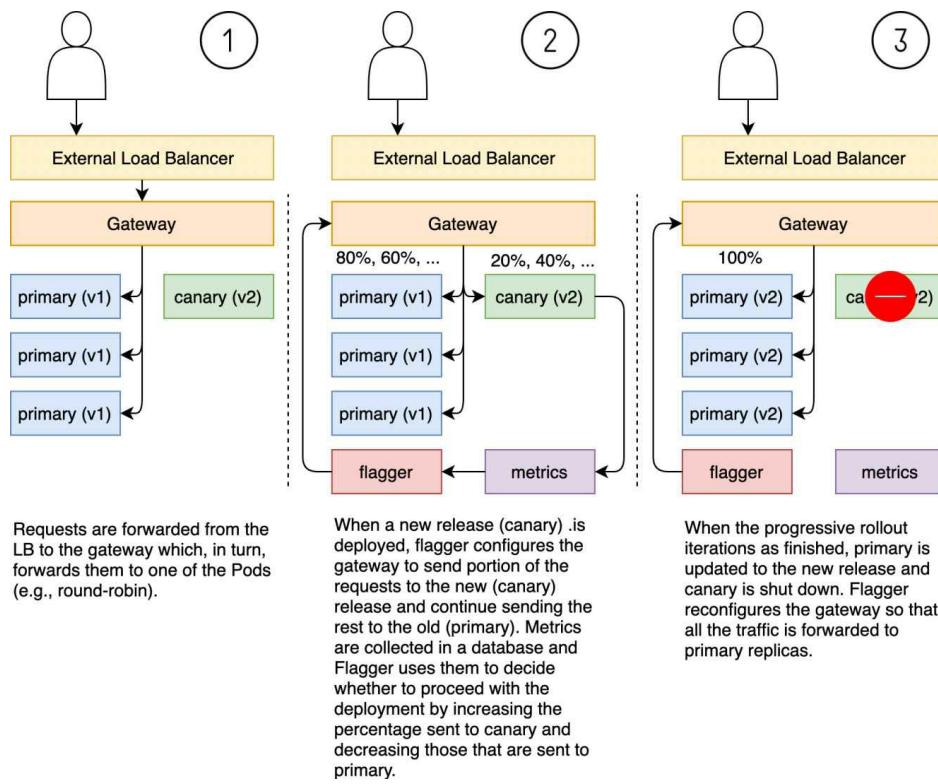


Figure 17-7: Canary deployment rollout

We finally found a big difference between `Canary` and `RollingUpdate` deployments. That difference was in the evaluation of the metrics as a way to decide whether to proceed or not with the rollout. Given that everything worked correctly. We are yet to see what would happen if one of the metrics reached the threshold.

We're finished exploring the happy path, and we can just as well stop the loop. There's no need, for now, to continue sending requests. But, before we do that, I should explain that there was a reason for the loop beside the apparent need to see the progress.

If we were not sending requests to the application, there would be no metrics to collect. In such a case, Flagger would think that there's something fishy with the new release. Maybe it was so bad that no one could send a request to our application. Or maybe there was some other reason. In any case, lack of metrics used to validate the release is considered a problem. For now, we wanted to see the happy path with the new release fully rolled out. The stream of requests was there to ensure that there are sufficient metrics for Flagger to say: "everything's fine; let's move on."

With that out of the way, please go back to the **second terminal**, stop the loop by pressing `ctrl+c`, and go back again to the **first terminal**.

Next, we'll see how does the "unhappy" path look.

## **Rolling Back Canary Deployments**

Here's the good news. Flagger will automatically roll back if any of the metrics we set fails the number of times set as the `threshold` configuration option. Similarly, it will also roll back if there are no metrics.

In our case, there are at least three ways we can run an example that would result in a rollback. We could start sending requests that would generate errors until we reach the `threshold` of the `request-success-rate` metric. Or we can create some slow requests so that `request-duration` is messed up (above 500 milliseconds). Finally, we could not send any request and force Flagger to interpret lack of metrics as an issue.

We won't do a practical exercise that would demonstrate canary deployment roll backs. To begin with, the quickstart application we're

using does not have a mechanism to produce slow responses or “fake” errors. Also, such an exercise would be mostly repetition of the practices we already explored. Finally, the last reason for avoiding rollbacks lies in the scope of the subject. Canary deployments with Istio, Flagger, Prometheus, and a few other tools are a huge subject that deserves more space than a part of a chapter. It is a worthy subject though and I will most likely release a set of articles, a book, or a course that would dive deeper into those tools and the process of canary deployments. Stay tuned.

So, you’ll have to trust me when I say that if any metrics defined in the canary resource fail a few times, the result would be rollback to the previous version.

Finally, I will not show you how to implement canary deployments in production. All you have to do is follow the same logic as the one we used with *jx-progressive* in the staging environment. All you’d need to do is set the `jx-progressive.canary.enable` variable to `true` inside the `values.yaml` file in the production environment repository. The production host is already set in the chart itself.

## To Canary Or Not To Canary?

We saw one possible implementation of canary deployments with Flagger, Istio, and Prometheus. As we discussed, we could have used other tools. We could have just as well created more complex formulas that would be used to decide whether to proceed or roll back a release. Now the time has come to evaluate canary deployments and see how they fit the requirements we defined initially.

Canary deployments are, in a way, an extension of rolling updates. As such, all of the requirements fulfilled by one are fulfilled by the other.

The deployments we did using the canary strategy were highly available. The process itself could be qualified as rolling updates on steroids. New releases are initially being rolled out to a fraction of users and, over time, we were increasing the reach of the new release until all requests were going there. From that perspective, there was no substantial difference between rolling updates and canary releases. However, the process behind the scenes was different.

While rolling updates are increasing the number of Pods of the new release and decreasing those from the old, canaries are leaving the old release untouched. Instead, a new Pod (or a set of Pods) is spin up, and service mesh (e.g., Istio) is making sure to redirect some requests to the new and others to the old release. While rolling updates are becoming available to users by changing how many Pods of the new release are available, canaries are accomplishing the same result through networking. As such, canaries allow us to have much greater control over the process.

Given that with canary deployments we are in control over who sees what, we can fine-tune it to meet our needs better. We are less relying on chance and more on instructions we're giving to Flagger. That allows us to create more complicated calculations. For example, we could let only people from one country see the new release, check their reactions, and decide whether to proceed with the rollout to everyone else.

All in all, with canary deployments, we have as much high-availability as with rolling updates. Given that our applications are running in multiple replicas and that we can just as easily use HorizontalPodAutoscaler or any other Kubernetes resource type, canary deployments also make our applications as responsive as rolling updates.

Where canary deployments genuinely shine and make a huge difference is in the progressive rollout. While, as we already mentioned, rolling updates give us that feature as well, the additional control of the process makes canary deployments true progressive rollout.

On top of all that, canary deployments were the only ones that had a built-in mechanism to roll back. While we could extend our pipelines to run tests during and after the deployment and roll back if they fail, the synergy provided by canary deployments is genuinely stunning. The process itself decides whether to roll forward, to temporarily halt the process, or to roll back. Such tight integration provides benefits which would require considerable effort to implement with the other deployment strategies. I cannot say that only canary deployments allow us to roll back automatically. But, it is the only deployment strategy that we explored that has rollbacks integrated into the process. And that should come as no surprise given that canary deployments rely on using a defined set of rules to decide when to progress with rollouts. It would be strange if the opposite is true. If a machine can decide when to move forward, it can just as well decide when to move back.

Judging by our requirements, the only area where the `Canary` deployment strategy is not as good or better than any other is cost-effectiveness. If we want to save on resources, serverless deployments are in most cases the best option. Even rolling updates are cheaper than canaries. With rolling updates, we replace one Pod at the time (unless we specify differently). However, with `Canary` deployments, we keep running the full old release throughout the whole process, and add the new one in parallel. In that aspect, canaries are similar to blue-green deployments, except that we do not need to duplicate everything. Running a fraction (e.g., one) of the Pods with the new release should be enough.

All in all, canaries are expensive or, at least, more expensive than other strategies, excluding blue-green that we already discarded.

All in all, canary deployments, at least the version we used, provide **high-availability**. They are **responsive**, and they give us **progressive rollout** and **automatic rollbacks**. The major downside is that they are **not as cost-effective** as some other deployment strategies.

The summary of the fulfillment of our requirements for the `Recreate` deployment strategy is as follows.

Requirement	Fullfilled
High-availability	Fully
Responsiveness	Fully
Progressive rollout	Fully
Rollback	Fully
Cost-effectiveness	Not

The last piece of the canary puzzle is to figure out how to visualize canary deployments.

## Visualizing Rollouts Of Canary Deployments

Jenkins X Flagger addon includes Grafana with a dashboard that we can use to see metrics related to canary deployments visually. However, there is a tiny problem. Grafana Ingress was not created so, for now, Grafana's UI is not accessible. We'll fix that easily by creating an Ingress pointing to the Grafana service. For that, first we need to find out the IP of the Ingress controller.

Please note that the commands that follow will differ depending on whether you are using EKS or some other Kubernetes provider.



Please run the command that follows only if you are **NOT** using EKS (e.g., GKE, AKS, etc.).

```
1 LB_IP=$(kubectl \
2   --namespace kube-system \
3   get svc jxing-nginx-ingress-controller \
4   -o jsonpath='{.status.loadBalancer.ingress[0].ip}'")
```



Please run the commands that follow only if you are using **EKS**.

```
1 LB_HOST=$(kubectl \
2   --namespace kube-system \
3   get svc jxing-nginx-ingress-controller \
4   --output jsonpath='{.status.loadBalancer.ingress[0].hostname}') \
5 
6 export LB_IP="$(dig +short $LB_HOST \
7   | tail -n 1)"
```

Finally, you might not be using Ingress controller installed by Jenkins X. For example, you might be using the “official” NGINX Ingress (the one from Jenkins X is a variation of it). If that’s the case, you’ll have to modify the command(s) to fit your situation.

No matter how we retrieved the IP of the external load balancer, we’ll output it as a way to have a visual confirmation that it looks OK.

```
1 echo $LB_IP
```

Now that we have the IP, we can create the missing Ingress.

```
1 echo "apiVersion: extensions/v1beta1
2 kind: Ingress
3 metadata:
4   annotations:
5     kubernetes.io/ingress.class: nginx
6   name: flagger-grafana
7   namespace: istio-system
8 spec:
9   rules:
10  - host: flagger-grafana.$LB_IP.nip.io
11    http:
```

```
12     paths:
13       - backend:
14         serviceName: flagger-grafana
15         servicePort: 80
16   " | kubectl create -f -
```



Do not create resources with ad-hoc commands like the one we just executed. That is undocumented and hard to reproduce. Store everything in a declarative format (e.g., YAML) in a Git repository and let Jenkins X deal with its deployment. We did what we did only to make it easy, not as a suggestion to follow the same practice in “real world” situations.

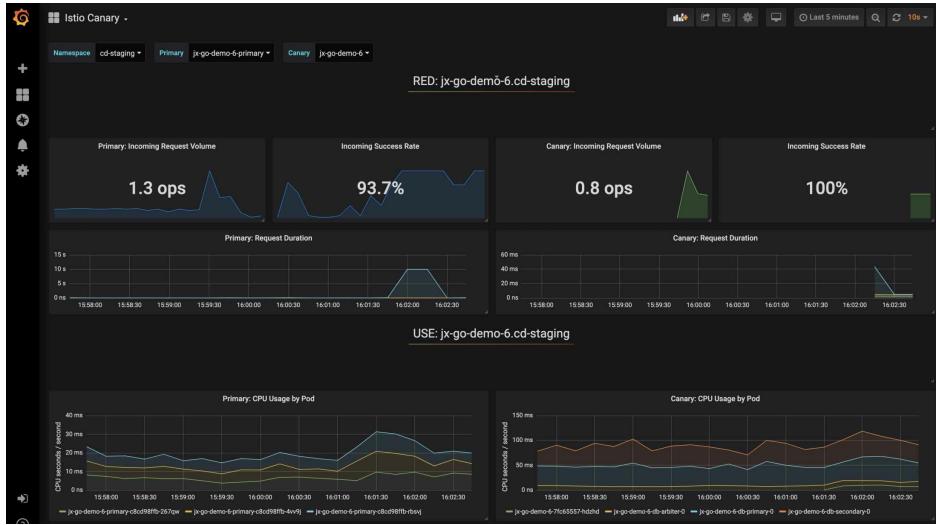
With Ingress defined, we can, finally, open Grafana UI.

```
1 open "http://flagger-grafana.$LB_IP.nip.io"
```

Just as Istio is not the main subject of this book, neither is Grafana. So, we won’t go into details. I expect you to figure them out yourself. Instead, we’ll take a quick look at the predefined Flagger dashboard.

Please select *Dashboards* from the left-hand menu, click the *Manage* button, and select *Istio Canary*,

The dashboard is in front of us, and I’ll leave you to explore it. But, before I go, please note that to visualize the process, you should make yet another change to the source code, push it to GitHub, and wait until the pipeline-initiated canary deployment starts. From there on, please select `jx-staging` as the *namespace*, choose `jx-progressive-primary` as *primary* and `jx-progressive` as *canary*. Now you should be able to visualize the process.



**Figure 17-9: The dashboard with visualizations related to canary deployments**

That's all I'm going to say about Grafana. It's not the focus of this book, so you're on your own if you're not already familiar with it. Right now, we are left with the last and potentially the most important discussion.

## Which Deployment Strategy Should We Choose?

We saw some of the deployment strategies. There are others, and this chapter does not exclude you from exploring them. Please do that. The more you learn, the more educated decisions you'll make. Still, until you figure out all the other strategies and variations you can do, we accumulated enough material to talk summarize what we learned so far.

Can we conclude that canary deployments are the best and that everyone should use them for all their applications? Certainly not. To begin with, if an application is not eligible for rolling updates (e.g., a single replica app), it is almost certainly not suitable for canary deployments. If we think of canaries as extensions of rolling updates, if an app is not a good candidate for the latter, it will also not be fit for the former. In other words, there is a good use case for using the `Recreate` strategy in some cases, specifically for those applications that cannot (or shouldn't) use one of the other strategies.

So, if both `Canary` and `Recreate` strategies have their use cases, can we discard rolling updates and serverless?

Rolling updates should, in most cases, we replaced with canary deployments. The applications eligible for one deployment strategy qualify for the other, and canaries provide so much more. If nothing else, they give us a bigger safety net. The exceptions would be tiny clusters serving small teams. In those cases, the resource overhead added by a service mesh (e.g., Istio) and metrics collector and database (e.g., Prometheus) might be too much. Another advantage of rolling updates is simplicity. There are no additional components to install and manage, and there are no additional YAML files. Long story short, canary deployments could easily replace all your rolling updates, as long as the cost (on resources and operations) is not too high for your use case.

That leaves us with serverless (e.g., Knative). It would be hard for me to find a situation in which there is no use for serverless deployments. It has fantastic scaling capabilities on its own that can be combined with HorizontalPodAutoscaler. It saves us money by shutting (almost) everything down when our applications are not in use.

Knative ticks all the boxes, and the only downside is the deployment process itself, which is more elaborated with canaries. The more important potential drawback is that scaling from nothing to something can introduce a delay from the user's perspective. Nevertheless, that is rarely a problem. If an application is unused for an extended period, users rarely complain when they need to wait for an additional few seconds for the app to wake up.

So, we are in a situation where one solution (`Canary`) provides better capabilities for the deployment process itself, while the other (`serverless`) might be a better choice as a model for running applications. Ultimately, you'll need to make a choice. What matters more? Is it operational cost (`use serverless`) or deployment safety net (`use canary`)? You might be able to combine the two but, at the time of this writing (August 2019), that is not that easy since the integration is not available in Flagger or other similar tools.

What is essential, though, is that it is not the winner-takes-all type of a decision. We can use `Recreate` with some applications, `RollingUpdate` with others, and so on. But it goes beyond choosing a single deployment strategy for a single application. Deployment types can differ from one environment to the other. Canaries, for example, are not a good choice for

preview environments. All we'd get is increased time required to terminate the deployment process and potential failures due to the lack of metrics.

Let's make a quick set of rules when to use one deployment strategy over the other. Bear in mind that what follows is not a comprehensive list but rather elevator pitch for each deployment type.

Use the **recreate** strategy when working with legacy applications that often do not scale, that are stateful without replication, and are lacking other features that make them not cloud-native.

