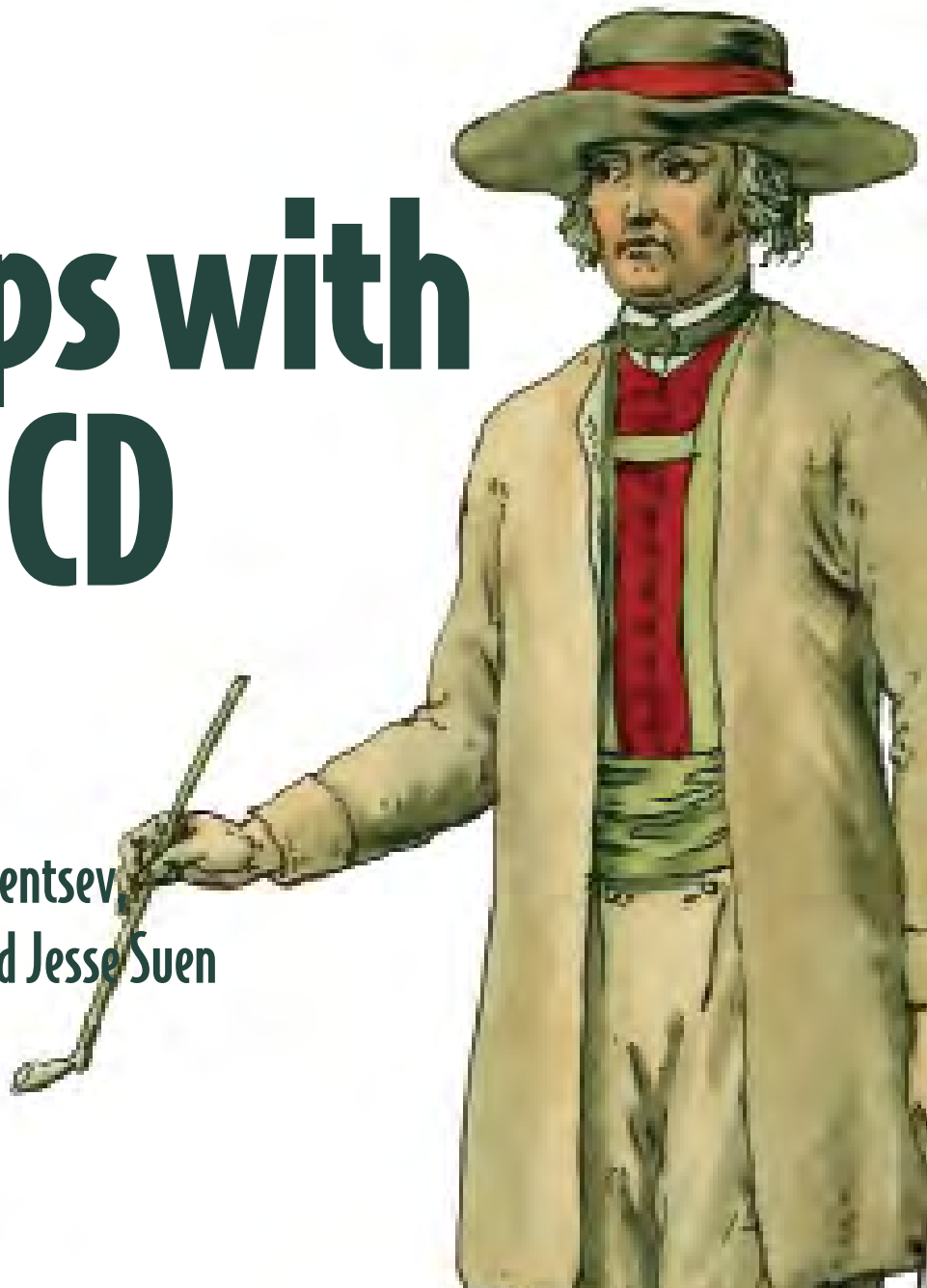


# GitOps with Argo CD

by Billy Yuen,  
Alexander Matyushentsev,  
Todd Ekenstam, and Jesse Suen





## *GitOps with Argo CD*

Billy Yuen, Alexander Matyushentsev,  
Todd Ekenstam, and Jesse Suen


Copyright 2021 Manning Publications

©2021 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

© Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

 Manning Publications Co.  
20 Baldwin Road Technical  
PO Box 761  
Shelter Island, NY 11964

Cover designer: Cica Tudor

ISBN: 9781633439733

# *contents*

---

*foreword iv*

**Chapter 1 Kubernetes and GitOps 1**

**Chapter 2 Argo CD 34**

**Appendix A Set up a test Kubernetes cluster 60**

**Appendix B Set up GitOps tools 63**

## *foreword*

---

Software teams that adopt GitOps deploy more often, have fewer regressions and recover from failures more quickly. The debate around the benefits of GitOps is settled by those who have adopted it. We've seen these successes time and again with Codefresh users, whether they're deploying at cloud scale, in small shops, or in edge clusters located around the world. GitOps works.

This special edition guide to GitOps, is written by Alexander Matyushentsev, one of the founders behind Argo. Argo is the fastest growing and most popular open-source GitOps tool in the world today. In a few short years, Argo has surged past big tech darlings and legacy open source projects for one simple reason. User's love it. It's the same reason Codefresh has joined the project and has based its enterprise platform on this beloved open-source tool: GitOps works. This book will show you how.

—Dan Garfield  
Co-Founder and Chief Open  
Source Officer, Codefresh

# NEW GitOps with argo Certification

Becoming GitOps with Argo Certified will teach you how GitOps works and how to implement it using Argo CD and Argo Rollouts, the world's most popular CNCF open-source GitOps tools. In this certification course, you will learn how to minimize failed deployments and safely accelerate features to your customers. GitOps with Argo Certification benefits include:



## Credibility

Increase your credibility as a GitOps expert.



## Confidence

Gain the confidence you need to build and deploy world-class software, more quickly than before.



## Career Advancement

Become a valuable resource to your organization + make a name for yourself as a GitOps expert in the market.



Visit <https://codefresh.io/courses/get-gitops-certified/> for more information

# *Kubernetes and GitOps*

---



## ***This chapter covers***

- Solving problems with Kubernetes
- Running and managing Kubernetes locally
- Understanding the basics of GitOps
- Implementing a simple Kubernetes GitOps operator

In this chapter, you'll get a bird's-eye view of Kubernetes and explore how this useful architecture helps you deploy, scale, and manage your containers. Then you'll zoom in on the differences between imperative and declarative syntax, why they matter to you, and why Kubernetes and GitOps work so well together.

## **1.1 *Kubernetes introduction***

Before diving into why Kubernetes and GitOps work so well together, let's talk about Kubernetes itself. This section provides a high-level overview of Kubernetes, how it compares to other container orchestration systems, and its architecture. We will also have an exercise that demonstrates how to run Kubernetes locally, which will be used for the other exercises in this book. This section is only a brief introduction and

refresher on Kubernetes. For a fun but informative overview of Kubernetes, check out “The Illustrated Children’s Guide to Kubernetes” and “Phippy Goes to the Zoo” by the Cloud Native Computing Foundation.<sup>1</sup> If you are completely new to Kubernetes, we recommend reading *Kubernetes in Action, Second Edition*, by Marko Lukša (Manning, 2020) and then returning to this book. If you are already familiar with Kubernetes and running minikube, you may skip to the exercise at the end of section 1.1.

### 1.1.1 What is Kubernetes?

Kubernetes is an open source container orchestration system released in 2014. OK, but what are containers, and why do you need to orchestrate them?

Containers provide a standard way to package your application’s code, configuration, and dependencies into a single resource. This enables developers to ensure that the application will run properly on any other machine regardless of any customized settings that machine may have that could differ from the machine used for writing and testing the code. Docker simplified and popularized containerization, which is now recognized as a fundamental technology used to build distributed systems.