Use the **rolling update** strategy with cloud-native applications which, for one reason or another, cannot use canary deployments.

Use the **canary** strategy instead of **rolling update** when you need the additional control when to roll forward and when to roll back.

Use **serverless** deployments in permanent environments when you need excellent scaling capabilities or when an application is not in constant use.

Finally, use **serverless** for all the deployments to preview environments, no matter which strategy you're using in staging and production.

Finally, remember that your Kubernetes cluster might not support all those choices, so choose among those that you can use.

## What Now?

That was a long chapter, and you deserve a break.

If you created a cluster only for the exercises we executed, please destroy it. We'll start the next, and each other chapter from scratch as a way to save you from running your cluster longer than necessary and pay more than needed to your hosting vendor. If you created the cluster or installed Jenkins X using one of the Gists from the beginning of this chapter, you'll find the instructions on how to destroy the cluster or uninstall everything at the bottom of those Gists.

If you did choose to destroy the cluster or to uninstall Jenkins X, please remove the repositories we created as well as the local files. You can use the commands that follow for that. Just remember to replace [...] with your GitHub user and pay attention to the comments.

```
1 cd ..
2
3 GH_USER=[...]
4
5 hub delete -y \
6     $GH_USER/environment-jx-rocks-staging
7
8 hub delete -y \
9     $GH_USER/environment-jx-rocks-production
10
11 hub delete -y \
12     $GH_USER/jx-progressive
13
14 rm -rf environment-jx-rocks-staging
15
16 rm -rf $GH_USER/jx-progressive
```

Finally, you might be planning to move into the next chapter right away. If that's the case, there are no cleanup actions to do. Just keep reading.

# Applying GitOps Principles To Jenkins X



At the time of this writing (October 2019), the examples in this chapter are validated only with **serverless Jenkins X** in **GKE**. Jenkins X Boot is currently verified by the community to work only there, even though it likely works in EKS, AKS, and other Kubernetes flavors. Over time, the community will be adding support for all Kubernetes distributions, and, with a slight delay, I will be updating this chapter to reflect that. Still, there will be an inevitable delay between the progress of that support and me incorporating it into this book, so I strongly advise you to check the official documentation to see whether your Kubernetes flavor is added.

If there is a common theme in Jenkins X, that is GitOps. The platform was designed and built around the idea that everything is defined as code, that everything is stored in Git, and that every change to the system is initiated by a Git webhook. It's a great concept, and we should ask ourselves whether we were applying it in all our examples. What do you think?

Using a single `jx create cluster` command to create a whole cluster and install all the tools that comprise Jenkins X is excellent. Isn't it?

The community behind Jenkins X thought that providing a single command that will create a cluster and install Jenkins X is a great idea. But we were wrong. What we did not envision initially was that there is an almost infinite number of permutations we might need. There are too many different hosting providers, too many different Kubernetes flavors, and too many components we might want to have inside our clusters. As a result, the number of arguments in `jx create cluster` and `jx install` commands was growing continuously. What started as a simple yet handy feature ended up as a big mess. Some were confused with too many arguments, while others thought that too many are missing. In my opinion, Jenkins X, in this specific context, tried to solve problems that were already addressed by other tools. We should not use Jenkins X to create a cluster. Instead, we should use the tool specialized for that. My favorite is Terraform, but almost any other (e.g., CloudFormation, Ansible, Puppet, Chef, etc.) would do a better job. What that means is that Jenkins X should assume that we already have a fully operational Kubernetes cluster. We

should have an equivalent of the `jx install` command, while `jx create cluster` should serve particular use-cases, mostly related to demos and playgrounds, not a real production setting.

Now, you might just as well say, “if I’m not going to use `jx create cluster`, why should we use configuration management tools like Terraform? Isn’t it easier to simply run `gcloud`, `eksctl`, or `az` CLI?” It is indeed tempting to come up with a single one-off command to do things. But, that leads us to the same pitfalls as when using UIs. We could just as easily replace the command with a button. But that results in a non-reproducible, non-documented, and not omnipotent way of doing things. Instead, we want to modify some (preferably declarative) files, push them to Git, and let that initiate the process that will converge the actual with the desired state. Or, it can be vice versa as well. We could run a process based on some files and push them later. As long as a Git repository always contains the desired state, and the system’s actual state matches those desires, we have a reproducible, documented, and (mostly) automated way to do things. Now, that does not mean that commands and buttons are always a bad thing. Running a command or pressing a button in a UI works well only when that results in changes of the definition of the desired state being pushed to a Git repository so that we can keep track of changes. Otherwise, we’re just performing arbitrary actions that are not documented and not reproducible. What matters is that Git should always contain the current state of our system. How we get to store definitions in Git is of lesser importance.

I’m rambling how bad it is to run arbitrary commands and pushing buttons because that’s what we were doing so far. To be fair, we did follow GitOps principles most of the time, but one crucial part of our system kept being created somehow arbitrarily. So far, we were creating a Kubernetes cluster and installing Jenkins X with a single `jx create cluster` command, unless you are running it in an existing cluster. Even if that’s the case, I taught you to use the `jx install` command. No matter which of the two commands you used, the fact is that we do not have a place that defines all the components that constitute our Jenkins X setup. That’s bad, and it breaks the GitOps principle that states that everything is defined as code and stored in a Git repository, and that Git is the only one that initiates actions aimed at converging the actual into the desired state. We’ll change that next by adding the last piece of the puzzle that prevented us from having the full system created by following the GitOps principles.

We'll learn how *Jenkins X Boot* works but, before we do that, we'll take a quick look at a cardinal sin we were committing over and over again.

## Discussing The Cardinal Sin

Our applications and their direct third-party dependencies are managed by Jenkins X pipelines. A process is initiated with a push to a Git repository, thus adhering to GitOps principles. If our applications have direct third-party dependencies, we'll specify them in `requirements.yaml`. We've been doing that for a while, and there is probably no need to revisit the process that works well.

If we move higher into system-level third-party applications, we had mixed results so far. We could specify them as dependencies in environment repositories. If, for example, we need Prometheus in production, all we have to do is define it as yet another entry in `requirements.yaml` in the repository associated with it. Similarly, if we'd like to deploy it to the staging environment for testing, we could specify it in the repository related to staging. So far, so good. We were following GitOps principles. But, there was one crucial application in this group that we chose to deploy ignoring all the best practices we established so far. The one that eluded us is Jenkins X itself.

I was very vocal that we should enforce GitOps principles as much as we can. I insisted that everything should be defined as code, that everything should be stored in Git, and that only Git can initiate actions that will change the state of the cluster. Yet, I failed to apply that same logic to the creation of a cluster and installation of Jenkins X. Running commands like `jx create cluster` and `jx install` goes against everything I preached (at least in this book). Those commands were not stored in Git, and they are not idempotent. Other people working with us would not be able to reproduce our actions. They might not even know what was done and when.

So, why did I tell you to do things that go against my own beliefs?

Running `jx create cluster` command was the easiest way to get you up to speed with Jenkins X quickly. It is the fastest way to create a cluster and install Jenkins X. At the same time, it does not need much of an explanation. *Run a command, wait for a few minutes, and start exploring the examples that explain how Jenkins X works.* For demos, for workshops,

and for examples in a book, `jx create cluster` is probably the best way to get up-and-running fast. But, the time has come to say that such an approach is not good enough for any serious usage of Jenkins X. We explored most of the essential features Jenkins X offers, and I feel that you are ready to start using it in “real world”. For that to happen, we need to be able to set up Jenkins X using the same principles we applied to other use-cases. We should store its definition in Git, and we should allow webhooks to notify the system if any aspect of Jenkins X should change. We should have a pipeline that drives Jenkins X through different stages so that we can validate that it works correctly. In other words, Jenkins X should be treated the same way as other applications. Or, to be more precise, maybe not in exactly the same way but, at least, with a process based on the same principles. We’ll accomplish that with `jx boot`. But, before we explore it, there is one more level we should discuss.

We used `jx create cluster` both to create a cluster and to install Jenkins X. The latter part will, from now on, be done differently using `jx boot`. We still need to figure out how to create a cluster using code stored in Git. But, for better or worse, I won’t teach you that part. That’s outside the scope of this book. The only thing I will say is that you should define your whole infrastructure as code using one of the tools designed for that. My personal preference is [Terraform by HashiCorp](#). However, since infrastructure-as-code is not the subject, I won’t argue why I prefer Terraform over other tools. You can use anything you like; CloudFormation, Ansible, Puppet, Chef, SaltStack, etc.

All in all, we’ll continue deploying our application and associated third-party services as well as system-level third-party apps using Jenkins X pipelines. We’ll explore `jx boot` as a much better way to install and manage Jenkins X. As for infrastructure, it’ll be up to you to choose what you prefer. From now on, I will provide commands to create a Kubernetes cluster. Do NOT take those commands as a recommendation. They are only the representation of my unwillingness to dive into the subject of “infrastructure as code” in this book. So, from now on, whenever you see `gcloud`, `eksctl`, or `az` commands, remember that is not what you should do with the production system.

Now that we got that out of the way, let’s create a Kubernetes cluster and start “playing” with `jx boot`.

## Creating A Kubernetes Cluster (Without Jenkins X)



As mentioned at the beginning of this chapter, the examples are verified only in GKE, given that's the only currently (October 2019) supported platform for `jx boot`. Double-check the documentation to see whether that changed or be brave and try it out yourself with other Kubernetes flavors.

From now on, we will not use `jx create cluster` to create a Kubernetes cluster and install Jenkins X. Instead, I will assume that you will create a cluster any way you like (e.g., Terraform, `gcloud`, etc.) and we'll focus only on how to set up Jenkins X. If you're lazy and do not yet want to figure out the best way to create a cluster, the Gists that follow can get you up-to-speed fast. Just remember that the commands in them are not the recommended way to create a cluster, but rather the easiest and fastest method I could come up with.

I opted for commands instead of, let's say, Terraform for two reasons. First of all, I'm trying to keep this neutral as a way to avoid influencing you about the tool you should use for managing infrastructure. Now, I'm aware that I already stated a couple of times that I prefer Terraform, so I cannot say that I am neutral (I'm not), so that argument is not really valid, so let's jump to the second one. The more important reason for not using any tool for creating and setting up a cluster lies in the decision not to spend time explaining such a tool. This is neither time nor place to enter into configuration management discussion.

Finally, please note that, just as before, the Gists contain instructions not only to create but also to destroy the cluster. For now, focus on creating it and postpone your destructive tendencies for the end.



All the commands from this chapter are available in the [18-boot.sh](#) Gist.

The Gists are as follows.

- Create new **GKE** cluster: [gke.sh](#)

Now that we have a Kubernetes cluster, we can turn our attention to Jenkins X Boot.

## What Is Jenkins X Boot?

What's wrong with `jx create cluster` and `jx install` commands? Why do we need a different way to install, manage, and upgrade Jenkins X? Those are ad-hoc commands that do not follow GitOps principles. They are not idempotent (you cannot run them multiple times and expect the same result). They are not stored in Git, at least not in a form that the system can interpret and consume in an attempt to converge the desired into the actual state. They are not declarative.

We know all those reasons. They are the same issues we're trying to get rid of with other applications by implementing pipelines and declarative configurations and storing everything in Git. But, there is one more reason that we did not discuss yet.

The `jx` CLI contains a lot of custom-made code written specifically for Jenkins X. Part of it, however, consists of wrappers. The `jx create pullrequest` command is a wrapper around `hub`. Similarly, `jx create cluster` is a wrapper around `az` (AKS), `eksctl` (EKS), `gcloud` (GKE), and a few others. That's all great because `jx` allows us to have a single user-facing CLI that enables us to do (almost) everything we need, at least with the tasks related to the application lifecycle. But, in my opinion, Jenkins X tried to do more than it should. Its scope went overboard by attempting to enter into the configuration management sphere and trying to provide ways to create Kubernetes clusters. Truth be told, that was done mostly so that project contributors could spin up clusters easily.

Nevertheless, the feature became popular, and many started using it. That begs the question. If many are adopting something, isn't that a sign that it is useful?

Over time, the number of requests to add additional features (arguments) to `jx create cluster` kept increasing. People were missing things that are already available in the arguments of the commands it was wrapping, so the community kept adding them. At one point, the command became too confusing with too many arguments, and it went far beyond the initial idea to quickly spin up a cluster, mostly for demo purposes.

Jenkins X boot is a result of two primary needs; to create infrastructure and to install and maintain Jenkins X and other system-level components. It, in a way, puts the creation of the cluster back to the commands like `gcloud`, `az`, and `eksctl` (to name a few). At the same time, it urges people not to use them and to opt for tools like Terraform to create a Kubernetes cluster. On the other hand, it transforms the `jqx install` process from being a command into a GitOps-based process. As you will see soon, `jqx boot` entirely depends on a set of declarative files residing in a Git repository. It embraces GitOps principles, as it should have been done from the start.

So, given that having a declarative GitOps-based process is better than one-shot commands, you might be asking, “why did it take the community so long to create it?” Jenkins X Boot was released sometime around August 2019, over a year and a half after Jenkins X project started. What took us so long?

Jenkins X Boot had the “chicken and egg” type of a problem. Jenkins X assumes that everything is a pipeline triggered by a Git webhook. We make a change to Git, Git triggers a webhook, and that webhook notifies the cluster that there is a change that should trigger a pipeline that will ultimately converge the actual into the desired state. The process that performs that convergence is executed by Jenkins X running in that cluster. Now, if we say that the installation of Jenkins X should follow the same process, who will run those steps? How can we make Jenkins X install itself? If it doesn’t exist, it cannot run the process that will install it. When there is nothing in a cluster, nothing cannot create something. That would be magic.

The solution to the problem was to allow `jqx` CLI to run a pipeline as well. After all, until we install Jenkins X inside a Kubernetes cluster, the only thing we have is the CLI. So, the community made that happen. It created the necessary changes in the CLI that allows it to run pipelines locally. After that, all that was needed was to create a pipeline definition that we can use together with a bunch of configuration files that define all aspects of Jenkins X installation.

As a result, now we can use Jenkins X Boot to run the first build of a pipeline specifically defined for installation and upgrades of Jenkins X and all its dependencies. To make things even more interesting, the same pipeline can (and should) be executed by Jenkins X inside a Kubernetes

cluster. Finally, we can have the whole platform defined in a Git repository, let a pipeline run the first time locally and install Jenkins X, as well as configure webhooks in that repo so that each consecutive change to it triggers new builds that will change the actual state of the platform into the desired one. There is no need to run the pipeline locally ever again after the initial run. From there on, it'll be handled by Jenkins X inside a Kubernetes cluster just like any other pipeline. Finally, we can treat Jenkins X just like any other application (platform) controlled by pipelines and defined as declarative files.

We could summarize all that by saying that ad-hoc commands are bad, that GitOps principles backed by declarative files are good, and that Jenkins X can be installed and managed by pipelines, no matter whether they are run by a local CLI or through Jenkins X running inside a Kubernetes cluster. Running a pipeline from a local CLI allows us to install Jenkins X for the first time (when it's still not running inside the cluster). Once it's installed, we can let Jenkins X maintain itself by changing the content of the associated repository. As an added bonus, given that a pipeline can run both locally and inside a cluster, if there's something wrong with Jenkins X in Kubernetes, we can always repair it locally by re-running the pipeline. Neat, isn't it?

Still confused? Let's see all that in practice.

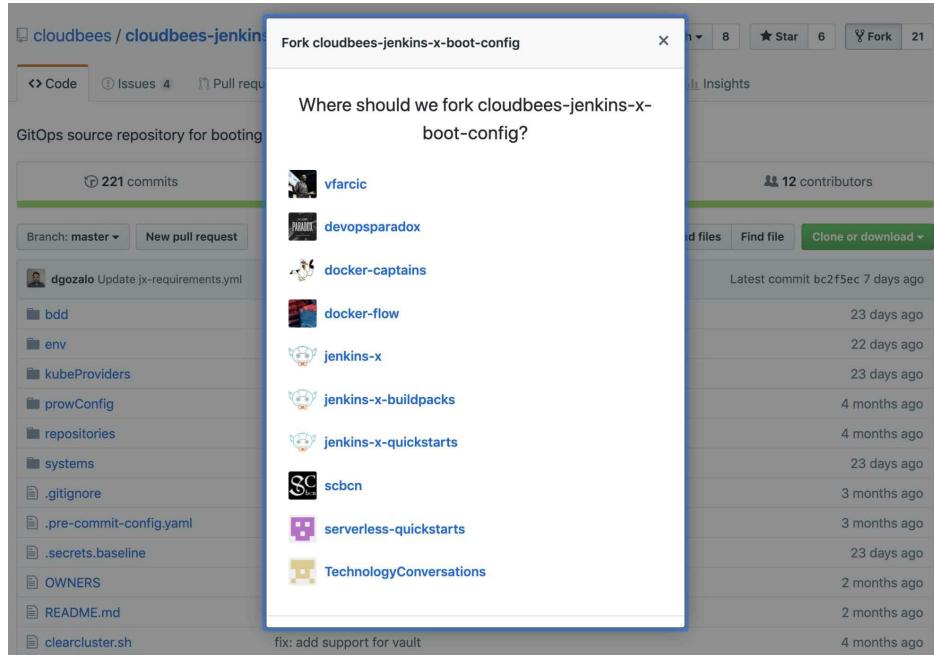
## Installing Jenkins X Using GitOps Principles

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.

Let's take a look at the repository.

```
1 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. For now, what matters is that you should fork the repository since we'll make some modifications and use it as yet another environment repo firing webhooks to Jenkins X.



**Figure 18-1: Forking Jenkins X Boot repository**

Now that you forked the repository, 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.

```
1 CLUSTER_NAME=[...]
2
3 GH_USER=[...]
```

Now that we forked the Boot repo and we know how 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.

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

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.

```
1 cd environment-$CLUSTER_NAME-dev
2
```

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

The output is as follows.

```
1 cluster:
2   clusterName: ""
3   environmentGitOwner: ""
4   project: ""
5   provider: gke
6   zone: ""
7 gitops: true
8 environments:
9 - key: dev
10 - key: staging
11 - key: production
12 ingress:
13   domain: ""
14   externalDNS: false
15   tls:
16     email: ""
17     enabled: false
18     production: false
19 kaniko: true
20 secretStorage: local
21 storage:
22   logs:
23     enabled: false
24     url: ""
25   reports:
26     enabled: false
27     url: ""
28   repository:
29     enabled: false
30     url: ""
31 versionStream:
32   ref: "master"
33   url: https://github.com/jenkins-x/jenkins-x-versions.git
34 webhook: prow
```

As you can see, that file contains values in a format that resembles `requirements.yaml` 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.

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 my Gist 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) than 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`.

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

Next, we have `secretStorage` currently set to `vault`. 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`.

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 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.

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

As you already know, Prow 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).

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

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

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

```
1 cluster:
2   clusterName: "jx-boot"
3   environmentGitOwner: "vfarcic"
4   project: "devops-26"
5   provider: gke
6   zone: "us-east1"
```

```
7 gitops: true
8 environments:
9 - key: dev
10 - key: staging
11 - key: production
12 ingress:
13   domain: ""
14   externalDNS: false
15   tls:
16     email: ""
17     enabled: false
18     production: false
19 kaniko: true
20 secretStorage: vault
21 storage:
22   logs:
23     enabled: true
24     url: ""
25   reports:
26     enabled: true
27     url: ""
28   repository:
29     enabled: true
30     url: ""
31 versionStream:
32   ref: "master"
33   url: https://github.com/jenkins-x/jenkins-x-versions.git
34 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.

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`.

Off we go. Let's install Jenkins X.

```
1 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 is 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.

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.

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`.

```
1 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).

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.

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

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).

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.

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.

## Exploring The Changes Done By The Boot

Now, let's take a look at the changes Jenkins X Boot did to the local copy of the repository.

```
1 git --no-pager diff origin/master..HEAD
```

That's a long output, isn't it? Jenkins X Boot changed a few files. Some of those changes are based on our answers, while others are specific to the Kubernetes cluster and the vendor we're using.

We won't comment on all the changes that were done, but rather on the important ones, especially those that we might choose to modify in the future.

If you take a closer look at the output, you'll see that it created the `env/parameters.yaml` file. Let's take a closer look at it.

```
1 cat env/parameters.yaml
```

The output is as follows.

```
1 adminUser:  
2   password: vault:jx-boot/adminUser:password  
3   username: admin  
4 enableDocker: false  
5 pipelineUser:  
6   email: viktor@farcic.com  
7   token: vault:jx-boot/pipelineUser:token  
8   username: vfarcic  
9 prow:  
10  hmacToken: vault:jx-boot/prow:hmacToken
```

The `parameters.yaml` file contains the data required for Jenkins X to operate correctly. There are the administrative `username` and `username` and the information that Docker is not enabled (`enableDocker`). The latter is not really Docker but rather that the system is not using an internal Docker Registry but rather an external service. Further on, we can see the `email`, the `token`, and the `username` pipeline needs for accessing our GitHub account. Finally, we can see the `hmacToken` required for the handshake between GitHub webhooks and `prow`.

As you can see, the confidential information is not available in the plain-text format. Since we are using Vault to store secrets, some values are references to the Vault storage. That way, it is safe for us to keep that file in a Git repository without fear that sensitive information will be revealed to prying eyes.

We can also see from the `git diff` output that the `parameters.yaml` did not exist before. Unlike `requirements.yml` that existed from the start (and we modified it), that one was created by `jx boot`.

Which other file did the Boot process create or modify?

We can see that it made some modifications to the `jenkins-x.yml` file. We did not explore it just yet, so let's take a look at what we have right now.

```
1 cat jenkins-x.yml
```

The output is too long to be presented in a book, and you can see it on your screen anyway.

That is a `jenkins-x.yml` file like any other Jenkins X pipeline. The format is the same, but the content is tailor-made for the Boot process. It contains

everything it needs to install or, later on, upgrade the platform.

What makes this pipeline unique, when compared with those we explored earlier, is that the `buildPack` is set to `none`. Instead of relying on a buildpack, the whole pipeline is defined in that `jenkins-x.yml` file. Whether that's a good thing or bad depends on the context. On the one hand, we will not benefit from future changes the community might make to the Boot process. On the other hand, those changes are likely going to require quite a few other supporting files. So, the community decided not to use the buildpack. Instead, if you'd like to update the Boot process, you'll have to merge the repository you forked with the upstream.

Now, let's get to business and take a closer look at what's inside `jenkins-x.yml`.

We can see that it is split into two pipelines; `pullRequest` and `release`.

The `pullRequest` is simple, and it consists of a single `stage` with only one step. It executes `make build`. If you take a look at `env/Makefile` you'll see that it builds the charts in the `kubeProviders` directory, and afterward, it lints it. The real purpose of the `pullRequest` pipeline is only to validate that the formatting of the charts involved in the Boot process is correct. It is very similar to what's being done with pull requests to the repositories associated with permanent environments like staging and production.

The “real” action is happening in the `release` pipeline, which, as you already know, is triggered when we make changes to the master branch.

The `release` pipeline contains a single stage with the same name. What makes it special is that there are quite a few steps inside it, and we might already be familiar with them from the output of the `jx boot` command.

We'll go through the steps very briefly. All but one of those are based on `jx` commands, which you can explore in more depth on your own.

The list of the steps, sorted by order of execution, is as follows.

Step	Command	Description
validate-git	<code>jx step git validate</code>	Makes sure that the <code>.gitconfig</code> file is configured correctly so that Jenkins X can interact with our repositories

verify-preinstall	jx step verify preinstall	Validates that our infrastructure is set up correctly before the process installs or upgrades Jenkins X
install-jx-crds	jx upgrade crd	Installs or upgrades Custom Resource Definitions required by Jenkins X
install-velero	jx step helm apply	Installs or upgrades <a href="#">Velero</a> used for creating backups of the system
install-velero-backups	jx step helm apply	Installs or upgrades Jenkins X nginx Ingress implementation
install-nginx-controller	jx step helm apply	Installs nginx Ingress
create-install-values	jx step create install values	Adds missing values (if there are any) to the <code>cluster/values.yaml</code> file used to install cluster-specific charts
install-external-dns	jx step helm apply	Installs or upgrades the support for external DNSes
install-cert-manager-crds	kubectl apply	Installs or upgrades CertManager CRDs
install-cert-manager	jx step helm apply	Installs or upgrades CertManager in charge of creating Let's Encrypt certificates
install-acme-issuer...	jx step helm apply	Installs or upgrades CertManager issuer
install-vault	jx step boot vault	Installs or upgrades HashiCorp Vault
create-helm-values	jx step create values	Creates or updates the <code>values.yaml</code> file used by Charts specific to the selected Kubernetes provider
install-jenkins-x	jx step helm apply	Installs Jenkins X
verify-jenkins-x-env...	jx step verify	Verifies the Jenkins X environment
install-repositories	jx step helm apply	Makes changes to the repositories associated with environments (e.g., webhooks)
install-pipelines	jx step scheduler	Creates Jenkins X pipelines in charge of environments
update-webhooks	jx update webhooks	Updates webhooks for all repositories associated with applications managed by

webhooks	webhooks	Jenkins X
verify- installation	jx step verify install	Validates Jenkins X setup

Please note that some of the components (e.g., Vault) are installed, upgraded, or deleted depending on whether they are enabled or disabled in `jx-requirements.yml`.

As you can see, the process consists of the following major groups of steps.

- Validate requirements
- Define values used to apply Helm charts
- Apply Helm charts
- Validate the setup

The Helm charts used to set up the system are stored in `systems` and `env` directories. They do not contain templates, but rather only `values.yaml` and `requirements.yaml` files. If you open any of those, you'll see that the `requirements.yaml` file is referencing one or more Charts stored in remote Helm registries.

That's all we should know about the `jenkins-x.yml` file, at least for now. The only thing left to say is that you might choose to extend it by adding steps specific to your setup, or you might even choose to add those unrelated with Jenkins X. For now, I will caution against such actions. If you do decide to modify the pipeline, you might have a hard time merging it with upstream. That, by itself, shouldn't be a big deal, but there is a more important reason to exercise caution. I did not yet explain everything there is to know about Jenkins X Boot. Specifically, we are yet to explore Jenkins X Apps (not to be confused with Applications). They allow us to add additional capabilities (components, applications) to our cluster and manage them through Jenkins X Boot. We'll get there in due time. For now, we'll move to yet another file that was modified by the process.

Another file that changed is `jx-requirements.yml`. Part of the changes are those we entered, like `clusterName`, `environmentGitOwner`, and quite a few others. But, some of the modifications were done by `jx boot` as well. Let's take a closer look at what we got.

The output is as follows.

```
1 autoUpdate:
2   enabled: false
3   schedule: ""
4 bootConfigURL: https://github.com/vfarcic/jenkins-x-boot-config
5 cluster:
6   clusterName: jx-boot
7   environmentGitOwner: vfarcic
8   gitKind: github
9   gitName: github
10  gitServer: https://github.com
11  namespace: jx
12  project: devops-26
13  provider: gke
14  registry: gcr.io
15  zone: us-east1
16 environments:
17 - ingress:
18   domain: 34.74.196.229.nip.io
19   externalDNS: false
20   namespaceSubDomain: -jx.
21   tls:
22     email: ""
23     enabled: false
24     production: false
25   key: dev
26 - ingress:
27   domain: ""
28   externalDNS: false
29   namespaceSubDomain: ""
30   tls:
31     email: ""
32     enabled: false
33     production: false
34   key: staging
35 - ingress:
36   domain: ""
37   externalDNS: false
38   namespaceSubDomain: ""
39   tls:
40     email: ""
41     enabled: false
42     production: false
43   key: production
44 gitops: true
45 ingress:
46   domain: 34.74.196.229.nip.io
47   externalDNS: false
48   namespaceSubDomain: -jx.
49   tls:
50     email: ""
51     enabled: false
52     production: false
53 kaniko: true
54 repository: nexus
55 secretStorage: vault
56 storage:
57   backup:
58     enabled: false
59     url: ""
60   logs:
61     enabled: true
62     url: gs://jx-boot-logs-0976f0fd-80d0-4c02-b694-3896337e6c15
63 reports:
```

```

61      enabled: true
62      url: gs://jx-boot-logs-0976f0fd-80d0-4c02-b694-3896337e6c15
63  reports:
64    enabled: true
65    url: gs://jx-boot-reports-8c2a58cc-6bcd-47aa-95c6-8868af848c9e
66  repository:
67    enabled: true
68    url: gs://jx-boot-repository-2b84c8d7-b13e-444e-aa40-cce739e77028
69 vault: {}
70 velero: {}
71 versionStream:
72   ref: v1.0.214
73   url: https://github.com/jenkins-x/jenkins-x-versions.git
74 webhook: prow

```

As you can see both from `git diff` and directly from `jx-requirements.yml`, the file does not contain only the changes we made initially. The `jx boot` command modified it as well by adding some additional information. I'll assume that you do remember which changes you made, so we'll comment only on those done by Jenkins X Boot.



My output might not be the same as the one you're seeing on the screen. Jenkins X Boot might have features that did not exist at the time I wrote this chapter (October 2019). If you notice changes that I am not commenting on, you'll have to consult the documentation to get more information.

At the very top, we can see that `jx boot` added the `autoUpdate` section, which did not even exist in the repository we forked. That's normal since that repo does not necessarily contain all the entries of the currently available Boot schema. So, some might be added even if they are empty.

We won't go into `autoUpdate` just yet. For now, we're interested only in installation. Updates and upgrades are coming later.

Inside the `cluster` section, we can see that it set `gitKind` and `gitName` to `github` and `gitServer` to `https://github.com`. Jenkins X's default assumption is that you're using GitHub. If that's not the case, you can change it to some other Git provider. As I already mentioned, we'll explore that together with Lighthouse later. It also set the `namespace` to `jx`, which, if you remember, is the default Namespace for Jenkins X.

The `environments` section is new, as well. We can use one of its sub-entries to change Ingress used for a specific environment (e.g.,

Speaking of `ingress`, you'll notice that the `domain` entry inside it was auto-generated. Since we did not specify a domain ourselves, it used `.nip.io` in the same way we were using it so far. Ingress, both on the cluster and on the environment level, is a separate topic that we'll explore later.

Finally, we can see that it added values to `url` entries in `storage`. We could have created storage and set those values ourselves. But we didn't. We let Jenkins X Boot create it for us and update those entries with the URLs. We'll explore in more detail what we can do (if anything) with those storages. For now, just note that the Boot created them for us.

That's it. Those are all the files that were changed.

As you already saw, Jenkins X Boot run a pipeline that installed the whole platform. Given that we did not have Jenkins X running inside the cluster, that pipeline was executed locally. But, we are not expected to keep maintaining our setup by running pipelines ourselves. If, for example, we'd like to upgrade some aspect of the platform, we should be able to push changes to the repository and let the system run the pipeline for us, even if that pipeline will affect Jenkins X itself. The first step towards confirming that is to check which pipelines we currently have in the cluster.

## Verifying Jenkins X Boot Installation

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

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

The output is as follows.

```
1 - vfarcic/environment-jx-boot-dev/master
2 - vfarcic/environment-jx-boot-production/master
3 - 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.

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 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.

```
1 echo "A trivial change" \
2     | tee -a README.md
3
4 git add .
5
6 git commit -m "A trivial change"
7
8 git push
9
10 jx get activities \
11     --filter environment-$CLUSTER_NAME-dev \
12     --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
1 vfarhic/environment-jx-boot-dev/master #1	4m10s	3m52s	Succeeded
2 release	4m10s	3m52s	Succeeded
3 Credential Initializer 7jh9t	4m10s	0s	Succeeded
4 Working Dir Initializer Tr2wz	4m10s	2s	Succeeded
5 Place Tools	4m8s	2s	Succeeded
6 Git Source Vfarcic Environment Jx...	4m6s	36s	Succeeded https://
7 hub.com/vfarhic/environment-jx-boot-dev.git			
8 Git Merge	3m30s	1s	Succeeded
9 Validate Git	3m29s	1s	Succeeded
10 Verify Preinstall	3m28s	26s	Succeeded
11 Install Jx Crds	3m2s	10s	Succeeded
12 Install Velero	2m52s	12s	Succeeded
13 Install Velero Backups	2m40s	2s	Succeeded
14 Install Nginx Controller	2m38s	16s	Succeeded
15 Create Install Values	2m22s	0s	Succeeded
16 Install External Dns	2m22s	16s	Succeeded
17 Install Cert Manager Crds	2m6s	0s	Succeeded
18 Install Cert Manager	2m6s	16s	Succeeded
19 Install Acme Issuer And Certificate	1m50s	2s	Succeeded
20 Install Vault	1m48s	8s	Succeeded
21 Create Helm Values	1m40s	3s	Succeeded
22 Install Jenkins X	1m37s	1m3s	Succeeded
23 Verify Jenkins X Environment	34s	5s	Succeeded
24 Install Repositories	29s	5s	Succeeded
25 Install Pipelines	24s	1s	Succeeded
26 Update Webhooks	23s	4s	Succeeded
27 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.

did was push the change of 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.