**CHROOT** An operation available in UNIX operating systems, which changes the apparent root directory for the current running process and its children. Chroot provides a way to isolate a process and its children from the rest of the system. It was a precursor to containerization and Docker.<sup>2</sup>

While Docker solved the packaging and isolation problem of individual applications, there were still many questions about how to orchestrate the operation of multiple applications into a working distributed system:

- How do containers communicate?
- How is traffic routed between containers?
- How are containers scaled up to handle additional application load?
- How is the underlying infrastructure of the cluster scaled up to run the required containers?

All these operations are the responsibility of a container orchestration system and are provided by Kubernetes. Kubernetes helps to automate the deployment, scaling, and management of applications using containers.

**NOTE** Borg is Google’s internal container cluster management system used to power online services like Google search, Gmail, and YouTube. Kubernetes leverages Borg’s innovations and lessons learned, explaining why it is more stable and moves so much more quickly than its competitors.<sup>3</sup>

---

<sup>1</sup> <https://www.cncf.io/phippy>.

<sup>2</sup> <https://en.wikipedia.org/wiki/Chroot>.

<sup>3</sup> <https://kubernetes.io/blog/2015/04/borg-predecessor-to-kubernetes>.



Kubernetes was initially developed and open-sourced by Google based on a decade of experience with container orchestration using Borg, Google's proprietary cluster management system. Because of this, Kubernetes is relatively stable and mature for a system so complex. Because of its open API and extendable architecture, Kubernetes has developed an extensive community around it, which has further fueled its success. It is one of the top GitHub projects (as measured by stars), provides excellent documentation, and has a significant Slack and Stack Overflow community. An endless number of blogs and presentations from community members share their knowledge of using Kubernetes. Despite being started by Google, Kubernetes is not influenced by a single vendor. This makes the community open, collaborative, and innovative.

### 1.1.2 Other container orchestrators

Since late 2016, Kubernetes has become recognized as the dominant de facto industry-standard container orchestration system in much the same way that Docker has become the standard for containers. However, several Kubernetes alternatives address the same container orchestration problem as Kubernetes. Docker Swarm is Docker's native container orchestration engine that was released in 2015. It is tightly integrated with the Docker API and uses a YAML-based deployment model called Docker Compose. Apache Mesos was officially released in 2016 (although it has a history well before then) and supports large clusters, scaling to thousands of nodes.

While it may be possible to apply a GitOps approach to deploying applications using other container orchestration systems, this book focuses on Kubernetes.

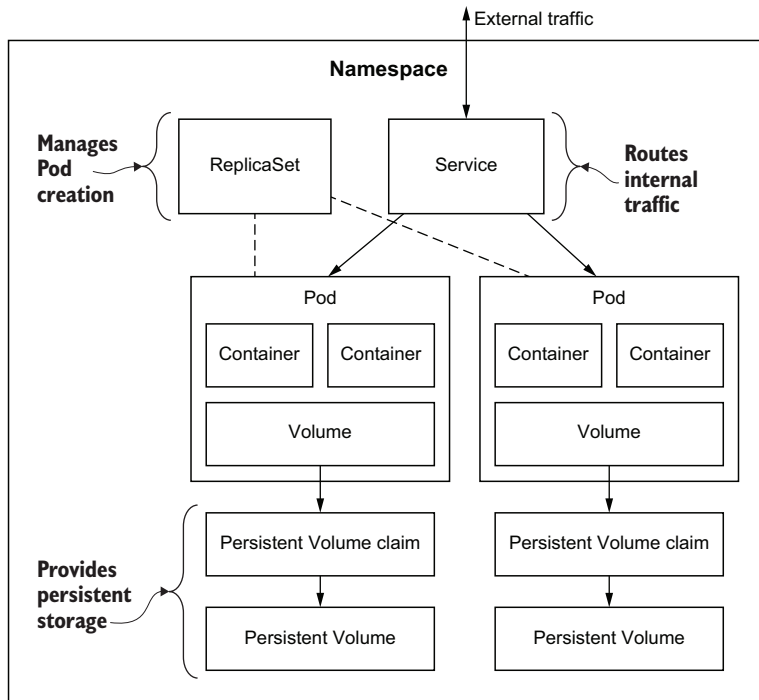
### 1.1.3 Kubernetes architecture

By the end of this chapter, you will complete an exercise that implements a basic GitOps continuous deployment operator for Kubernetes. But to understand how a GitOps operator functions, it is essential that you first understand a few Kubernetes core concepts and learn how it is organized at a high level.

Kubernetes is an extensive and robust system with many different types of resources and operations that can be performed on those resources. Kubernetes provides a layer of abstraction over the infrastructure and introduces the following set of basic objects that represent the desired cluster state:

- *Pod*—A group of containers deployed together on the same host. The Pod is the smallest deployable unit on a node and provides a way to mount storage, set environment variables, and provide other container configuration information. When all the containers of a Pod exit, the Pod dies also.
- *Service*—An abstraction that defines a logical set of Pods and a policy to access them.
- *Volume*—A directory accessible to containers running in a Pod.

Kubernetes architecture uses primary resources as a foundational layer for a set of higher-level resources. The higher-level resources implement features needed for real



**Figure 1.1** This diagram illustrates a typical Kubernetes environment deployed in a Namespace. A ReplicaSet is an example of a higher-level resource that manages the life cycle of Pods, which are lower-level, primary resources.

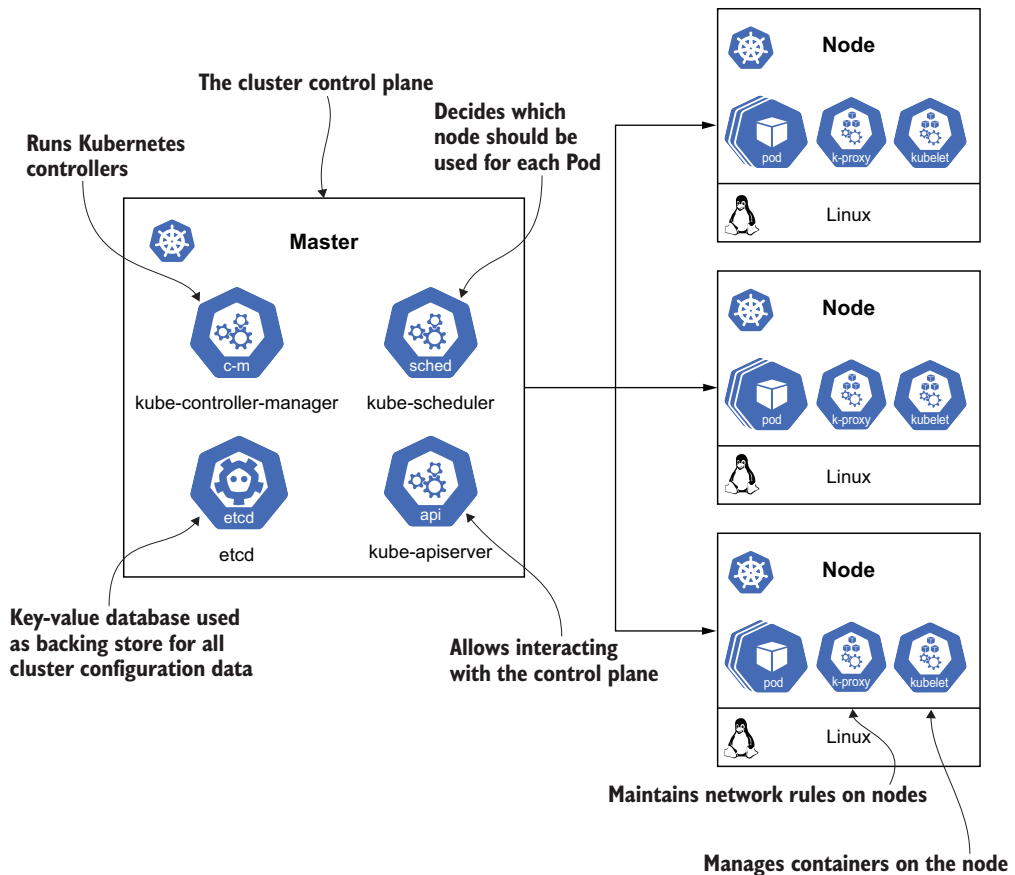
production use cases that leverage/extend the primary resources' functionality. In figure 1.1, you see that the ReplicaSet resource controls the creation of one or more Pod resources. Some other examples of high-level resources include

- *ReplicaSet*—Defines that a desired number of identically configured Pods are running. If a Pod in the ReplicaSet terminates, a new Pod will be started to bring the number of running Pods back to the desired number.
- *Deployment*—Enables declarative updates for Pods and ReplicaSets.
- *Job*—Creates one or more Pods that run to completion.
- *CronJob*—Creates Jobs on a time-based schedule.