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

```
1 kubectl get namespaces
```

The output is as follows.

```
1 NAME      STATUS AGE
2 cert-manager Active 28m
3 default    Active 74m
4 jx         Active 34m
5 kube-public Active 74m
6 kube-system Active 74m
7 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.

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

```
1 jx get env
```

The output is as follows.

The output is as follows.

1	NAME	LABEL	KIND	PROMOTE	NAMESPACE	REF	ORDER	CLUSTER	SOURCE
2						PR			
3	dev	Development	Development	Never	jx		0		<a href="https://git">https://git</a>
4	b.com/vfarcic/environment-jx-boot-dev.git					master			<a href="https://git">https://git</a>
5	staging	Staging	Permanent	Auto	jx-staging		100		<a href="https://git">https://git</a>
6	b.com/vfarcic/environment-jx-boot-staging.git					master			<a href="https://git">https://git</a>
7	production	Production	Permanent	Manual	jx-production		200		<a href="https://git">https://git</a>
8	b.com/vfarcic/environment-jx-boot-production.git					master			

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.

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.

```
1 cd ..
2
```

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.

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

```
1 jx get activity \
2   --filter jx-boot/master \
3   --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.

```
1 jx get activity \
2   --filter environment-$CLUSTER_NAME-staging/master \
3   --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.

```
1 kubectl get namespaces
```

The output is as follows.

```
1 NAME          STATUS AGE
2 cert-manager  Active 34m
3 default       Active 80m
4 jx            Active 40m
```

```
1 NAME           STATUS AGE
2 cert-manager   Active 34m
3 default        Active 80m
4 jx             Active 40m
5 jx-staging     Active 55s
6 kube-public    Active 80m
7 kube-system    Active 80m
8 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.

## What Now?

We did not dive into everything Jenkins X Boot offers. We still need to discuss how to use it to update or upgrade Jenkins X and the related components. There are also a few other features that we skipped. For now, I wanted you to get a taste of what can be done with it. We'll discuss it in more detail in the next chapter. Right now, you should have at least a basic understanding of how to install the platform using Jenkins X Boot. More will come soon.

Just like in the previous chapters, now you need to choose whether you want to move to the next chapter right away or you'd like to have a break. If you choose the latter, the Gists we used to create the cluster contain the instructions on how to destroy it as well. Once the cluster is no more, use the commands that follow to delete (most of) the repositories. The next chapter will contain an explanation of how to create everything we need from scratch.

```
1 hub delete -y \
2   $GH_USER/environment-$CLUSTER_NAME-staging
3
4 hub delete -y \
5   $GH_USER/environment-$CLUSTER_NAME-production
6
7 hub delete -y \
8   $GH_USER/jx-boot
```

You'll notice that we did NOT delete the `dev` repository used with Jenkins X Boot. We'll keep it because that'll simplify the installation in the next chapter. That's the beauty of GitOps. The Jenkins X definition is in Git, and it can be used or reused whenever we need it.

# Managing Third-Party Applications



The examples in this chapter should work with any Jenkins X (static and serverless), installed in any way (`jx create cluster`, `jx install`, Boot), and inside any Kubernetes flavor. That being said, as you will see later, some of the commands and instructions will be different depending on the installation type. More importantly, the outcome will differ significantly depending on whether you installed Jenkins X using the Boot or through `jx create cluster` or `jx install` commands.

So far, in most cases, our clusters had only Jenkins X. Few exceptions were the cases when we added Knative as a way to showcase its integration with Jenkins X for serverless deployments. Also, we added Istio and Flagger while we were exploring progressive delivery. Those were the only examples of system-wide third-party applications. We installed them using `jx create addon` commands, and we run those commands more or less “blindly”. We did not discuss whether that was a good way to install what we needed because that was not the focus at the time.

In the “real world” scenario, your cluster will contain many other third-party applications and platforms that you’ll have to manage somehow. Assuming that I managed to imprint into your consciousness the need to operate under GitOps principles, you will store the definitions of those additional applications in Git. You will make sure that Git triggers webhooks to the cluster so that pipelines can converge the actual into the desired state. To accomplish that, you’ll likely store the information in the declarative format, preferably YAML.

Now that we refreshed our memory on the goals we’re trying to accomplish, we can start discussing how to achieve them. How can we manage third-party applications without abandoning the principles on which we’re building processes around the life-cycle of our applications and the system as a whole? If we manage to figure out how to manage system-wide third-party applications, everything inside our cluster will be fully automated and reproducible. That’s the last piece of the puzzle.

Our applications (e.g., *go-demo-6*) and its direct dependencies (e.g., MongoDB) are managed by pipelines triggered by webhooks created whenever we push something to one of our Git repositories. Jenkins X itself is installed and managed by the Boot, which is yet another Jenkins X pipeline that uses definitions stored in Git.



Truth be told, you might not yet be using the Boot if it does not support your Kubernetes flavor. But, if that's the case, you will be using it later.

If Jenkins X and our applications and their direct third-party dependencies are managed by pipelines and adhere to the GitOps principles, the only thing we're missing is system-wide third-party applications. If we add them to the mix, the circle will close, and we'll be in full control of everything happening inside our clusters. We could also store cluster definitions in Git and automate its maintenance through pipelines, but, as I mentioned before, that is and will continue being out of the scope of the book. We're focused only on what is happening after a Kubernetes cluster is created.

Now that I repeated a few times the words “system-wide third-party” applications, it might help to clarify what I mean by that. I said “cluster-wide” because I'm referring to applications that are not directly related to one of those we're developing. As an example, a database associated with a single application does not fall into that category. We already saw how to treat those through examples with MongoDB associated with *go-demo-6*. As a refresher, all we have to do is add that application as a dependency in `requirements.yaml` inside the repository of our application.

The group we're interested in (system-wide third-party applications) contains those either used by multiple applications we're developing or those that provide benefits to the whole cluster. An example of the former could as well be a database that various applications of ours would use. On the other hand, an example of the latter would be, for example, Prometheus, which collects metrics from across the whole system, or Istio, which provides advanced networking capabilities to all those who choose to use it. All in all, system-wide third-party applications are those used by or providing services to the system as a whole or a part of it and are not directly related to any single application.

Now that I mentioned databases, Prometheus, and Istio as examples, I should probably commit to them as well. We'll use those to explore different techniques on how to manage system-wide third-party applications. As you will see soon, there is more than one way to tackle that challenge, and each might have its pros and cons or be used in specific situations.

That was enough of a pep talk. Let's jump into examples that will illustrate things much better than theory. As always, we need a cluster with Jenkins X before we proceed.

## Creating A Kubernetes Cluster With Jenkins X

If you kept the cluster from the previous chapter, you can skip this section. Otherwise, we'll need to create a new Kubernetes cluster with Jenkins X.



All the commands from this chapter are available in the [19-apps.sh](#) Gist.

For your convenience, the Gists with different cluster creation and Jenkins X installation combinations are provided below. Choose one that best fits your needs and run it as-is, or modify it to fit your use-case. Alternatively, feel free to create a cluster and install Jenkins X in any other way. At this point, you should be able to install Jenkins X, not the way I tell you, but in the way that best fits your situation, so Gists might not be needed at all. Nevertheless, they are here just in case. What matters is that you have a cluster with Jenkins X up-and-running.

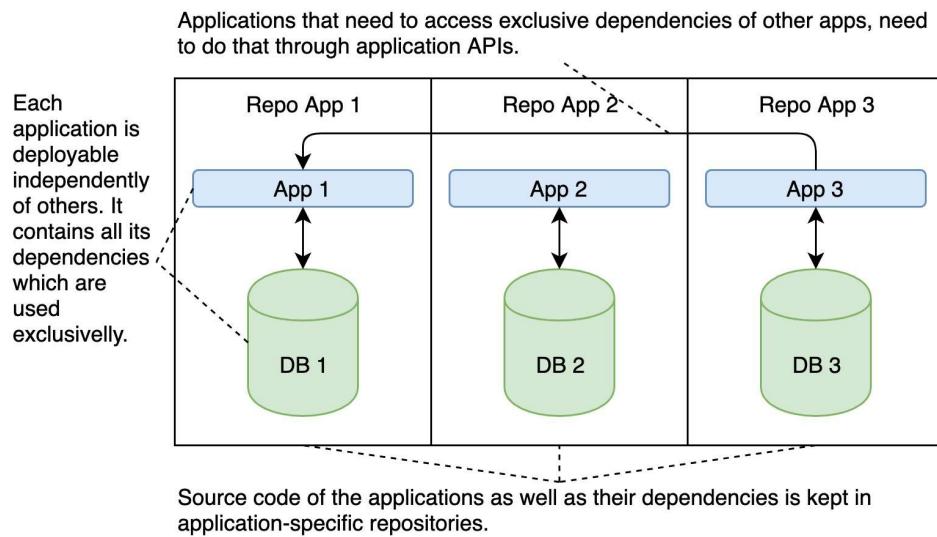
- Create a new serverless **GKE** cluster created **with the Boot**: [gke-jx-boot.sh](#)
- Create a new serverless **EKS** cluster created **without the Boot**: [eks-jx-serverless.sh](#)
- Create a new serverless **AKS** cluster created **without the Boot**: [aks-jx-serverless.sh](#)
- Use an **existing** serverless cluster created **without the Boot**: [install-serverless.sh](#)

Now we are ready to talk about installing, validating, and managing third-party applications.

## Managing Application-Specific Dependencies

Some third-party applications are used by a single application developed in-house. We already saw at least one such case in action when we used *go-demo-6*. It depends on Mongo DB. Given that the relation is one-on-one, the database is used by a single application. Other applications that might potentially need to access the data in that DB would need to go through *go-demo-6* API. That is the preferable model today. Using shared databases introduces too many problems and severely impacts our ability to iterate fast and deploy improvements only to a part of the system. That is especially evident in the microservices architecture.

One of the main characteristics of the microservices architecture is that they must be deployable independently from the rest of the system. If they would share the database with other applications, such independence would not be possible or be severely limited. For that reason, we moved to the model where an application is self-sufficient and contains all its dependencies. In such a scenario where a database and other dependencies are directly related to a single in-house application, the most logical and the most efficient place to define those dependencies is the repository of the application in question. Assuming that we are using Helm to package and deploy our applications, the dependencies are defined in `requirements.yaml`.



**Figure 19-1: Self-sufficient applications with exclusive dependencies**

In the diagram 19-1, each application has its own database or any other dependency. Everything that belongs to or is related to an application is

defined in the same repository. So, if we have three applications, we would have three repositories. If each of those applications has a database, it would be defined in the repository of the application that has it as the dependency.

Following the diagram, we can see that both App 1 and App 3 need to access the data from the same database (DB 1). Given that only App 1 can communicate with it directly, App 3 needs to go through App 1 API to retrieve from or write data to DB 1.

Such an approach allows us to have a separate team in charge of each of those applications. Those teams can deliver releases at their own speed without the fear that their work will affect others and without waiting for others to finish their parts. If we change anything in the repository of an application, those changes will be deployed as a preview environment. If we approve the pull request, a new release will be deployed to all permanent environments with the promotion set to *auto* (e.g., staging), and we'll be able to test the change inside the system and together with the other applications in it. When we're happy with the result, we can promote that specific version of the application to production or to any other environment with the promotion set to manual. In that case, it does not matter whether we change the code of our application, a configuration file, or a dependency like a database. Everything related to that application is deployed to one environment or another. Testing becomes easier while providing the mechanism to deliver responsive, scalable, highly-available, and fault-tolerant applications at a rapid pace.

**To summarize, third-party applications that are used exclusively by a single in-house application should be defined in the repository of that application.**

I do not believe that there is a reason to revisit the subject of defining exclusive (direct) dependencies. We went through it quite a few times, and, in case you are of a forgetful nature, you can refresh your memory by exploring `/charts/go-demo-6/requirements.yaml` file inside the repository that hosts `go-demo-6`. After all, this chapter is about installation and maintenance of system-wide third-party applications. Those that we are discussing now are not used by the system as a whole, but by a single application. Nevertheless, I thought that you do need a refresher of how we manage such dependencies given that everything else we'll discuss is

based on the same principles of dependency management, even though the techniques will vary.

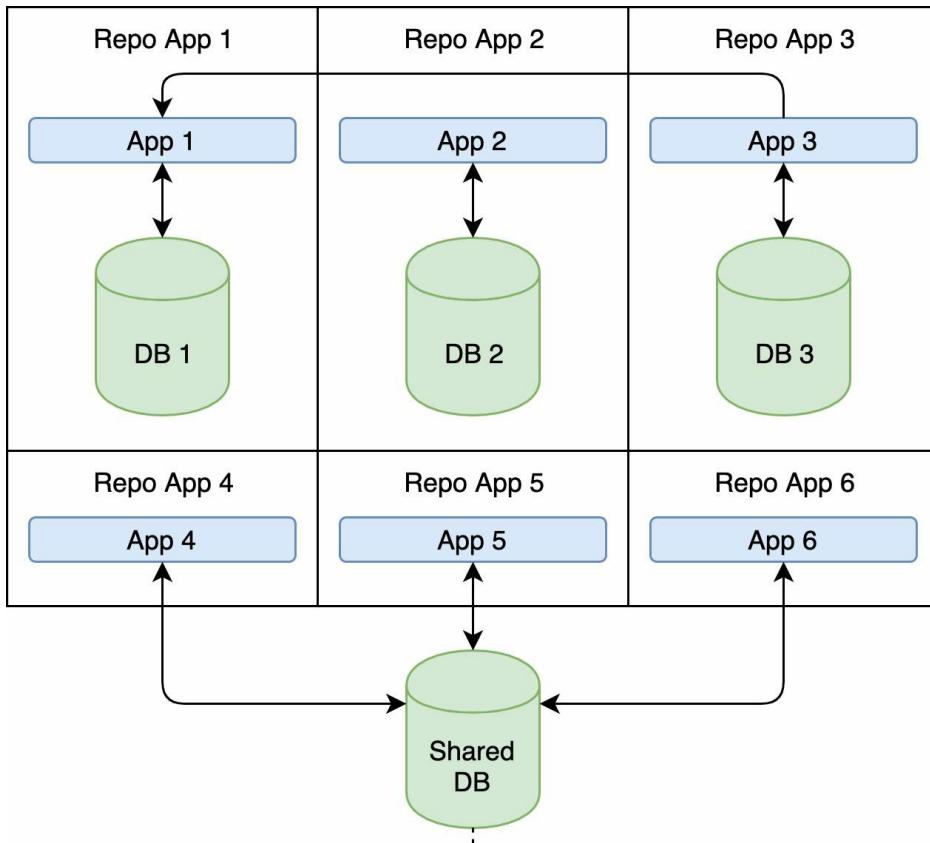
Let's jump into the cases that we did not discuss just yet in detail.

## **Managing Third-Party Applications Through Permanent Environments**

Imagine a scenario in which we have multiple applications using the same shared database. While I prefer the model in which a database is not shared, such a situation is all too common. Even if each application does have its own DB, the chances are that there will be some shared dependency. Maybe all your applications will publish events to the same queue. Or perhaps they will all use Redis to store cache. In most cases, there is always at least one shared dependency, and we need to figure out how to deal with such situations.

Let's expand on the previous example that contained three applications, each with its own DB. How would we define three additional apps if they would share the same database? To be more precise, where would we store the information about that DB?

It pretty clear that the code of each application and its exclusive dependencies should be in a separate repository to facilitate independence of the teams working on them as well as deployments. So, following the before-mentioned example, we would end up with six repositories for six applications. What is problematic is where to define the shared database.



The shared database does not belong exclusively to any single application so it cannot be defined in the repository of any specific app.

**Figure 19-2: Combining self-sufficient applications with those using shared third-party apps**

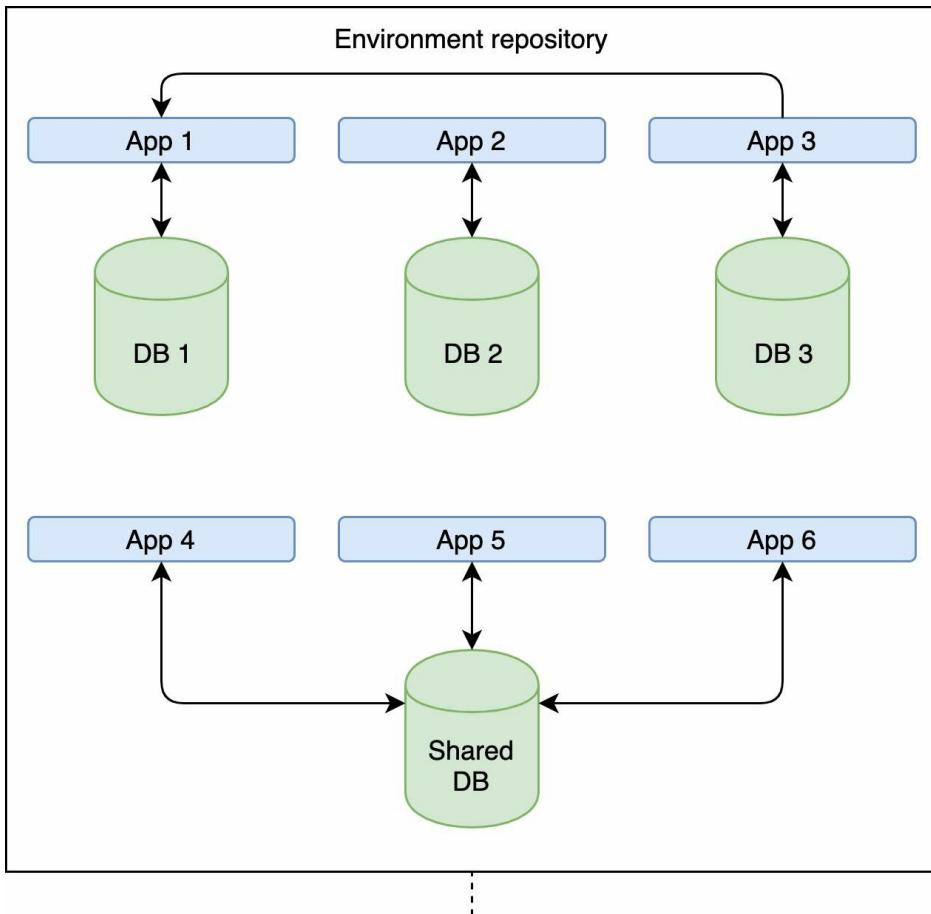
What we might be missing is the fact that each environment has its own Git repository. Unlike the repositories of the applications that contain code, configuration files, tests, and everything else related to them, environment repos contain only the references to the releases that should be running in those environments (Namespaces). Truth be told, there are other files in those repositories (e.g., pipeline definitions, Makefile, etc.), but they are not crucial for the point I'm trying to make.

Each application running in an environment is specified in the `requirements.yaml` file. If that's a reference to a release of our application, that release will be running in the environment. If that release has its own dependencies, they will run as well.

Here comes the important bit. If environment repositories contain references to some releases, nothing is telling us that those releases should be done by us. It could be, for example, a release of PostgreSQL created

by the community, or anyone else. Typically, only a fraction of what's running in our clusters was developed by us. Much of it is releases created by others. Right now, you are running a specific release of Kubernetes. On top of it is Jenkins X, which happens to be a bundle that includes many other third-party applications. Tomorrow you might add Istio, Prometheus, Grafana, and many other third-party applications. All those need to be specified somewhere and stored in Git if we are to continue following the GitOps principles. However, for now, we're focused on those that are directly used by our applications but not exclusively owned by any single one of them.

Given that everything is a release done by someone (us, a community, a vendor, etc.) and that a Git repository associated with an environment is a collection of references to those releases, it stands to reason that all we have to do is add shared third-party dependencies to those repos. The good news is that we already know how to do that, even though we might not have done it for such a specific purpose.



Environment repositories contain references to all the applications running in the associated Namespace. The whole environment is a list of dependencies which can be references to our applications, but also to shared dependencies.

**Figure 19-3: The whole environment defined in a Git repository**

Let's say that we have several applications that will all use a shared PostgreSQL and that, for now, we're focused on running those applications in staging. How would we do that?

The first step is to clone the staging environment repository. Given that we have quite a few variations of Jenkins X setup and I cannot be sure which one you chose, the address of the staging environment repo will differ. So, the first thing we'll do is define a variable that contains the environment prefix.



Please replace [...] in the command that follows with `jx-boot` if you used **Jenkins X Boot** to install Jenkins X. If that's not the case, use `jx-rocks` if you're running **static Jenkins X**, and `tekton` if it's **serverless**.

```
1 ENVIRONMENT=[...]
```

To be on the safe side, we'll remove the local copy of the repository just in case it is a left-over from previous chapters.

```
1 rm -rf environment-$ENVIRONMENT-staging
```

Now we can clone the repo.



Please replace [...] with your GitHub user in the commands that follow.

```
1 GH_USER=[...]
2
3 git clone \
4   https://github.com/$GH_USER/environment-$ENVIRONMENT-staging.git
5
6 cd environment-$ENVIRONMENT-staging
```

We cloned the repository and entered into the local copy.

Now let's take a look at the dependencies of our staging environment.

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

The output will differ depending on whether you created a new cluster for this chapter or you reused one from before. As a minimum, you should see two `exposecontroller` entries used by Jenkins X internally to create Ingress resources with dynamically created domains. You might see other entries as well, depending on whether there is something currently running in your staging environment.

Ignoring what is currently defined as the `dependencies` of our staging environment, what matters is that the `requirements.yaml` file is where everything related to that environment is defined. Since it is set to auto promotion, whenever we push something to the master branch of any of

the repositories with our applications, the result will be a new entry in that file. Or, if that application is already running in staging, the process will modify the `version`. That's what Jenkins X gives us out of the box, and it served us well. But now we're facing a different challenge. How can we add an application to staging that is not directly related to any single application we're developing? As you might have guessed, the solution is relatively simple.

Let's say that a few of our applications need to use PostgreSQL as a shared database. Given that we know that `requirements.yaml` contains the information of all the applications in that environment (namespace), all we have to do is add it as yet another entry.

If you Google "Helm PostgreSQL", the first result (excluding ads) will likely reveal that it is one of the charts in the official repository and that it is considered stable. That makes it easy. All we have to do is find the name of the chart (`postgresql`) and the release we want to use. We can see from `Chart.yaml` what the `version` of the latest release is. For our exercise, we'll use `5.0.0` even though the latest stable version is likely higher. This is not the production setup but rather an exercise, and it does not really matter which version we're running.

All in all, we'll add another entry to the `requirements.yaml` file.

```
1 echo "- name: postgresql
2   version: 5.0.0
3   repository: https://kubernetes-charts.storage.googleapis.com" \
4   | tee -a env/requirements.yaml
```

All that's left is to push the changes to GitHub. From there on, a webhook will notify Jenkins X that there is a change which, in turn, will create a new pipeline activity that will converge the actual into the desired state of the staging environment.

```
1 git add .
2
3 git commit -m "Added PostgreSQL"
4
5 git push
6
7 jx get activities \
8   --filter environment-$ENVIRONMENT-staging \
9   --watch
```

Once the activity is finished, the output of the activity should show that the status of all the steps is set to `succeeded`, and you can press `ctrl+c` to stop

watching.

Let's see what we got.

```
1 NAMESPACE=$ (kubectl config view \  
2   --minify \  
3   --output 'jsonpath={..namespace}')  
4  
5 kubectl \  
6   --namespace $NAMESPACE-staging \  
7   get pods,services
```

We retrieved the current Namespace (remember that it differs from one setup to another) and used that information to construct the Namespace where the staging environment is running. After that, we retrieved all the Pods and the Services. The output should be similar to the one that follows.

```
1 NAME          READY STATUS RESTARTS AGE  
2 pod/jx-postgresql-0 1/1   Running 0      7m9s  
3  
4 NAME          TYPE      CLUSTER-IP      EXTERNAL-IP PORT(S) AGE  
5 service/jx-postgresql     ClusterIP 10.23.240.243 <none>      5432/TCP 7m9s  
6 service/jx-postgresql-headless ClusterIP None        <none>      5432/TCP 7m9s
```

In your case, the output might be bigger and contain other Pods and Services. If that's the case, ignore everything that does not contain `postgresql` as part of the name.

We can see that we installed a specific release of PostgreSQL into the staging environment by adding a single entry to the `requirements.yaml` file. From now on, all the applications that should query or write data to this shared database can do so. Given that this is staging and not production, we can also experiment with what would happen if, for example, we change the version or any other aspect of it. That's what the staging environment is for. It allows us to test changes before applying them to production. It does not really matter whether those changes are related directly to our applications or to some of the third-party apps.

One thing that we did not do is customize PostgreSQL by changing one or more of its chart values. In the “real world” situation, we would likely not run PostgreSQL as-is, but tweak it to fit our needs. But, this is not the “real world”, so we'll skip customizing it assuming that you are familiar with how Helm values work. Instead, we'll assume that the default setup is just what we need in staging.

Using environment repositories to define non-exclusive dependencies of our applications has apparent benefits like automation through pipelines, change history through Git, reviews through pull requests (even though we skipped that part), and so on. Those are the same benefits that we saw countless times before throughout this book. What matters is that we have environment repositories to specify what should run in the associated Namespaces, no matter whether that's our application or one released by others. The key is that environment repositories are a great place to define applications that should run in a specific Namespace and are not used exclusively by a single application, but shared by many.

As you'll see later, there might be a better way to define applications that are not directly related to those we're developing. For now, we'll focus on the task at hand.

Now that PostgreSQL is running in staging, we can safely assume that it should be running in production as well. We need it in each permanent environment so that applications that need it can access it. So, let's add it to production as well.

```
1 cd ..
2
3 rm -rf environment-$ENVIRONMENT-production
4
5 git clone \
6   https://github.com/$GH_USER/environment-$ENVIRONMENT-production.git
7
8 cd environment-$ENVIRONMENT-production
```

Just as with staging, we deleted the local copy (leftover) of the production environment (if there was any). Then we cloned the repository, and entered inside the newly-created directory.

Next, we'll add PostgreSQL to `requirements.yaml` just as we did for staging.

```
1 echo "- name: postgresql
2   version: 5.0.0
3   repository: https://kubernetes-charts.storage.googleapis.com" \
4     | tee -a env/requirements.yaml
```

If we'd push the changes right away, Jenkins X would create a pipeline activity, and, soon after, PostgreSQL would run in production as well. That might compel you to ask, "why do we want to have the same thing defined twice?" "Why not keep it in the same repository, whichever it is?"

The truth is that even if an application is exactly the same in staging and production (or any other environment), sooner or later we will want to make a change to, for example, staging, test it, and apply the changes to production only if we're satisfied with the results. So, no matter whether PostgreSQL in those two environments is the same most of the time or only occasionally, sooner or later, we'll change some value or upgrade it, and that needs to be tested first.

There is another reason why it should run in at least two environments. Data usually needs to be different. More often than not, we do not run all the tests against production data. Some? Maybe. All? Never.

Anyways, there is yet another reason for having it in two repositories matching different environments (besides tweaking, testing, and upgrading). Unless you have money to waste, you will not run the same scale in staging as in production. The former is often limited to a single replica, or maybe a few for the sake of testing data replication. The production, on the other hand, can be any scale and is often much bigger. Even if both staging and production are using the same number of replicas, resource requests and limits are almost certainly going to differ. Production data needs more memory and CPU than the one used for testing.

No matter how much your setup differs (if at all), we'll imagine that our production needs to have more replicas. If for no other reason, that will allow me to refresh your memory about chart values.

All in all, we'll enable replication (scale) in production and leave staging running as a single replica database.

To enable replication, we need to figure out which values we should change. So, the next step is to inspect values.

```
1 helm inspect values stable/postgresql
```

The output, limited to the relevant parts, is as follows.

```
1 ...
2 replication:
3   enabled: false
4 ...
```

So, if we need replication in production, we need to set `replication.enabled` to `true`. That's easy. We'll just add it to `env/values.yaml`. There are already a few values there which I'll leave you exploring alone. We'll add those related to `postgresql` at the bottom by using `tee` with the argument `-a` (short for append).

```
1 echo "postgresql:
2   replication:
3     enabled: true" \
4   | tee -a env/values.yaml
```

Now that both the dependency and its values are defined, we can push the changes to GitHub and wait until Jenkins X does the job.

```
1 git add .
2
3 git commit -m "Added PostgreSQL"
4
5 git push
6
7 jx get activities \
8   --filter environment-$ENVIRONMENT-production \
9   --watch
```

Please wait until the newly created activity is finished executing and press `ctrl+c` to stop watching it.

Now comes the moment of truth. Was PostgreSQL deployed? Is it replicated? I'm sure that you know that the answer is yes, but we'll check it anyways.

```
1 kubectl \
2   --namespace $NAMESPACE-production \
3   get pods,services
```

The output is as follows.

```
1 NAME                  READY STATUS  RESTARTS AGE
2 pod/jx-postgresql-master-0 1/1   Running 0      51s
3 pod/jx-postgresql-slave-0 1/1   Running 0      51s
4
5 NAME                  TYPE      CLUSTER-IP      EXTERNAL-IP PORT(S) AGE
6 service/jx-postgresql   ClusterIP 10.100.242.238 <none>      5432/TCP 51s
7 service/jx-postgresql-headless ClusterIP None        <none>      5432/TCP 51s
8 service/jx-postgresql-read    ClusterIP 10.100.45.32  <none>      5432/TCP 51s
```

We can see by looking at the Pods that now we have two (`master` and `slave`). PostgreSQL was indeed deployed, and it is replicated, so adding it as a dependency and defining its values worked as expected.

That's it. There's not much more we can say about adding third-party applications to repositories associated with permanent environments. You already saw how they worked before, and this section made sure that you know that you can use them not only to promote your applications but also to add those released by others (third-party).

Before we move on, we'll remove PostgreSQL from both environments. That will demonstrate how to remove an application from an environment. But, you probably already know that. The more important reason we'll remove them lies in me being cheap. We will not need PostgreSQL any more, and I don't want you to use more resources in your cluster than necessary.

If we'd like to remove one of our (in-house) applications from a permanent environment, all we have to do is execute `jx delete application`. However, that won't work with PostgreSQL since it is not an application with a lifecycle managed by Jenkins X. Only quickstarts (`jx create quickstart`) and those we import (`jx import`) are considered Jenkins X applications. Still, knowing what `jx delete application` does should give us a clue what we can do without that command.

When we execute `jx delete application`, that command iterates through the repositories and removes the application entry from `jx-requirements.yaml`. In cases (like this one), when `jx delete application` is not an option, we can do the same operation ourselves.

Now, I could tell you to open `env/requirements.yaml` in your favorite editor and delete the `postgresql` entry. But, since I'm freak about automation and given that it is the last entry in that file, we'll accomplish the same with `sed`.

```
1 cat env/requirements.yaml \
2     | sed '$d' | sed '$d' | sed '$d' \
3     | tee env/requirements.yaml
```

The `sed '$d'` command removes the last line from an input. Since `postgresql` contains three lines, we repeated it three times, and we stored the result in `env/requirements.yaml`. The output of the command should show that `postgresql` is gone.

Please note that we did not remove the associated values, but only the dependency. While that might produce some confusion to those exploring

`env/values.yaml`, it is perfectly valid to leave `postgresql` in there. What truly matters is that it was removed as a dependency. If you feel like being pedantic, remove the values as well.

With the dependency gone, we can push the change to GitHub.

```
1 git add .
2
3 git commit -m "Removed PostgreSQL"
4
5 git push
```

We'll let Jenkins X take care of removing it from the production environment, while we turn our attention to staging.

```
1 cd ../environment-$ENVIRONMENT-staging
2
3 cat env/requirements.yaml \
4     | sed '$d' | sed '$d' | sed '$d' \
5     | tee env/requirements.yaml
6
7 git add .
8
9 git commit -m "Removed PostgreSQL"
10
11 git push
```

That's it. PostgreSQL was removed as a dependency from both production and staging repositories, and Jenkins X is running activities that will remove it from the cluster as well. Feel free to monitor those activities and to check the Pods in those Namespaces if you'd like to confirm that they were indeed removed. I'll leave that to you.