Another important Kubernetes resource is the Namespace. Most kinds of Kubernetes resources belong to one (and only one) Namespace. A Namespace defines a naming scope where resources within a particular Namespace must be uniquely named. Namespaces also provide a way to isolate users and applications from each other through role-based access controls (RBACs), network policies, and resource quotas. These controls allow creating a multitenant Kubernetes cluster where multiple users share the same cluster and avoid impacting each other (for example, the “noisy neighbor” problem).

Kubernetes objects are stored in a control plane,<sup>4</sup> which monitors the cluster state, makes changes, schedules work, and responds to events. To perform these duties, each Kubernetes control plane runs the following three processes:

- kube-apiserver—An entry point to the cluster providing a REST API to evaluate and update the desired cluster state



**Figure 1.2** A Kubernetes cluster consists of several Services that run on the master nodes of the control plane and several other Services that run on the cluster's worker nodes. Together, these Services provide the essential Services that make up a Kubernetes cluster.

<sup>4</sup> <https://kubernetes.io/docs/concepts/overview/components/#control-plane-components>.

- `kube-controller-manager`—Daemon continuously monitoring the shared state of the cluster through the API server to make changes attempting to move the current state toward the desired state
- `kube-scheduler`—A component that is responsible for scheduling the workloads across the available nodes in the cluster
- `etcd`—A highly available key-value database typically used as Kubernetes' backing store for all cluster configuration data

The actual cluster workloads run using the compute resources of Kubernetes nodes. A node is a worker machine (either a VM or physical machine) that runs the necessary software to allow it to be managed by the cluster. Similar to the masters, each node runs a predefined set of processes:

- `kubelet`—The primary “node agent” that manages the actual containers on the node
- `kube-proxy`—A network proxy that reflects Services as defined in the Kubernetes API on each node and can do simple TCP, UDP, and SCTP stream forwarding

### 1.1.4 **Deploying to Kubernetes**

In this exercise, you will deploy a website using NGINX on Kubernetes. You will review some basic Kubernetes operations and become familiar with `minikube`, the single-node Kubernetes environment you will use for most exercises in this book.

**KUBERNETES TEST ENVIRONMENT: MINIKUBE** Refer to appendix A to set up a Kubernetes test environment using `minikube` to complete this exercise.

#### **CREATING A POD**

As was mentioned earlier in the chapter, a Pod is the smallest object in Kubernetes and represents a particular application workload. A Pod represents a group of related containers running on the same host and having the same operating requirements. All containers of a single Pod share the same network address, port space, and (optionally) file system using Kubernetes Volumes.

**NGINX** NGINX is an open source software web server used by many organizations and enterprises to host their websites because of its performance and stability.

In this exercise, you will create a Pod that hosts a website using NGINX. In Kubernetes, objects can be defined by a YAML text file “manifest” that provides all the information needed for Kubernetes to create and manage the object. Here is the listing for our NGINX Pod manifest.

**Listing 1.1 NGINX Pod manifest (<http://mng.bz/e5JJ>)**

```

kind: Pod
apiVersion: v1
metadata:
  name: nginx
spec:
  restartPolicy: Always
  volumes:
    - name: data
      emptyDir: {}
  initContainers:
    - name: nginx-init
      image: docker/whalesay
      command: [sh, -c]
      args: [echo "<pre>$(cowsay -b 'Hello Kubernetes')</pre>" >
        /data/index.html]
      volumeMounts:
        - name: data
          mountPath: /data
  containers:
    - name: nginx
      image: nginx:1.11
      volumeMounts:
        - name: data
          mountPath: /usr/share/nginx/html

```

The field `kind` and `apiVersion` are present in every Kubernetes resource and determine what type of object should be created and how it should be handled.

In this example, metadata has a name field that helps identify each Kubernetes resource. Metadata may also contain UID, labels, and other fields that will be covered later.

The spec section contains configuration that is specific to a particular kind of object. In the Pod example, spec includes a list of containers, the Volume shared between containers, and the Pod's restartPolicy.

The Volume that is used to share data between containers

The init section contains HTML generated using the `cowsay`<sup>5</sup> command.

The main container that serves the generated HTML file using the NGINX server

You are welcome to type in this listing and save it with a filename of `nginx-Pod.yaml`. However, since this book's object isn't to improve your typing skills, we recommend cloning our public Git repository that contains all the listings in this book and using those files directly:

<https://github.com/gitopsbook/resources>

Let's go ahead and start a minikube cluster and create the NGINX Pod using the following commands:

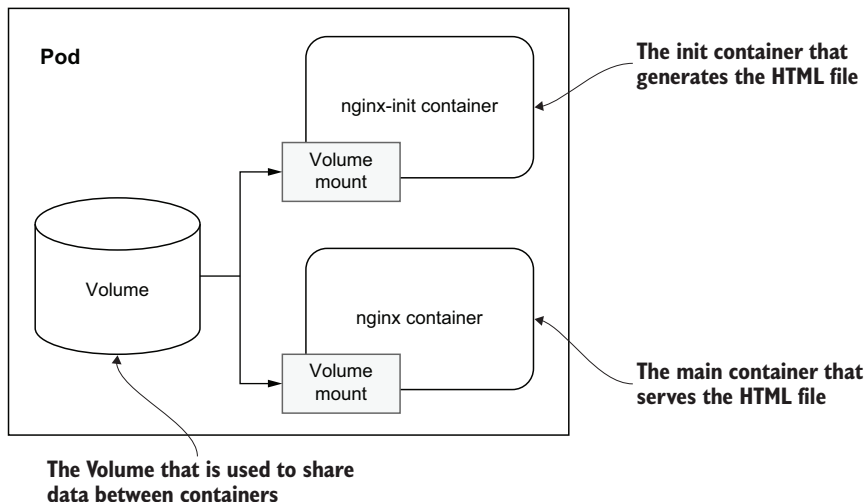
```

$ minikube start
(minikube/default)
🐳 minikube v1.1.1 on darwin (amd64)
👉 Creating virtualbox VM (CPUs=2, Memory=2048MB, Disk=20000MB) ...
👉 Configuring environment for Kubernetes v1.14.3 on Docker 18.09.6
👉 Pulling images ...
👉 Launching Kubernetes ...
👉 Verifying: apiserver proxy etcd scheduler controller dns
👉 Done! kubectl is now configured to use "minikube"
$ kubectl create -f nginx-Pod.yaml
Pod/nginx created

```

<sup>5</sup> <https://en.wikipedia.org/wiki/Cowsay>.

Figure 1.3 shows what the Pod looks like running inside the minikube.



**Figure 1.3** The nginx-init container writes the desired index.html file to the mounted Volume. The main NGINX container also mounts the Volume and displays the generated index.html when receiving HTTP requests.

### GETTING POD STATUS

As soon as the Pod is created, Kubernetes inspects the `spec` field and attempts to run the configured set of containers on an appropriate node in the cluster. The information about progress is available in the Pod manifest in the `status` field. The `kubectl` utility provides several commands to access it. Let's try to get the Pod status using the `kubectl get Pods` command:

```
$ kubectl get Pods
NAME    READY   STATUS    RESTARTS   AGE
nginx   1/1     Running   0           36s
```

The `get Pods` command provides a list of all the Pods running in a particular Namespace. In this case, we didn't specify a Namespace, so it gives the list of Pods running in the default Namespace. Assuming all goes well, the NGINX Pod should be in the Running state.

To learn even more about a Pod's status or debug why the Pod is not in the Running state, the `kubectl describe Pod` command outputs detailed information, including related Kubernetes events:

```
$ kubectl describe Pod nginx
Name:          nginx
Namespace:    default
Priority:      0
Node:         minikube/192.168.99.101
```

```

Start Time:   Sat, 26 Oct 2019 21:58:43 -0700
Labels:      <none>
Annotations:  kubect1.kubernetes.io/last-applied-configuration:

{"apiVersion": "v1", "kind": "Pod", "metadata": {"annotations": {}, "name": "nginx",
  "Namespace": "default"}, "spec": {"containers": [{"image": "nginx:1.11...
Status:      Running
IP:          172.17.0.4
Init Containers:
  nginx-init:
    Container ID:
      docker://128c98e40bd6b840313f05435c7590df0eacfc6ce989ec15cb7b484dc60d9bca
    Image:      docker/whalesay
    Image ID:   docker-
      pullable://docker/whalesay@sha256:178598e51a26abbc958b8a2e48825c90bc22e641
      de3d31e18aaf55f3258ba93b
    Port:      <none>
    Host Port: <none>
    Command:
      sh
      -c
    Args:
      echo "<pre>$(cowsay -b 'Hello Kubernetes')</pre>" > /data/index.html
    State:     Terminated
      Reason:   Completed
      Exit Code: 0
      Started:  Sat, 26 Oct 2019 21:58:45 -0700
      Finished: Sat, 26 Oct 2019 21:58:45 -0700
    Ready:     True
    Restart Count: 0
    Environment: <none>
    Mounts:
      /data from data (rw)
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-vbhsd
      (ro)
Containers:
  nginx:
    Container ID:
      docker://071dd946709580003b728cef12a5d185660d929ebfeb84816dd060167853e245
    Image:      nginx:1.11
    Image ID:   docker-
      pullable://nginx@sha256:e6693c20186f837fc393390135d8a598a96a833917917789d6
      3766cab6c59582
    Port:      <none>
    Host Port: <none>
    State:     Running
      Started:  Sat, 26 Oct 2019 21:58:46 -0700
    Ready:     True
    Restart Count: 0
    Environment: <none>
    Mounts:
      /usr/share/nginx/html from data (rw)
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-vbhsd (ro)
Conditions:
  Type          Status
  Initialized   True

```

```

Ready                True
ContainersReady     True
PodScheduled         True
Volumes:
  data:
    Type:            EmptyDir (a temporary directory that shares a Pod's lifetime)
    Medium:
    SizeLimit:       <unset>
  default-token-vbhsd:
    Type:            Secret (a volume populated by a Secret)
    SecretName:      default-token-vbhsd
    Optional:        false
QoS Class:           BestEffort
Node-Selectors:      <none>
Tolerations:         node.kubernetes.io/not-ready:NoExecute for 300s
                    node.kubernetes.io/unreachable:NoExecute for 300s

Events:
  Type    Reason      Age   From              Message
  ----    -
Normal   Scheduled   37m   default-scheduler Successfully assigned
        default/nginx to minikube
Normal   Pulling     37m   kubelet, minikube Pulling image "docker/whalesay"
Normal   Pulled      37m   kubelet, minikube Successfully pulled image
        "docker/whalesay"
Normal   Created     37m   kubelet, minikube Created container nginx-init
Normal   Started     37m   kubelet, minikube Started container nginx-init
Normal   Pulled      37m   kubelet, minikube Container image "nginx:1.11"
        already present on machine
Normal   Created     37m   kubelet, minikube Created container nginx
Normal   Started     37m   kubelet, minikube Started container nginx

```

Typically, the events section will contain clues as to why a Pod is not in the Running state.

The most exhaustive information is available via `kubectl get Pod nginx -o=yaml`, which outputs the full internal representation of the object in YAML format. The raw YAML output is difficult to read, and it is typically meant for programmatic access by resource controllers. Kubernetes resource controllers will be covered in more detail later in this chapter.

### ACCESSING THE POD

A Pod in the Running state means that all containers successfully started and the NGINX Pod is ready to serve requests. If the NGINX Pod in our cluster is running, we can try accessing it and prove that it is working.

Pods are not accessible from outside the cluster by default. There are multiple ways to configure external access, which include Kubernetes Services, Ingress, and more. For the sake of simplicity, we are going to use the command `kubectl port-forward` that forwards connections from a local port to a port on a Pod:

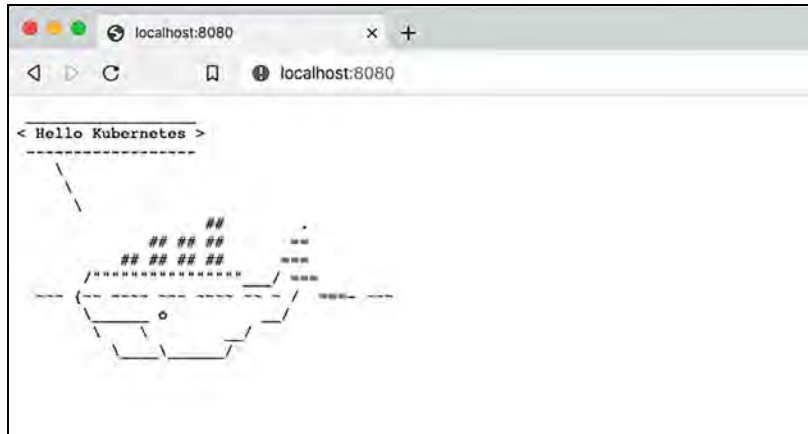
```

$ kubectl port-forward nginx 8080:80
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80

```



Keep the `kubectl port-forward` command running, and try opening `http://localhost:8080/` in your browser. You should see the generated HTML file!



**Figure 1.4** The generated HTML file content from the `docker/whalesay` image is an ASCII rendering of a cute whale with a speech bubble of greeting passed as a command argument. The `port-forward` command allows port 80 of the Pod (HTML) to be accessed on port 8080 of the local host.

### Exercise 1.1

Now that your NGINX Pod is running, use the `kubectl exec` command to get a shell on the running container.

**HINT** The command would be something like `kubectl exec -it <POD_NAME> -- /bin/bash`. Poke around in the shell. Run `ls`, `df`, and `ps -ef` as well as other Linux commands. What happens if you terminate the NGINX process?

As the final step in this exercise, let's delete the Pod to free up cluster resources. The Pod can be deleted using the following command:

```
$ kubectl delete Pod nginx
Pod "nginx" deleted
```

## 1.2 Declarative vs. imperative object management

The Kubernetes `kubectl` command-line tool is used to create, update, and manage Kubernetes objects and supports imperative commands, imperative object configuration, and declarative object configuration.<sup>6</sup> Let's go through a real-world example that demonstrates the difference between an imperative/procedural configuration and a

<sup>6</sup> <http://mng.bz/pVdP>.

declarative configuration in Kubernetes. First, let's look at how `kubectl` can be used imperatively.

In the following example, let's create a script that will deploy an NGINX service with three replicas and some annotations on the deployment.