When should we use the repositories associated with permanent environments to define third-party applications?

One obvious use case for using the technique we just explored is for **applications that must run in those Namespaces but are not exclusively used by a single application of ours**. A typical example would be a shared database, but there can be many others. Generally speaking, whenever you need a third-party application that is used by at least two applications of yours, repositories associated with permanent environments are the right choice. But that's not all. In some cases, you'll have system-level third-party applications that are not directly used by any of your applications. Jenkins X would be one example, but we'll leave it aside from this discussion.

An example of an application that might benefit from being managed through environment repos is Prometheus. It collects metrics from across the whole cluster, but it is not a dependency of any one application. We could define it in the same way as we defined PostgreSQL. Depending on the level of confidence, we might want to run it only in production, or in staging as well. If we choose the latter, we would have a safe way to test new releases of Prometheus and promote them to production only if we are satisfied with the outcome of validations. We could choose to keep it in staging always, or only while testing new releases. No rule says that all the applications need to be always defined in all the environment repositories. All in all, **permanent environments are the right place for any type of third-party applications that need to exist in multiple environments, even for a short while.**

How about Istio? We cannot run it in both staging and production. There can be only one Istio installation per cluster. The same goes for many other applications, Knative being the first one that comes to my mind. In those cases, you might choose to define such applications only in the production repository. However, you would not be able to test new releases. If testing is a must (as it should be), you'd need to spin up a new cluster to test such applications like Istio and Knative. If you do create a separate cluster, you could define an environment that would point to that cluster, but that's not the subject we explored, so we'll ignore that option (for now) and I'll assume that you are either not testing new releases of cluster-wide applications that do not live exclusively in a single Namespace, or that you are using a separate cluster for that.

In any case, if you do have an application that cannot run multiple instances in the same cluster, you can define it in a single environment. But there's more to come, and soon I'll argue that those cases are better managed differently, at least for some of you.

Before we move on, let's go out of the local copy of the staging repository. We won't need it anymore.

```
1 cd ..
```

## Managing Third-Party Applications Running In The Development Environment



The examples in this sub-chapter work only if you installed the platform using **Jenkins X Boot**. If that's not the case, you might skip to the next section. But, before you jump, please consider reading what's in this sub-chapter. It might motivate you to switch to Jenkins X Boot, even if that means waiting for a while until the support for your favorite Kubernetes flavor comes along.

We saw a few ways to deal with third-party applications. But, so far, we did not touch any of those installed with Jenkins X Boot. As a matter of fact, we did not yet explore how they came into being. All we know is that the `dev` environment is running some applications, just as staging and production are running others. However, the knowledge that something is running is not good enough. We might need to be able to choose what is running there. We might want to remove some components or to change their behavior.

We'll start from the beginning and take a quick look at what we're currently running in our development environment.

```
1 kubectl get pods
```

The output will not show you anything new. Those are the same components that we've been using countless times before. They are all applications that form the Jenkins X bundle. So, why am I showing you something you already saw quite a few times?

All the components that are currently running in the development environment were installed as dependencies defined in `requirements.yaml`. Let's take a look at what's defined there right now.



Please replace `[...]` with the name of your cluster. If you used the gist to install Jenkins X using the Boot, the variable `CLUSTER_NAME` is already defined, and you can skip the first command from those that follow.

```
1 CLUSTER_NAME=[...]
2
3 cd environment-$CLUSTER_NAME-dev
4
5 cat env/requirements.yaml
```

We entered into the local copy of the `dev` repository and retrieved the contents of `env/requirements.yaml`. The output of the last command is as follows.

```
1 dependencies:
2 - name: jxboot-resources
3   repository: http://chartmuseum.jenkins-x.io
4 - alias: tekton
5   name: tekton
6   repository: http://chartmuseum.jenkins-x.io
7 - alias: prow
8   condition: prow.enabled
9   name: prow
10  repository: http://chartmuseum.jenkins-x.io
11 - alias: lighthouse
12   condition: lighthouse.enabled
13   name: lighthouse
14   repository: http://chartmuseum.jenkins-x.io
15 - name: jenkins-x-platform
16   repository: http://chartmuseum.jenkins-x.io
```

We can see that there are a few dependencies. There are `jxboot-resources`, `tekton`, `prow`, `lighthouse`, and `jenkins-x-platform`. If you take another look at the Pods, you'll notice that there is a mismatch. For example, there is no trace of `lighthouse` (whatever it is). On the other hand, we can see that Nexus is running even though it is not defined as a dependency. How can that be?

The fact that we have some dependencies defined does not necessarily mean that they are enabled. That is the case with `lighthouse`. It is defined as a dependency, but it is not enabled. Given that those dependencies are Helm charts, we can only guess that there is a file with values in which `lighthouse` is disabled. We'll explore soon where exactly is that value coming from.

As for Nexus, it is a sub-dependency of `jenkins-x-platform`. Just like `lighthouse`, it was enabled, and we can disable it if we choose to do so. It is only wasting resources since we won't use it in our examples, so disabling it will be our next objective. Through it, we'll have an opportunity to get more familiar with Jenkins X Boot environment and, in particular, with what's inside the `env` directory.

For now, what matters is that the `dev` repository used by Jenkins X Boot acts similarly as permanent repositories like staging and production. In both cases, there is a `requirements.yaml` file that defines the dependencies that are running in the associated Namespace. So, it stands to reason that if we want to change something in the development workspace,

all we have to do is modify that file. However, as you will see later, Jenkins X Boot extends Helm's capabilities and adds a few new tricks to the mix.

For now, before we start tweaking Jenkins X setup, let's confirm that Nexus is indeed running.

```
1 kubectl get ingresses
```

The output is as follows.

1 NAME	HOSTS	ADDRESS	PORTS	AGE
2 chartmuseum	chartmuseum-jx.35.229.41.106.nip.io	35.229.41.106	80	23m
3 deck	deck-jx.35.229.41.106.nip.io	35.229.41.106	80	23m
4 hook	hook-jx.35.229.41.106.nip.io	35.229.41.106	80	23m
5 jx-vault-jx-boot	vault-jx.35.229.41.106.nip.io	35.229.41.106	80	27m
6 nexus	nexus-jx.35.229.41.106.nip.io	35.229.41.106	80	23m
7 tide	tide-jx.35.229.41.106.nip.io	35.229.41.106	80	23m

As you can see, one of the Ingress resources is called `nexus`. Let's open it in a browser.

```
1 NEXUS_ADDR=$(kubectl get ingress nexus \
2   --output jsonpath=".spec.rules[0].host")
3
4 open "http://$NEXUS_ADDR"
```

You should see the Nexus home screen in your browser. It is indeed running, so let's see how we can remove it.

```
1 git pull
2
3 ls -l env
```

We pulled the latest version of the repo and retrieved the list of files and directories inside the `env` folder. The output is as follows.

```
1 Chart.yaml
2 Makefile
3 controllerbuild
4 controllerteam
5 controllerworkflow
6 docker-registry
7 jenkins
8 jenkins-x-platform
9 jxboot-resources
10 lighthouse
11 nexus
12 parameters.schema.json
13 parameters tmpl.schema.json
14 parameters.yaml
15 prow
16 requirements.yaml
```

```
17 tekton
18 templates
19 values tmpl.yaml
```

We can see that quite a few components are represented as directories inside `env`. You are probably familiar with most (if not all) of them. Since we are interested in Nexus or, to be more precise, in its removal, we should take a look inside the `env/nexus` directory. Given that Nexus is not defined separately in `env/requirements.yaml`, but it is part of the `jenkins-x-platform` dependency, there must be a value we can use to disable it. So, let's take a look at what's inside `env/nexus/values.yaml`.

```
1 cat env/nexus/values.yaml
```

The output is as follows.

```
1 enabled: true
```

If you are familiar with Helm, this is probably the moment you got confused. If the dependencies are defined in `env/requirements.yaml`, only `env/values.yaml` should be the valid place Helm would use in search of customization values. Helm ignores `values.yaml` in any other location. Therefore, it stands to reason that `env/nexus/values.yaml` would be ignored and that setting `enabled` to `false` in that file would accomplish nothing. On the other hand, you can rest assured that the file is not there by accident.

Jenkins X Boot extends Helm client capabilities in a few ways. In the context of values, it allows nested files. Instead of cramming `env/values.yaml` with deep-nested entries for all the applications, it allows us to use additional `values.yaml` files inside subdirectories. If we go back to the `env/nexus` directory, there is `values.yaml` file inside. At runtime, `jx` will read that and all other nested files and convert the values inside them into those expected by Helm.

On top of that, we can also use `values tmpl.yaml` files both in `env` as well as inside any of the product directories. That is also one of the Jenkins X additions to Helm. Inside `values tmpl.yaml` we can use Go/Helm templates. That is used heavily by Jenkins X to construct values based on secrets, but it can be used by you as well. Feel free to explore files with that name if you'd like to get more familiar with it.

All in all, we can define all values in `env/values.yaml`, or split them into separate files placed inside directories corresponding with dependencies. The files can be named `values.yaml` and hold only the values, or they can be `values.tpl.yaml`, in which case they can use templating as well.

Going back to the task at hand... We want to remove Nexus, so we'll modify the `env/nexus/values.yaml` file.

This is the moment we'd create a new branch and start making changes there. Later on, we'd create a pull request and, once it's merged to master, it would converge the actual into the desired state. But, in the interest of brevity, we'll work directly on the master branch. Don't take that as a recommendation, but rather as a way to avoid doing things that are not directly related to the subject at hand.

```
1 cat env/nexus/values.yaml \
2   | sed -e \
3     's@enabled: true@enabled: false@g' \
4   | tee env/nexus/values.yaml
```

We output the contents of `env/nexus/values.yaml`, changed the value of `enabled` to `false`, and stored the output back to `env/nexus/values.yaml`.

All that's left now is to let Jenkins X Boot do the work. We just need to initiate the process, and there are two ways to do that.

We can push changes to Git. That is the preferable way given that it complies with the GitOps principles. But, you might not be able to do that.

Webhook initiated pipeline associated with the `dev` repository works only if we're using Vault to store secrets. Otherwise, Jenkins X would not have all the information (passwords, API tokens) since they exist only on your laptop. So, the commands that follow will differ depending on whether you're using Vault or local storage for secrets. In the latter case, pushing to Git will not accomplish anything, and you'll have to execute `jx boot` from your laptop.



Please run the commands that follow only if you are using Vault to store secrets. If that's not the case, execute the `jx boot` command instead. I'll skip this warning from now on and assume that you'll run `jx boot` locally whenever you see the commands that push the changes to the `dev` environment.

```
1 git add .
2
3 git commit -m "Removed Nexus"
4
5 git push
6
7 jx get activity \
8   --filter environment-$CLUSTER_NAME-dev/master \
9   --watch
```

We pushed the changes to Git and watched the activities to confirm that the pipeline run was executed correctly. Once it's done, press `ctrl+c` to stop watching.

Let's take a quick look at Ingresses and see whether Nexus is still there.

```
1 kubectl get ingresses
```

The output, limited to the relevant parts, is as follows.

```
1 NAME    HOSTS          ADDRESS        PORTS AGE
2 ...
3 nexus   nexus-jx.35.229.41.106.nip.io 35.229.41.106 80      33m
4 ...
```

Ingress is still there. Does that mean that Nexus was not removed? Let's check it out.

```
1 open "http://$NEXUS_ADDR"
```

Nexus does not open in a browser. It was indeed removed, but Ingress is still there. Why is that?

Ingress is not part of the chart that deployed Nexus. It could have been if we added a few additional arguments to `values.yaml`. If we did that, it would be removed as well. But we let Jenkins X create Ingress for us instead of defining it explicitly, just as it does for other applications. Since only the components that were defined in the chart were removed, Ingress is still there. However, that's not an issue. Ingress definitions are not using

almost any resources. What matters is that everything else (Pods included) was removed. Nexus (except Ingress) is no more. If having Ingress bothers you, you can remove it separately.

I just said that everything (except Ingress) was removed. But you might not trust me, so let's confirm that's indeed true.

```
1 kubectl get pods
```

You will see all the Pods running in the `dev` Namespace. However, what you see does not matter. What matters, in this context, is what's not in front of you. There is no Nexus, so we can conclude that it is gone for good or, at least, until we change our mind and enable it by modifying `env/nexus/values.yaml` again.

So, when should we use Jenkins X Boot `dev` repository to control third-party applications by modifying what's inside the `env` directory? The answer, based on what we learned so far, is simple. **Change values in the `env` directory of the `dev` repo when you want to choose which components of Jenkins X platform to use.** You can tweak them further by adding the standard Helm chart values of those components inside `values.yaml` or `values.tpl.yaml` files.

As an alternative, you could even disable those components (e.g., Nexus) in the `dev` repository and define them yourself in, for example, the `production` repo. However, in most cases, that would not yield any benefits but only introduce additional complications given that those components are often tweaked to work well inside Jenkins X.

So far, we saw that we can define third-party applications as direct and exclusive dependencies of our apps, that we can manage them through the repositories associated with permanent environments (e.g., staging and production), and that we can control them through the `env` directory in the `dev` repository used by Jenkins X Boot. But, there's more, and we did not yet finish exploring the options. We are going to talk about Jenkins X Apps.

## Managing Third-Party Applications As Jenkins X Apps

Jenkins X supports something called `apps`, which are a cause for a lot of confusion, so I'll start by making sure that you understand that `apps` and

applications are not the same things. In the English language, `app` is an acronym of `application`, and many think (or thought) that in Jenkins X they are the same thing, just as repositories and repos are the same. But that's not the case, so let's set the record straight.

Jenkins X `applications` are those that you imported into the system, or you created as quickstarts. Those are your applications that are managed by Jenkins X. The `apps`, on the other hand, are Jenkins X extensions. They are a replacement for what is known as Jenkins X addons, and that might create even more confusion, so let's clarify that as well. Jenkins X `addons` are set to be deprecated in not so far future and will be replaced with `apps`. Even though both `addons` and `apps` are used to extend the components running inside the cluster, `apps` provide more features. We'll explore them next. For now, remember that `applications` and `apps` are different things and that `addons` will be (or already are) deprecated in favor of `apps`.

Jenkins X Apps are a way to install and maintain Helm charts in Jenkins X. Some of those charts are maintained by the Jenkins X community. However, who maintains the charts does not matter since we can convert any Helm chart into Jenkins X app. Now, if that's all that Jenkins X Apps are, there would be no good reason to use them. But there is more. Helm charts are only the base or, to be more precise, a mechanism to define applications. On top of Helm's standard features, Jenkins X Apps add a few more things that might help us simplify management of third-party applications.

Jenkins X Apps add the following capabilities on top of those you're already familiar with.

- The ability to interactively ask questions to generate `values.yaml` based on JSON Schema
- The ability to create pull requests against the GitOps repo that manages your team/cluster
- The ability to store secrets in Vault
- The ability to upgrade all apps to the latest version

At the time of this writing (October 2019), the following features are in the pipeline and are expected to be released sometime in the future.

- Integrating Kustomize to allow existing charts to be modified
- Storing Helm repository credentials in Vault

- Taking existing `values.yaml` as defaults when asking questions based on JSON Schema during app upgrade
- Only asking new questions during app upgrade
- The ability to list all apps that can be installed
- Integration for bash completion

We'll explore some, if not all, of those features soon. For now, let's take a quick look at the apps currently maintained by Jenkins X community.

```
1 open "https://github.com/jenkins-x-apps"
```

As you can see, there are quite a few Apps available. But don't get too excited just yet. Most of those are ports of Jenkins X Addons that do not fully leverage the concepts and features of the Apps. For now (October 2019), the “real” Apps are yet to come, and the most anticipated one is Jenkins X UI. We'll explore the UI in one of the next chapters, mostly because it is not yet public at the time I'm writing this. But it's coming...

So, we'll explore a better integration with Apps when we go through Jenkins X UI. For now, I only want to give you a taste, so we'll pick a random one to experiment with. Given that I'm in love with Prometheus, we'll use that one. So, let's take a quick look at what's inside the `jx-app-prometheus` App.

If you open the `jx-app-prometheus` repository, you'll see that it has the `jenkins-x.yml` file that defines the pipeline that will be executed as part of adding or modifying that App. Further on, inside the directory `jx-app-prometheus` (the same as the name of the repo), you'll see that there is a Helm chart. The `requirements.yaml` file inside it defines `Prometheus` as a dependency.

Judging by the contents of the repo, we can conclude that an App is a combination of a pipeline and a Helm chart that defines the application as a dependency.

Let's see it in action.

```
1 jx add app jx-app-prometheus
```

The outcome of that command will vary greatly depending on how you installed Jenkins X. We'll comment later on how that affects your decisions whether to use the Jenkins X Apps or not. For now, we're only

going to observe the outcome and maybe perform one or two additional actions.

If you did **NOT** use **Jenkins X Boot** to set up the platform, you'll get a message stating that it successfully installed `jx-app-prometheus` 0.0.3. That's it. There's nothing else left for you to do but wait until Prometheus is up-and-running. If you are NOT using Jenkins X Boot and you know that you never will, you can just as well skip the rest of the explanation. But that would be a mistake. I'm sure that, sooner or later, you will switch to the Boot process for installing and maintaining Jenkins X. That might not be today if, for example, your Kubernetes flavor is not supported, but it will be later. So, I strongly suggest you keep reading even if you won't be able to observe the same outcome from your system.

If you are **using Jenkins X Boot** and if your secrets are **stored in Vault**, the output should be similar to the one that follows.

1 Read credentials for <http://chartmuseum.jenkins-x.io> from vault helm/repos  
2 Created Pull Request: <https://github.com/vfarcic/environment-jx-boot-dev/pull/21>  
3 Added app via Pull Request <https://github.com/vfarcic/environment-jx-boot-dev/pull/21>

Instead of just installing the App, as we're used to with Jenkins X Addons, the system did a series of actions. It retrieved the credentials from Vault. More importantly, it created a pull request and placed the information about the new App inside it. In other words, it did not install anything just yet. Instead, by creating a PR, it gave us an opportunity to review the changes and choose whether we want to proceed or not.

Please open the pull request link in your browser and click the *Files changed* tab. You'll see that a few files were created or modified. We'll start at the top.

The first in line is the `env/jx-app-prometheus/README.MD` file. Jenkins X created it and stored the README through which we can get the humanly-readable information about the App we want to add to the system. Prometheus README does not seem to be very descriptive, so we'll skip over it. Later on, you'll see how README files can be much more expressive than that.

The next in line is `env/jx-app-prometheus/templates/app.yaml`. Unlike the README that does not serve any purpose but to provide information to other humans, the `app.yaml` file does serve a particular objective inside the

cluster. It defines a new Kubernetes resource called `App`. That, as you probably already know, does not exist in the default Kubernetes setup. Instead, it is one of the custom resources (CRDs) created by Jenkins X.

While `App` definitions can get complex, especially if there are many values and customizations involved, this one is relatively simple. It defines the repository of the chart (`jenkins.io/chart-repository`) and the name (`jenkins.io/app-name`) and the version (`jenkins.io/app-version`) of the App. Those are used by the system to know which Helm chart to install or update, and where to get it.

The two files we explored so far are located in the `env/jx-app-prometheus` directory. That follows the same logic as what we observed with Nexus. Each App is defined in its own directory that matches the name of the App. But those definitions are useless by themselves. The system needs to know which Apps we do want to run, and which are only defined but shouldn't be running at all. The file that ties it all together is `env/requirements.yaml`, which happens to be the file that was modified by the pull request.

If we take a look at the changes in `env/requirements.yaml`, we'll see that `jx-app-prometheus` was added together with the `repository` where the chart is stored as well as the `version` which we'd like to use.

Now, let's say that we finished reviewing the proposed changes and that we'd like to proceed with the process. Following the logic we used many times before, if we want to change the state of our cluster, we need to push or, in this case, merge the changes to the master branch. That's what we'll do next.

Please click the *Conversation* tab to get back to where we started. Assuming that you do indeed want to run Prometheus, click the *Merge pull request*, followed by the *Confirm Merge* button.

Once we merged the pull request, GitHub triggered a webhook that notified Jenkins X that there is a change. As a result, it created a new pipeline activity. Let's monitor it to confirm that it executed successfully.

```
1 jx get activity \
2   --filter environment-$CLUSTER_NAME-dev/master \
3   --watch
```

Press *ctrl+c* when the activity is finished.



This is the part that will be the same no matter whether you use Jenkins X Boot or not.

Let's take a look at the Pods to confirm that Prometheus is indeed running.

```
1 kubectl get pods \
2     --selector app=prometheus
```

The output is as follows.

1 NAME	READY	STATUS	RESTARTS	AGE
2 jenkins-x-prometheus-alertmanager-...	2/2	Running	0	16m
3 jenkins-x-prometheus-kube-state-metrics-...	1/1	Running	0	16m
4 jenkins-x-prometheus-node-exporter-...	1/1	Running	0	16m
5 jenkins-x-prometheus-node-exporter-...	1/1	Running	0	16m
6 jenkins-x-prometheus-node-exporter-...	1/1	Running	0	16m
7 jenkins-x-prometheus-pushgateway-...	1/1	Running	0	16m
8 jenkins-x-prometheus-server-...	2/2	Running	0	16m

The end result is the same no matter whether `jx add app` applied the chart directly, or it created a pull request. In both cases, Prometheus is up-and-running. However, the end result is not all that matters. The road often matters as much as the destination. Adding Jenkins X Apps without a `dev` repository used by Jenkins X Boot is not better than if we executed `helm apply`. In both cases, it would be a single command that changed the state of the cluster without storing the information in Git, without allowing us to review the changes, and so on and so forth. It's not much more than an ad-hoc command that lacks all the things we deem important in GitOps, and it does not have an associated, repeatable, and automated pipeline.

On the other hand, executing `jx add app` on top of Jenkins X installed with the Boot provides all the steps that allow us to continue following GitOps principles. It stored the information about the new app in an idempotent declarative format (YAML). It created a branch with the required changes, and it created a pull request. It allowed us to review the changes and choose whether to proceed or abort. It is recorded and reproducible, and it runs through a fully automated pipeline or, to be more precise, as part of the `dev` pipeline.

All that's left is to discover Prometheus' address and open it in browser to confirm that it is indeed accessible.

```
1 kubectl get ingresses
```

The output will differ depending on which Jenkins X flavor you’re running (static or serverless) and whether you have other applications running in that Namespace. Still, no matter which Ingress resources are in front of you, they all have one thing in common. There is no Prometheus.

If you ever applied the Prometheus Helm chart, you probably know that Ingress is disabled by default. That means that we need to set a variable (or two) to enable it. That will give us an excellent opportunity to explore how we can tweak Jenkins X Apps. As long as you remember that they are Helm charts (on steroids), all you have to do is explore the available variables and choose which ones you’d like to customize. In our case, we need to enable Ingress and provide at least one host. We could as well add labels to the Service that would tell Jenkins X that it should create Ingress automatically. But we’ll go with the first option and enable Ingress explicitly.

Given that you likely did not associate the cluster with a “real” domain, the first step is to find the IP of the cluster. With it, we’ll be able to construct a `nip.io` domain.

Unfortunately, the way how to discover the IP differs depending on whether you’re using EKS or some other type of the Kubernetes cluster.



Please execute the command that follows if you are **NOT running EKS**.

```
1 LB_IP=$(kubectl --namespace kube-system \
2     get service jxing-nginx-ingress-controller \
3     --output jsonpath=".status.loadBalancer.ingress[0].ip")
```



Please execute the command that follows if you are **running EKS**.

```
1 LB_HOST=$(kubectl --namespace kube-system \
2     get service jxing-nginx-ingress-controller \
3     --output jsonpath=".status.loadBalancer.ingress[0].hostname")
4
5 export LB_IP="$$(dig +short $LB_HOST \
6     | tail -n 1)"
```

Next, we'll output the load balancer IP to confirm whether we retrieved it correctly.

```
1 echo $LB_IP
```

The output should be an IP.

Now we need to get back to the differences between creating Jenkins X Apps with and without the `dev` repository.

If you did **not install Jenkins X using the Boot** and the platform definition is **not stored in the dev repo**, just add Ingress any way you like. Do not even bother trying to use YAML files or to extend the Prometheus Helm chart. That train has left the station the moment you executed `jx add app`. It was yet another undocumented command. As a matter of fact, do not use `jx add app` at all. If you're not yet convinced, I'll provide more arguments later. In any case, just as before, what follows next does not apply to you if you do not have the `dev` repository.

Now, if you do **use the dev repo** and you did **install the platform using Jenkins X Boot**, you should continue specifying everything in Git repositories. We already discussed how each app is in its own directory and how they are (extended) Helm charts. That means that we can simply add standard Helm `values.yaml` file and, in it, specify that we want to enable Ingress and which host it should respond to.

I'll save you from discovering the exact syntax of Prometheus Helm chart values by providing the command that defines what we need. But, before we do that, we'll pull the latest version of the `dev` repo. Remember that a pull request was created and merged to the master and that we do not have that latest version on our laptop.

```
1 git pull
```

Now let's get back to the values we need.

```
1 echo "prometheus:
2   server:
3     ingress:
4       enabled: true
5       hosts:
6         - prometheus.$LB_IP.nip.io" \
7 | tee env/jx-app-prometheus/values.yaml
```

We should probably create a new branch, create a pull request, review it, and merge it. However, given that you already know all that and that there is no need to rehearse it again, we'll push the change directly to the master branch.

```
1 git add .
2
3 git commit -m "Prometheus Ingress"
4
5 git push
6
7 jx get activity \
8   --filter environment-$CLUSTER_NAME-dev/master \
9   --watch
```

We pushed the changes to Git, and we started watching the activities to confirm that the process finished successfully. Once it's done, press *ctrl+c* to stop watching.

Did we get Ingress? Let's check it out.

```
1 kubectl get ingresses
```

The output should show that now we have, among others, Prometheus Ingress, and we should be able to access it from outside the cluster.

```
1 open "http://prometheus.$LB_IP.nip.io"
```

It works. We can see the Prometheus UI.

Now, we do not really need Prometheus. I used it only to demonstrate how to add one of the Apps defined by Jenkins X community. Since we won't use Prometheus in the rest of the exercises and since I'm committed not to waste resources without a good reason, we'll remove the Prometheus App we just added. Besides being cheap, that will also allow us to see how we can remove Apps and the process behind such destructive operations.

If you do NOT have the `dev` repository (if you did not use Jenkins X Boot to install the cluster), you'll have to specify the Namespace where the App is running. Otherwise, the system will assume that it is always in the `jx` Namespace.



Please execute the command that follows only if you used **Jenkins X Boot**, and you do have the `dev` repository.

```
1 jx delete app jx-app-prometheus
```



Please execute the command that follows only if you did **NOT use Jenkins X Boot**, and you do NOT have the `dev` repository.

```
1 jx delete app jx-app-prometheus \
2   --namespace $NAMESPACE
```



If you see `fatal: cherry-pick failed` errors, remove the local copy of the environment by executing `rm -rf ~/.jx/environments` and repeat the `jx delete app` command. It's a bug that will hopefully be fixed soon.

If you did **NOT use Jenkins X Boot**, GitOps principles do not apply to App, and all the associated resources were deleted right away. They were created without storing anything in Git, and now they were deleted in the same way.

On the other hand, if you did **use Jenkins X Boot**, the process for deleting an App is the same as for adding it. The system created a new branch, changed a few files, and created a new pull request. It's up to us to review it and decide whether to merge it to master. So, let's do just that.

If you're using the `dev` repository, please open the link to the newly created pull request and click the *Files changed* tab to review the proposed modifications. You'll see that the only file modified is `env/requirements.yaml`. The command removed the `jx-app-prometheus` dependency. It left the rest of the changes introduced by adding the App just in case we decide to add it back again. Those will be ignored since the dependency is removed.

Assuming that you are happy with the changes (they are simple after all), please select the *Conversation* tab, and click *Merge pull request*, followed by the *Confirm Merge* button. As always, that will trigger a webhook that will notify Jenkins X that there is a change in one of the repositories it manages. As a result, yet another pipeline activity will start, and we can watch the progress with the command that follows.

```
1 jx get activity \
2   --filter environment-$CLUSTER_NAME-dev/master \
3   --watch
```

Please press *ctrl+c* once the activity is finished.

No matter how you installed Jenkins X and whether you do or you don't have the `dev` repository, the end result is the same. Prometheus was removed from the cluster, and we can confirm that by listing all the Pods with the `app=prometheus` label.

```
1 kubectl get pods \
2   --selector app=prometheus
```

The output should show that no resources were found. Prometheus is no more.

## Using Any Helm Chart As A Jenkins X App

We saw that we can add an App that was created by the Jenkins X community. Be it as it may, most of the third-party applications you'll need do not exist as Jenkins X Apps. Or, to be more precise, are not the Apps maintained by the community. That might lead you to think that the Apps have very limited usefulness, but that would not be true. I did not yet tell you that any Helm chart can become an App. All we have to do is specify the repository where that chart is stored.

Let's say that we'd like to install Istio. You might be compelled to use the [jx-app-istio App](#), but that might not be the best idea, besides running a demo and wanting to set it up fast and without much thinking. A much better approach would be to use the "official" chart maintained by the Istio community. So, that's what we'll do next. It will be an opportunity to see how any chart can be converted into a Jenkins X App.

If you read [Istio documentation](#), you'll discover that two charts need to be installed; `istio-init` and `istio`. You'll also find out that the repository

where the charts are stored is available from <https://storage.googleapis.com/istio-release/releases/1.3.2/charts/>. Given than one Jenkins X App references one Helm chart, we'll need to add two Apps; one for `istio-init` and the other for `istio`. Equipped with that knowledge, we can add the first of the two Apps. The command is as follows.

```
1 jx add app istio-init \
2   --repository https://storage.googleapis.com/istio-release/releases/1.3.2/chart
```

Just as before, the output of that command will differ depending on whether you used Jenkins X Boot with the `dev` repo, or you didn't.

If you **do NOT have the `dev` repo**, the command will install the App directly inside the cluster. We already discussed briefly that such a practice is a bad idea (it's not GitOps). Just in case you skipped it, I will reiterate my previous statement. **If you did not use Jenkins X Boot to install the platform and you do not have the `dev` repository, do NOT use Jenkins X Apps, unless you have to (e.g., for Jenkins X UI)**. Instead, add third-party applications as dependencies to the repository associated with the production (or any other) environment. The only reason I'm showing you the Apps is so that you don't feel left behind.

If you **do have the `dev` repo**, the process for adding an App based on a Helm chart is no different than when adding one of the Apps maintained by the Jenkins X community.

You should see the link to the newly created pull request. Open it and click the *Files changed* tab so that we review the suggested changes.

By adding `istio-init`, the same files changed as with Prometheus, except that two (of three) are in different directories.

The `env/istio-init/README.MD` file contains information about the App and the whole README from the original chart. Next, we have `env/istio-init/templates/app.yaml` that is the definition of the App, with the information about the repository `jenkins.io/chart-repository`, the name of the chart (`jenkins.io/app-name`), and the version(`jenkins.io/app-version`). Finally, `istio-init` was added together with other dependencies in `env/requirements.yaml`.

As you can see, it does not matter much whether an App was added from the catalog of those supported by the Jenkins X community, or from any available Helm chart. In all the cases, it is based on a Helm chart, and as long as Jenkins X has the information about the name, version, and the repository where the chart resides, it will convert it into an App.

To finish the process, please select the *Conversation* tab, and click *Merge pull request*, followed with the *Confirm Merge* button. As you already know, that will trigger a webhook that will notify the cluster that there is a change in one of the repositories and, as a result, a new pipeline activity will be created.

```
1 jx get activity \
2   --filter environment-$CLUSTER_NAME-dev/master \
3   --watch
```

Press *ctrl+c* to stop watching the activity once it is successfully finished.

Finally, we'll confirm that `istio-init` was installed correctly by outputting the Custom Resource Definitions (CRDs) that contain `istio.io` in their name. If you're not familiar with Istio, all that `istio-init` does is install those CRDs. The rest of the setup comes afterward.

```
1 kubectl get crds | grep 'istio.io'
```

Unless Istio changed since the time I wrote this (October 2019), there should be twenty-three CRDs in the output, and we can conclude that the first part of the Istio setup was done correctly.