### Listing 1.2 Imperative `kubectl` commands (`imperative-deployment.sh`)

<p><b>Creates a new deployment object called <code>nginx-imperative</code></b></p>	<p><b>Scales the <code>nginx-imperative</code> deployment to have three replicas of the Pod</b></p>
<p>→</p>	<p>←</p>
<pre>#!/bin/sh kubectl create deployment nginx-imperative --image=nginx:latest kubectl scale deployment/nginx-imperative --replicas 3 kubectl annotate deployment/nginx-imperative environment=prod kubectl annotate deployment/nginx-imperative organization=sales</pre>	
<p><b>Adds an annotation with the key <code>environment</code> and value <code>prod</code> to the <code>nginx-imperative</code> deployment</b></p>	<p><b>Adds an annotation with the key <code>organization</code> and value <code>sales</code> to the <code>nginx-imperative</code> deployment</b></p>
<p>→</p>	<p>←</p>

Try running the script against your `minikube` cluster, and check that the deployment was created successfully:

```
$ imperative-deployment.sh
deployment.apps/nginx-imperative created
deployment.apps/nginx-imperative scaled
deployment.apps/nginx-imperative annotated
deployment.apps/nginx-imperative annotated
$ kubectl get deployments
NAME                READY   UP-TO-DATE   AVAILABLE   AGE
nginx-imperative    3/3     3             3           27s
```

Great! The deployment was created as expected. But now let's edit our `deployment.sh` script to change the value of the `organization` annotation from `sales` to `marketing` and then rerun the script:

```
$ imperative-deployment-new.sh
Error from server (AlreadyExists): deployments.apps "nginx-imperative"
  already exists
deployment.apps/nginx-imperative scaled
error: --overwrite is false but found the following declared annotation(s):
  'environment' already has a value (prod)
error: --overwrite is false but found the following declared annotation(s):
  'organization' already has a value (sales)
```

As you can see, the new script failed because the deployment and annotations already exist. To make it work, we would need to enhance our script with additional commands and logic to handle the update case in addition to the creation case. Sure, this can be done, but it turns out we don't have to do all that work because `kubectl` can

itself examine the current state of the system and do the right thing using declarative object configuration.

The following manifest defines a deployment identical to the one created by our script (except that the deployment's name is `nginx-declarative`).

**Listing 1.3 Declarative (<http://mng.bz/OEP>)**

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-declarative
  annotations:
    environment: prod
    organization: sales
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
```

We can use the semi-magical `kubectl apply` command to create the `nginx-declarative` deployment:

```
$ kubectl apply -f declarative-deployment.yaml
deployment.apps/nginx-declarative created
$ kubectl get deployments
NAME                READY   UP-TO-DATE   AVAILABLE   AGE
nginx-declarative   3/3     3             3           5m29s
nginx-imperative    3/3     3             3           24m
```

After running the `apply`, we see the `nginx-declarative` deployment resource created. But what happens when we run `kubectl apply` again?

```
$ kubectl apply -f declarative-deployment.yaml
deployment.apps/nginx-declarative unchanged
```

Notice the change in the output message. The second time `kubectl apply` was run, the program detected that no changes needed to be made and subsequently reported that the deployment was unchanged. This is a subtle but critical difference between a `kubectl create` versus a `kubectl apply`. A `kubectl create` will fail if the resource already exists. The `kubectl apply` command first detects whether the resource exists and performs a create operation if the object doesn't exist or an update if it already exists.

As with the imperative example, what if we want to change the value of the organization annotation from sales to marketing? Let's edit the declarative-deployment.yaml file and change the metadata.annotations.organization field from sales to marketing. But before we run `kubectl apply` again, let's run `kubectl diff`:

```
$ kubectl diff -f declarative-deployment.yaml
:
-   organization: sales
+   organization: marketing
  creationTimestamp: "2019-10-15T00:57:44Z"
-   generation: 1
+   generation: 2
  name: nginx-declarative
  Namespace: default
  resourceVersion: "347771"
```

← The value of the organization label was changed from sales to marketing.

← The generation of this resource was changed by the system when doing `kubectl apply`.

```
$ kubectl apply -f declarative-deployment.yaml
deployment.apps/nginx-declarative configured
```

As you can see, `kubectl diff` correctly identified that the organization was changed from sales to marketing. We also see that `kubectl apply` successfully applied the new changes.

In this exercise, both the imperative and declarative examples result in a deployment resource configured in precisely the same way. And at first glance, the imperative approach may appear to be much simpler. It contains only a few code lines compared to the declarative deployment spec's verbosity that is five times the script's size. However, it contains problems that make it a poor choice to use in practice:

- The code is not idempotent and may have different results if executed more than once. If run a second time, an error will be thrown complaining that the deployment NGINX already exists. In contrast, the deployment spec is idempotent, meaning it can be applied as many times as needed, handling the case where the deployment already exists.
- It is more difficult to manage changes to the resource over time, especially when the difference is subtractive. Suppose you no longer wanted organization to be annotated on the deployment. Simply removing the `kubectl annotate` command from the scripted code would not help since it would do nothing to remove the existing deployment's annotation. A separate operation would be needed to remove it. On the other hand, with the declarative approach, you only need to remove the annotation line from the spec, and Kubernetes would take care of removing the annotation to reflect your desired state.
- It is more difficult to *understand* changes. If a team member sent a pull request modifying the script to do something differently, it would be like any other source code review. The reviewer would need to mentally walk through the script's logic to verify the algorithm achieves the desired outcome. There can even be bugs in the script. On the other hand, a pull request that changes a declarative deployment specification clearly shows the change to the system's

desired state. It is simpler to review, as there is no logic to check, only a configuration change.

- The code is not atomic, meaning that if one of the four commands in the script failed, the system's state would be partially changed and wouldn't be in the original state, nor would it be in the desired state. With the declarative approach, the entire spec is received as a single request, and the system attempts to fulfill all aspects of the desired state as a whole.

As you can imagine, what started as a simple shell script would need to become more and more complicated to achieve idempotency. There are dozens of options available in the Kubernetes deployment spec. With the scripted approach, if/else checks would need to be littered throughout the script to understand the existing state and conditionally modify the deployment.

### 1.2.1 How declarative configuration works

As we saw in the previous exercise, declarative configuration management is powered by the `kubectl apply` command. In contrast with imperative `kubectl` commands, like `scale` and `annotate`, the `kubectl apply` command has one parameter, the path to the file containing the resource manifest:

```
kubectl apply -f ./resource.yaml
```

The command is responsible for figuring out which changes should be applied to the matching resource in the Kubernetes cluster and update the resource using the Kubernetes API. It is a critical feature that makes Kubernetes a perfect fit for GitOps. Let's learn more about the logic behind `kubectl apply` and understand what it can and cannot do. To understand which problems `kubectl apply` is solving, let's go through different scenarios using the Deployment resource we created earlier.

The simplest scenario is when the matching resource does not exist in the Kubernetes cluster. In this case, `kubectl` creates a new resource using the manifest stored in the specified file.

If the matching resource exists, why doesn't `kubectl` replace it? The answer is obvious if you look at the complete manifest resource using the `kubectl get` command. Following is a partial listing of the Deployment resource that was created in the example. Some parts of the manifest have been omitted for clarity (indicated with ellipses):

```
$ kubectl get deployment nginx-declarative -o=yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    deployment.kubernetes.io/revision: "1"
    environment: prod
    kubectl.kubernetes.io/last-applied-configuration: |
      { ... }
    organization: marketing
```