That's it. You saw how you can create Jenkins X Apps through the `jx add app` command. We also explored how those Apps can be updated or removed. If you're using the `dev` repository, you saw some of the benefits of the Apps, mainly their support for GitOps processes. Every time an App is added or removed, `jx` creates a new branch and a pull request, and it waits for you to review and approve the changes by merging it to the master.

In some cases, however, you might want to skip reviewing and merging pull requests. You might want to let Jenkins X do that for you, as well. In such cases, you can add the `--auto-merge` argument.



The `--auto-merge` argument might not work due to the [issues 5761](#). Feel free to monitor it to see whether it was resolved.

You should understand that `jx add app` and `jx delete app` commands are only manipulating files in the `dev` repository and pushing them to Git. Everything else is done by Jenkins X running in the cluster. That means that you do not have to use those commands. Think of them more as “helpers” than as requirements for working with Apps. We can accomplish the same without them. We can create the files we need and push them to Git. As a result, a new App will be added without us executing any command (excluding `git`).

We still need to apply the second chart (`istio`), so we’ll use that as an excuse to try to add an App without executing `jx` commands.



I did not even bother adapting the commands that follow for those **not using the `dev` repository**. We already concluded that the Apps based on charts not maintained by the Jenkins X community are not worth the trouble. If you did not install your cluster using Jenkins X Boot, you’ll be better off adding dependencies to the repositories associated with the permanent environments (e.g., staging and production). As a matter of fact, you’d be better off even with an ad-hoc `helm apply` command.

Since we are about to create and modify a few files in the local copy of the `dev` repository, we should start by pulling the latest codebase from GitHub.

```
1 git pull
```

Now that we have a local copy of the latest version of the repository, we can create a new App. Remember, this time, we’re exploring how to do that by creating the files ourselves instead of using the `jx add app` command.

We can approach this challenge from two directions. One option could be to create all the files from scratch. The other is to copy a directory of one of the existing Apps and modify it to suit our needs. We’ll go with the second option since it is probably an easier one. Given that we already

have the App that's using `istio-init`, its files are probably the best candidate to be used as the base for `istio`.

```
1 cp -r env/istio-init env/istio
```

Now that we copied the `istio-init` directory as `istio`, all we have to do is change a few files. We'll skip modifying the `README`. It is important only for humans (we might read it), but it plays no role in the process. In the "real world" situations, I'd expect you to change it as well. But since this is not the "real world" but rather a learning experience, there's no need for us to spend time with it.

There are three files that we might need to change. We might create `env/istio/templates/values.yaml` if we'd like to change any of the chart's default values. We'll skip that one because `istio` is good as-is. Instead, we'll focus on the other two files.

```
1 cat env/istio/templates/app.yaml
```

That's the definition of the App we're about to add to the cluster. It is a copy of `istio-init`, so all we need to do is change the `jenkins.io/app-name` and `name` values from `istio-init` to `istio`. We'll also change `jenkins.io/chart-description`. It serves only informative purposes. But, since we're nice people and don't want to confuse others, changing it might provide additional clarity to whoever might explore it later.

The command that should make those changes is as follows.

```
1 cat env/istio/templates/app.yaml \
2   | sed -e 's@istio-init@istio@g' \
3   | sed -e \
4     's@initialize Istio CRDs@install Istio@g' \
5   | tee env/istio/templates/app.yaml
```

The definition of an App is useless by itself. Its existence will not result in it running inside the cluster. We need to add it as yet another dependency in `env/requirements.yaml`. So, let's take a quick peek at what's inside it.

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

The output is as follows.

```
1 dependencies:
2 - name: jxboot-resources
3   repository: http://chartmuseum.jenkins-x.io
4 - alias: tekton
```

```

5   name: tekton
6   repository: http://chartmuseum.jenkins-x.io
7 - alias: prow
8   condition: prow.enabled
9   name: prow
10  repository: http://chartmuseum.jenkins-x.io
11 - alias: lighthouse
12   condition: lighthouse.enabled
13   name: lighthouse
14   repository: http://chartmuseum.jenkins-x.io
15 - name: jenkins-x-platform
16   repository: http://chartmuseum.jenkins-x.io
17 - name: istio-init
18   repository: https://storage.googleapis.com/istio-release/releases/1.3.2/charts/
19   version: 1.3.2

```

All but the last dependency are those of the system at its default configuration. Later on, we used `jx add app` to add `istio-init` to the mix. Now we're missing an entry for `istio` as well. The `repository` and the `version` are the same, and the only difference is in the `name`.

```

1 echo "- name: istio
2   repository: https://storage.googleapis.com/istio-release/releases/1.3.2/charts/
3   version: 1.3.2" \
4 | tee -a env/requirements.yaml

```

All that's left is to push the changes to GitHub and let the system converge the actual into the desired state, which we just extended with an additional App. Normally, we'd create a branch, push the changes there, create a pull request, and merge it to the master branch. That would be the correct way to handle this or any other change. But, in the interest of time, we'll skip that with the assumption that you already know how to create PRs. If you don't, you're in the wrong industry.

```

1 git add .
2
3 git commit -m "Added Istio"
4
5 git push
6
7 jx get activity \
8   --filter environment-$CLUSTER_NAME-dev/master \
9   --watch

```

We committed and pushed the changes to the master branch and started watching the activities to confirm that the changes are applied to the cluster. Once the new activity is finished, please press `ctrl+c` to stop watching.

Istio should be fully up-and-running. We can confirm that by listing all the Pods that contain `istio` in their names.

```
1 kubectl get pods | grep istio
```

This is neither the time nor the place to dive deeper into Istio. That was not the goal. I used it only as an example of different ways to add Jenkins X Apps to the system.

Speaking of the Apps, let's see which ones are currently running in the cluster. You already saw that `jx` has a helper command for almost anything, so it should be no surprise to find out that retrieving the `apps` is available as well.

```
1 jx get apps
```

The output is as follows, at least for those who installed Jenkins X using the Boot.

```
1 Name      Version Chart Repository
2 Namespace Status          Description
3 istio-init 1.3.2  https://storage.googleapis.com/istio-release/releases/1.3.2/cha
4 s/           READY FOR DEPLOYMENT Helm chart to initialize Istio CRDs
5 istio       1.3.2  https://storage.googleapis.com/istio-release/releases/1.3.2/cha
6 s/           READY FOR DEPLOYMENT Helm chart to install Istio
```

That's it. We explored a few commonly used ways to add, manage, and delete Jenkins X Apps. We'll have a short discussion around them soon. For now, we'll remove `istio` and `istio-init` since we do not need them anymore.

```
1 git pull
2
3 jx delete app istio
```

You know what to do next. Merge the PR so that the change (`istio` removal) is applied to the system. We can see that through the proposed changes to the `env/requirements.yaml` file.

You'll notice that the `jx delete app` command works no matter whether the App was added through `jx add app` or by fiddling with the files directly in Git. It always operates through Git (unless you are NOT using the `dev` repo).

The next in line for elimination is `istio-init`, and the process is the same.



If you added the App without the `dev` repository created by Jenkins X Boot, you'll need to add the `--namespace $NAMESPACE` argument to the command that follows.

```
1 git pull
2
3 jx delete app istio-init
```

I'll leave you to do the rest yourself (if you're using the Boot). Merge that PR!

That's it. You learned the basics of extending the Jenkins X platform by adding Apps. As a matter of fact, it's not only about extending Jenkins X but more about having a reliable way to add any third-party application to your cluster. However, not all are equally well suited to be Jenkins X Apps.

Jenkins X Apps are beneficial for at least two scenarios. **When we want to extend Jenkins X, adding Apps is, without a doubt, the best option.** We'll explore that later on (in one of the next chapters) when we dive into Jenkins X UI. Given that the Apps can be any Helm chart, we can convert any application to be an App.

Besides those designed to extend Jenkins X, excellent candidates are the charts that do not need to be in multiple repositories. For example, if we'd like to run two instances of Prometheus (one for testing and the other for production), we're better off adding them to the associated permanent environment repositories. However, many are not well suited to run in testing or are not worth validating. Prometheus might be such a case. If we upgrade it and that turns out to be a bad choice, no harm will be done to the cluster. We might not be able to retrieve some metrics, but that would be only temporary until we roll back to the previous version. The exception would be if we hook HorizontalPodAutoscaler to Prometheus metrics, in which case testing it before deploying a new version to production is paramount. So, **when applications should run only in production (without a second instance used for testing), Apps are a better way to manage them due to a few additional benefits they provide.**

At the core, Jenkins X Apps follow the same GitOps principles as any other. Their definitions are stored in Git, we can create pull requests and review changes, and only a merge to the master branch will change the state of the cluster. Using Apps is not much different from defining dependencies in staging, production, or any other permanent environment repository. What makes them “special” is the addition of a few helper features and a few conventions that make management easier. We have a better-defined pipeline. Branches and pull requests are created automatically. Secrets are stored in Vault. Dependencies are better organized. And so on, and so forth. We explored only a few of those features. Later on, we’ll see a few more in action, and the community is bound to add more over time.

The previous two paragraphs make sense only if you used Jenkins X Boot to install the platform and if there is the `dev` repository. If that’s not the case, Jenkins X Apps do not provide almost any benefits, outside those defined by the community (e.g., UI). Do not even think about using the Apps to install Prometheus, Istio, or any other application available as a Helm chart. A much better strategy is to modify the repositories associated with permanent environments (e.g., staging, production). That way, definitions will be stored in Git repos. Otherwise, `jx add app` is yet another helpful command that results in action that did not pass through Git.

## Which Method For Installing and Managing Third-Party Applications Should We Use?

We saw a few ways how we can install and manage third-party applications inside our clusters. Having more than one choice can be confusing, so let’s try to summarize some kind of rules you can use as guidelines towards making a decision. But, before we do that, let’s repeat the rule to rule them all. **Do not make any modification to the state of a cluster without pushing a change to a Git repository.** That means that executing `jx add app` is not a good idea if you do not have the repository associated with the `dev` environment. The same rule applies to any other command that changes the state of the cluster without storing the information about the change in a Git repository. In other words, Git should be the only actor that can change the state of your cluster.

Now that we reiterated the importance of following the GitOps principles and excluded Jenkins X Apps without the `dev` repository from the equation, we can go over the rules.

1. A third-party application that is a dependency of a **single in-house application** should be defined in **the repository of that application**.
2. A third-party application that is a dependency of **multiple in-house applications** and might need to **run in multiple environments** (e.g., staging, production) should be defined in **repositories associated with those environments**.
3. A third-party application that is a dependency of **multiple in-house applications** and **does NOT need to run in multiple environments** should be defined as an **App stored in the dev repository**.
4. A third-party application that is **used by the system as a whole**, and that **does not need to run in multiple environments** should be defined as an **App stored in the dev repository**.

That's it. There are only four rules that matter when thinking where to define third-party applications. However, sometimes, it is easier to remember examples instead of rules, so I'll provide one for each.

An example of the **first rule** is a database used by a single application. Define it in the repo of that application.

An example of the **second rule** is a database used by multiple applications. Define it in all the repositories associated with permanent environments where you run those applications (e.g., staging, production).

An example of the **third rule** is Prometheus that often runs in a single environment and with no in-house application to depend on it. Define it as a Jenkins X App in the `dev` repository.

An example of the **fourth rule** is Istio that is a cluster-wide third-party application or, to be more precise, a system (Kubernetes) component. Define it as a Jenkins X App in the `dev` repository.

## What Now?

You know what comes next. You can delete the cluster and take a break, or you can jump right into the next chapter if there is any.

If you used `jx boot` to create a cluster, you are currently inside the local copy of the `dev` repository, so we'll have to go out of it.

```
1 cd ..
```

Next, we'll delete all the repositories used in this chapter, except for `dev` that can be reused.

```
1 rm -rf environment-$ENVIRONMENT-staging
2
3 hub delete -y \
4   $GH_USER/environment-$ENVIRONMENT-staging
5
6 rm -rf environment-$ENVIRONMENT-production
7
8 hub delete -y \
9   $GH_USER/environment-$ENVIRONMENT-production
```

Finally, delete the cluster. You can find the instructions at the end of the Gist you used at the beginning of the chapter. They are near the bottom.

## Now It's Your Turn

I showed a very opinionated way of doing continuous delivery and continuous deployment using Jenkins X in a Kubernetes cluster. Unlike **The DevOps 2.4 Toolkit: Continuous Deployment To Kubernetes**, this time opinions did not come from me, but rather the platform itself. Unlike the “traditional” Jenkins, Jenkins X dictates how you should do continuous delivery or continuous deployment. It can be tweaked, but you do need to comply with some, if not all, of its processes. GitOps, for example, is a must. Without it, you’d lose too many of the great features it offers.

At the time of this writing (October 2019), Jenkins X is still in its early phases, and we can expect that great new features are coming. I’ll do my best to keep this book up-to-date. For that reason, I hope you bought the eBook edition so that you can download updated versions. If you did, please keep in mind that I am in full control only of the editions in LeanPub.com and Amazon. I do not control other vendors so I cannot guarantee whether they will upload more recent versions.

Assuming that you did not skip directly to this section, you have an excellent foundation with Jenkins X. Now it’s up to you to implement it in your organization and put it to the only real test by running it in production. That is, if I managed to convince you that Jenkins X is the best way to implement continuous delivery inside your Kubernetes clusters.

That being said, I’d love to hear about your experience. Are you using Jenkins X? Did you create a custom build pack? How did you onboard teams? Any information about the way you use it and your experience with Jenkins X will help us improve the project. Please send me what you created. I want to see your processes and your pipelines. Even more, I’d love to work with you to publish them in a blog or as an appendix to this book. If you think you did something interesting, as I’m sure you did, please contact me on [DevOps20](#) Slack and show me what you created.

**You learned from others, now it’s time for others to learn from you. I’m done explaining what I did, now it’s your turn to share what you made.**

# Contributions

Like the previous books, this one was also a collaboration effort. Many helped shape this book through discussions, notes, and bug reports. I was swarmed with comments through [DevOps20](#) Slack (often private) messages and emails. The conversations I had with the readers of the early editions of the book influenced the end result significantly. I'm grateful to have such a great community behind me. **Thank you for helping me make this book great.**

A few rose above the crowd.

**Carlos Sanchez** was so kind as to contribute parts of the **Choosing The Right Deployment Strategy** chapter. He's one of those people that you MUST follow on social media. His articles are excellent, and he is one of the most gifted speakers I ever saw.

In his own words...

*I have been working for over 15 years in software automation, from build tools to Continuous Delivery and Progressive Delivery. In all this time I have been involved in Open Source, as a member of the Apache Software Foundation and contributing to a variety of popular projects such as Apache Maven or Jenkins. As I started playing with Kubernetes during its early days, I created the [Jenkins Kubernetes plugin](#) and recently started the [Progressive Delivery and Canary deployment implementation for Jenkins X](#).*

**Joost van der Griendt** is one of those people that embrace a challenge wholeheartedly. We worked together for the same customer and quickly established a great relationship. From there on, he started helping me with discussions and advice on the subjects I worked on. Later on, he began contributing to this book. At times he was so proactive and helpful that I could not keep up with his pull requests. Just when I would think that I'm done for the day, I'd see a new pull request with more questions, changes, additions, and corrections.

In his own words...

*Joost started out a Java backend developer but found himself fascinated by developer productivity through tools like Jenkins and Maven. Inspired by Viktor's books and Workshops, he moved his attention to Docker, Cloud, and DevOps to the point where he joined CloudBees to help spread the knowledge.*

**Darin Pope** was continually sending pull requests with corrections and suggestions. He made this book much clearer than it would be if I had to rely on my, often incorrect, assumptions of what readers expect. Without him, my “broken” English would be even more evident. Outside of the help with this book, together we co-host [DevOps Paradox podcast](#). Please check it out.

In his own words...

*I started out working with MultiValue database platforms in the late 80s. Since that time, I've been fortunate to work with a lot of great clients and companies using numerous technologies. Said differently, I'm old, and I've seen a lot. As we head into the new roaring 20s, I'm really looking forward to what the future holds, especially in the IoT space.*

**Eric Tavela** provided technical feedback to help keep the text up to date with the evolving Jenkins X platform.

In his own words...

*Working in the field of software engineering for over 25 years, I've continually been trying to balance my knowledge of effective coding practices, delivery infrastructure and software development process. Nowadays that generally means Agile BDD of microservices deployed to Kubernetes; tomorrow, who can say? For however long I can keep up, I look forward to finding out.*