```

creationTimestamp: "2019-10-15T00:57:44Z"
generation: 2
name: nginx-declarative
Namespace: default
resourceVersion: "349411"
selfLink: /apis/apps/v1/Namespace/default/deployments/nginx-declarative
uid: d41cf3dc-a3e8-40dd-bc81-76afd4a032b1
spec:
  progressDeadlineSeconds: 600
  replicas: 3
  revisionHistoryLimit: 10
  selector:
    matchLabels:
      app: nginx-declarative
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
    ...
status:
  ...

```

As you may have noticed, a live resource manifest includes all the fields specified in the file plus dozens of new fields such as additional metadata, the `status` field, and other fields in the resource spec. All these additional fields are populated by the Deployment controller and contain important information about the resource's running state. The controller populates information about resource state in the `status` field and applies default values of all unspecified optional fields, such as `revisionHistoryLimit` and `strategy`. To preserve this information, `kubectl` apply merges the manifest from the specified file and the live resource manifest. As a result, the command updates only fields specified in the file, keeping everything else untouched. So if we decide to scale down the deployment and change the `replicas` field to 1, then `kubectl` changes only that field in the live resource and saves it back to Kubernetes using an update API.

In real life, we don't want to control all possible fields that influence resource behavior in a declarative way. It makes sense to leave some room for imperativeness and skip fields that should be changed dynamically. The `replicas` field of the Deployment resource is a perfect example. Instead of hardcoding the number of replicas you want to use, the Horizontal Pod Autoscaler can be used to dynamically scale up or scale down your application based on load.

**HORIZONTAL POD AUTOSCALER** The Horizontal Pod Autoscaler automatically scales the number of Pods in a replication controller, deployment, or replica set based on observed CPU utilization (or, with custom metrics support, on some other application-provided metric).

Let's go ahead and remove the `replicas` field from the Deployment manifest. After applying this change, the `replicas` field is reset to the default value of one replica. But wait! The `kubectl apply` command updates only those fields that are specified in the file and ignores the rest. How does it know that the `replicas` field was deleted? The additional information that allows `kubectl` to handle the delete use case is hidden in an annotation of the live resource. Every time the `kubectl apply` command updates a resource, it saves the input manifest in the `kubectl.kubernetes.io/last-applied-configuration` annotation. So when the command is executed the next time, it retrieves the most recently applied manifest from the annotation, representing the common ancestor of the new desired manifest and live resource manifest. This allows `kubectl` to execute a three-way diff/merge and properly handle the case where some fields are removed from the resource manifest.

**THREE-WAY MERGE** A *three-way merge* is a merge algorithm that automatically analyzes differences between two files while also considering the origin or the common ancestor of both files.

Finally, let's discuss the situations where `kubectl apply` might not work as expected and should be used carefully.

First off, you typically should not mix imperative commands, such as `kubectl edit` or `kubectl scale`, with declarative resource management. This will make the current state not match the `last-applied-configuration` annotation and will defeat the merge algorithm `kubectl` uses to determine deleted fields. The typical scenario is when you experiment with the resource using `kubectl edit` and want to roll back changes by applying the original manifests stored in files. Unfortunately, it might not work since changes made by the `kubectl edit` command are not stored anywhere. For example, if you temporarily add the `resource limits` field to the deployment, the `kubectl apply` won't remove it since the `limits` field is not mentioned in the `last-applied-configuration` annotation or the manifest from the file. The `kubectl replace` command similarly ignores the `last-applied-configuration` annotation and removes that annotation altogether after applying the changes. So if you make any changes imperatively, you should be ready to undo the changes using imperative commands before continuing with declarative configuration.

You should also be careful when you want to stop managing fields declaratively. A typical example of this problem is adding the Horizontal Pod Autoscaler to manage scaling the number of replicas for an existing deployment. Typically, before introducing the Horizontal Pod Autoscaler, the number of deployment replicas is managed declaratively. To pass control of the `replicas` field over to the Horizontal Pod Autoscaler, the `replicas` field must first be deleted from the file that contains the Deployment manifest. This is so the next `kubectl apply` does not override the `replicas` value set by the Horizontal Pod Autoscaler. However, don't forget that the `replicas` field might also be stored in the `last-applied-configuration` annotation. If that is the case, the missing `replicas` field in the manifest file will be treated as a field dele-

tion, so whenever `kubectl apply` is run, the `replicas` value set imperatively by the Horizontal Pod Autoscaler will be removed from the live Deployment. The Deployment will scale down to the default of one replica.

In this section, we covered the different mechanisms for managing Kubernetes objects: imperative and declarative. You also learned a little about the internals of `kubectl` and how it identifies changes to apply to live objects. But at this point, you may be wondering what all this has to do with GitOps. The answer is simple: everything! Understanding how `kubectl` and Kubernetes manages changes to live objects is critical for understanding how the GitOps tools discussed in later chapters identify if the Git repository holding the Kubernetes configuration is in sync with the live state and how it tracks and applies changes.

### **1.3 Controller architecture**

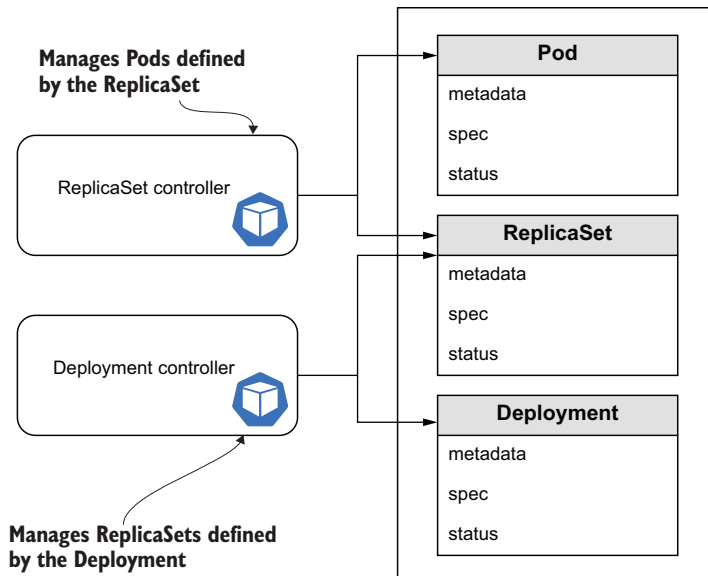
So far, we've learned about Kubernetes' declarative nature and the benefits it provides. Let's talk about what is behind each Kubernetes resource: the controller architecture. Understanding how controllers work will help us use Kubernetes more efficiently and understand how it can be extended.

Controllers are brains that understand what a particular kind of resource manifest means and execute the necessary work to make the system's actual state match the desired state as described by the manifest. Each controller is typically responsible for only one resource type. Through listening to the API server events related to the resource type being managed, the controller continuously watches for changes to the resource's configuration and performs the necessary work to move the current state toward the desired state. An essential feature of Kubernetes controllers is the ability to delegate work to other controllers. This layered architecture is powerful and allows you to reuse functionality provided by different resource types effectively. Let's consider a concrete example to understand the delegation concept better.

#### **1.3.1 Controller delegation**

The Deployment, ReplicaSet, and Pod resources perfectly demonstrate how delegation empowers Kubernetes. The Pod provides the ability to run one or more containers that have requested resources on a node in the cluster. This allows the Pod controller to focus simply on running an instance of an application and abstract the logic related to infrastructure provisioning, scaling up and down, networking, and other complicated details, leaving those to other controllers. Although the Pod resource provides many features, it is still not enough to run an application in production. We need to run multiple instances of the same application (for resiliency and performance), which means we need multiple Pods. The ReplicaSet controller solves this problem. Instead of directly managing multiple containers, it orchestrates multiple Pods and delegates the container orchestration to the Pod resource. Similarly, the





**Figure 1.5** Kubernetes allows for a resource hierarchy. Higher-level resources providing additional functionality, such as ReplicaSets and Deployments, can manage other higher-level resources or primary resources, such as Pods. This is implemented through a series of controllers, each managing events related to the resources it controls.

Deployment controller leverages functionality provided by ReplicaSets to implement various deployment strategies such as rolling updates.

**CONTROLLER DELEGATION BENEFIT** With controller delegation, Kubernetes functionality can be easily extended to support new capabilities. For example, services that are not backward-compatible can only be deployed with a blue/green strategy (not rolling updates). Controller delegation allows a new controller to be rewritten to support blue/green deployment and still leverage the Deployment controller functionality through delegation without reimplementing the Deployment controller’s core functionality.

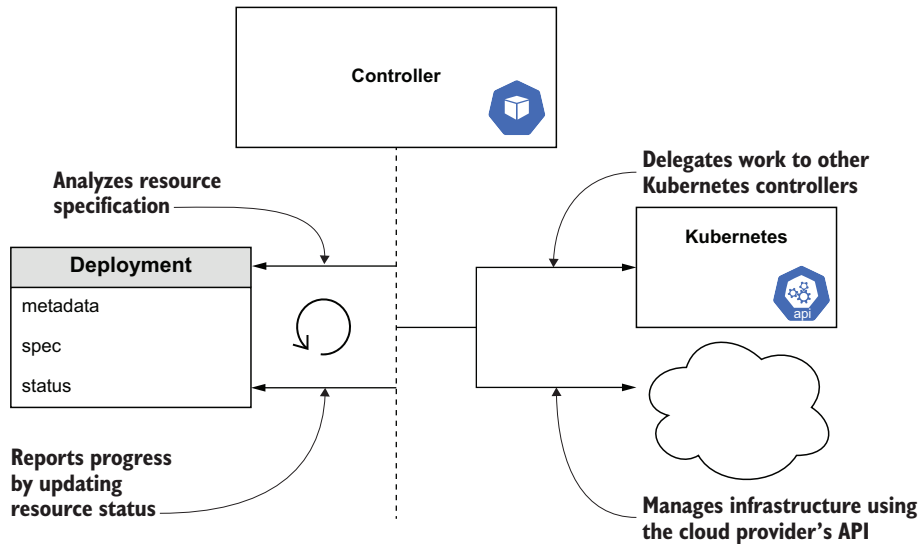
So as you can see from this example, controller delegation allows Kubernetes to build progressively more complex resources from simple ones.

### 1.3.2 Controller pattern

Although all controllers have different responsibilities, the implementation of each controller follows the same simple pattern. Each controller runs an infinite loop, and every iteration reconciles the desired and the actual state of the cluster resources it is responsible for. During reconciliation, the controller is looking for differences

between the actual and desired states and making the changes necessary to move the current state towards the desired state.

The desired state is represented by the `spec` field of the resource manifest. The question is, how does the controller know about the actual state? This information is available in the `status` field. After every successful reconciliation, the controller updates the `status` field. The `status` field provides information about cluster state to end users and enables the work of higher-level controllers. Figure 1.6 demonstrates the reconciliation loop.



**Figure 1.6** A controller operates in a continuous reconciliation loop where it attempts to converge the desired state as defined in the `spec` with the current state. Changes and updates to the resource are reported by updating the resource status. The controller may delegate work to other Kubernetes controllers or perform other operations, such as managing external resources using the cloud provider's API.

### CONTROLLERS VS. OPERATORS

Two terms that are often confused are *operator* and *controller*. In this book, the term *GitOps operator* is used to describe continuous delivery tools instead of *GitOps controller*. The reason for this is we are representing a specific type of controller that is application and domain-specific.

**KUBERNETES OPERATORS** A *Kubernetes operator* is an application-specific controller that extends the Kubernetes API to create, configure, and manage instances of complex stateful applications on behalf of a Kubernetes user. It builds upon the primary Kubernetes resource and controller concepts and includes domain or application-specific knowledge to automate everyday tasks.

The terms *operator* and *controller* are often confused since they are sometimes used interchangeably, and the line between the two is often blurred. However, another way to think about it is that the term *operator* is used to describe application-specific controllers. All operators use the controller pattern, but not all controllers are operators. Generally speaking, controllers tend to manage lower-level, reusable building-block resources, whereas operators operate at a higher level and are application-specific. Some examples of controllers are all of the built-in controllers that manage Kubernetes native types (Deployments, Jobs, Ingresses, and so on), as well as third-party controllers such as cert-manager (which provisions and manages TLS certificates) and the Argo Workflow Controller, which introduces a new job-like workflow resource in the cluster. An example of an operator is Prometheus, which manages Prometheus database installations.

### 1.3.3 NGINX operator

After learning about the controller fundamentals and the differences between controllers and operators, we are ready to implement an operator! The sample operator will solve a real-life task: managing a suite of NGINX servers with preconfigured static content. The operator will allow the user to specify a list of NGINX servers and configure static files mounted on each server. The task is not trivial and demonstrates the flexibility and power of Kubernetes.

#### DESIGN

As mentioned earlier in this chapter, Kubernetes' architecture allows you to leverage an existing controller's functionality through delegation. Our NGINX controller is going to leverage Deployment resources to delegate the NGINX deployment task.

The next question is which resource should be used to configure the list of servers and customized static content. The most appropriate existing resource is the ConfigMap. According to the official Kubernetes documentation, the ConfigMap is "an API object used to store non-confidential data in key-value pairs."<sup>7</sup> The ConfigMap can be consumed as environment variables, command-line arguments, or config files in a Volume. The controller will create a Deployment for each ConfigMap and mount the ConfigMap data into the default NGINX static website directory.

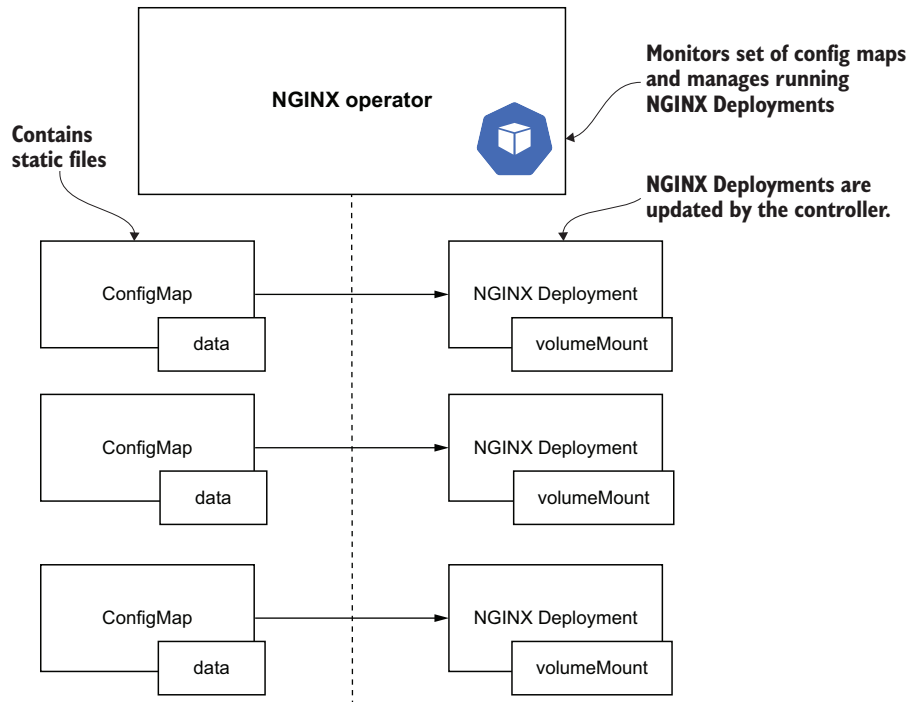
#### IMPLEMENTATION

Once we've decided on the design of the main building blocks, it is time to write some code. Most Kubernetes-related projects, including Kubernetes itself, are implemented using Go. However, Kubernetes controllers can be implemented using any language, including Java, C++, or even JavaScript. For the sake of simplicity, we are going to use a language that is most likely familiar to you: the Bash scripting language.

In section 1.3.2 we mentioned that each controller maintains an infinite loop and continuously reconciles the desired and actual state. In our example, the desired state is represented by the list of ConfigMaps. The most efficient way to loop through every

---

<sup>7</sup> <http://mng.bz/Yq67>.



**Figure 1.7** In the NGINX operator design, a ConfigMap is created containing the data to be served by NGINX. The NGINX operator creates a Deployment for each ConfigMap. Additional NGINX Deployments can be created simply by creating a ConfigMap with the web page data.

ConfigMap change is using the Kubernetes watch API. The watch feature is provided by the Kubernetes API for most resource types and allows the caller to be notified when a resource is created, modified, or deleted. The kubectl utility allows watching for resource changes using the get command with the `--watch` flag. The `--output-watch-events` command instructs kubectl to output the change type, which takes one of the following values: `ADDED`, `MODIFIED`, or `DELETED`.

**KUBECTL VERSION** Ensure that you are using the latest version of kubectl for this tutorial (version 1.16 or later). The `--output-watch-events` option was added relatively recently.

#### Listing 1.4 Sample ConfigMap (<http://mng.bz/GxRN>)

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: sample
data:
  index.html: hello world
```

In one window, run the following command:

```
$ kubectl get --watch --output-watch-events configmap
```

In another terminal window, run `kubectl apply -f sample.yaml` to create the sample ConfigMap. Notice the new output in the window running the `kubectl --watch` command. Now run `kubectl delete -f sample.yaml`. You should now see a DELETED event appear:

```
$ kubectl get --watch --output-watch-events configmap
EVENT      NAME      DATA  AGE
ADDED      sample    1      3m30s
DELETED    sample    1      3m40s
```

After running this experiment manually, you should be able to see how we can write our NGINX operator as a Bash script.

The `kubectl get --watch` command outputs a new line every time a ConfigMap resource is created, changed, or deleted. The script will consume the output of `kubectl get --watch` and either create a new Deployment or delete a Deployment depending on the output ConfigMap event type. Without further delay, the full operator implementation is shown in the following code listing.

#### Listing 1.5 NGINX controller (<http://mng.bz/zxmZ>)

```
#!/usr/bin/env bash
```

```

kubectl get --watch --output-watch-events configmap \
-o=custom-columns=type:type,name:object.metadata.name \
--no-headers | \
while read next; do
    NAME=$(echo $next | cut -d' ' -f2)
    EVENT=$(echo $next | cut -d' ' -f1)

    case $EVENT in
        ADDED|MODIFIED)
            kubectl apply -f - << EOF
apiVersion: apps/v1
kind: Deployment
metadata: { name: $NAME }
spec:
  selector:
    matchLabels: { app: $NAME }
  template:
    metadata:
      labels: { app: $NAME }
      annotations: { kubectl.kubernetes.io/restartedAt: $(date) }
    spec:
      containers:
      - image: nginx:1.7.9
        name: $NAME
        ports:
        - containerPort: 80

```

This kubectl command outputs all the events that occur for configmap objects.

The output from kubectl is processed by this infinite loop.

The name of the configmap and the event type are parsed from the kubectl output.

If the configmap has been ADDED or MODIFIED, apply the NGINX deployment manifest (everything between the two EOF tags) for that configmap.

```

        volumeMounts:
        - { name: data, mountPath: /usr/share/nginx/html }
    volumes:
    - name: data
      configMap:
        name: $NAME
EOF
        ;;
DELETED)
    kubectl delete deploy $NAME
        ;;
esac
done

```

← If the configmap has been **DELETED**, delete the NGINX deployment for that configmap.

### TESTING

Now that the implementation is done, we are ready to test our controller. In real life, the controller is packaged into a Docker image and runs inside the cluster. It is OK to run the controller outside of the cluster for testing purposes, which is precisely what we are going to do. Should we include Appendix A in the report., start a minikube cluster, save the controller code into a file called `controller.sh`, and start it using this Bash command:

```
$ bash controller.sh
```

**NOTE** This example requires `kubectl` version 1.16 or later.

The controller is running and waiting for the ConfigMap. Let's create one. Refer to listing 1.4 for the manifest of the ConfigMap.

We create the ConfigMap using the `kubectl apply` command:

```
$ kubectl apply -f sample.yaml
configmap/sample created
```

The controller notices the change and creates an instance of Deployment using the `kubectl apply` command:

```
$ bash controller.sh
deployment.apps/sample created
```

#### **Exercise 1.2**

Try accessing the NGINX controller by forwarding port 80 locally to make sure the controller works as expected. Try to delete or modify the ConfigMap and see how the controller reacts accordingly.

#### **Exercise 1.3**

Create additional ConfigMaps to launch an NGINX server for each member of your family that displays `Hello <name>!`. Also, don't forget to call `/text/Snaphchat` them IRL.

#### **Exercise 1.4**

Write a Dockerfile to package the NGINX controller. Deploy it to your test Kubernetes cluster. Hint: You will need to create RBAC resources for the operator.

## 1.4 Kubernetes + GitOps

GitOps assumes that every piece of infrastructure is represented as a file stored in a revision control system, and there is an automated process that seamlessly applies changes to the application runtime environment. Without a system like Kubernetes, this is, unfortunately, easier said than done. There are too many things to worry about and many different technologies that do not work well together. These two assumptions often become an unsolvable obstacle that prevents the implementation of an efficient infrastructure-as-code process.

Kubernetes has dramatically improved the situation. As Kubernetes gained more and more adoption, the idea of infrastructure as code (IaC) has evolved, which resulted in the creation of new tooling that implements GitOps. So what is so special about Kubernetes, and how and why did it lead to the rise of GitOps?

Kubernetes enables GitOps by fully embracing declarative APIs as its primary mode of operation and providing the controller patterns and backend framework necessary to implement those APIs. The system was designed with the principles of declarative specifications and eventual consistency and convergence from its inception.

**EVENTUAL CONSISTENCY** *Eventual consistency* is a consistency model used in distributed computing to achieve high availability that informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.

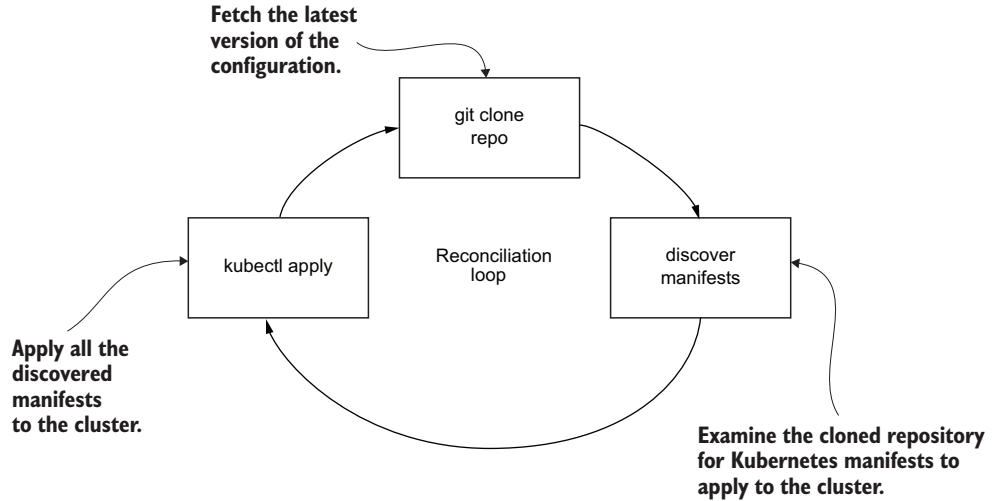
This decision is what led to the prominence of GitOps in Kubernetes. Unlike traditional systems, in Kubernetes there are almost no APIs that can modify only a subset of some existing resources. For example, there is no API (and never will be) that changes only the container image of a Pod. Instead, the Kubernetes API server expects all API requests to provide a complete manifest of the resource to the API server. It was an intentional decision not to give any convenience APIs to users. As a result, Kubernetes users are essentially forced into a declarative mode of operation, which leads these same users to the need to store these declarative specifications somewhere. Git became the natural medium to store these specifications, and GitOps then became the natural delivery tool to deploy these manifests from Git.

## 1.5 Getting started with CI/CD

Now that you've learned the basic architecture and principles of a Kubernetes controller and how Kubernetes is a good fit for GitOps, it's time to implement your own GitOps operator. In this tutorial, we will first be creating a rudimentary GitOps operator to drive continuous delivery. This is followed by an example of how you would integrate continuous integration (CI) with a GitOps-based continuous delivery (CD) solution.

### 1.5.1 Basic GitOps operator

To implement your own GitOps operator, a continuously running control loop needs to be implemented that performs the three steps illustrated in figure 1.8.



**Figure 1.8** The GitOps reconciliation loop begins by cloning the repository to fetch the configuration repository’s latest version into local storage. Next, the manifest discovery step walks the cloned repository’s filesystem, looking for any Kubernetes manifests to apply to the cluster. Last, the `kubectl apply` step performs the actual deployment by applying all of the discovered manifests to the cluster.

While this control loop could be implemented in any number of ways, most simply, it could be implemented as a Kubernetes CronJob.

#### Listing 1.6 CronJob GitOps operator (<http://mng.bz/0myz>)

```

apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: gitops-cron
  Namespace: gitops
spec:
  schedule: "*/5 * * * *"
  concurrencyPolicy: Forbid
  jobTemplate:
    spec:
      backoffLimit: 0
      template:
        spec:
          restartPolicy: Never
  
```

Executes the GitOps reconciliation loop every five minutes

Prevents concurrent executions of the job

Doesn't retry failed jobs since this is a recurring CronJob; retries happen naturally

Doesn't restart the container when it completes



A Kubernetes Service account that has sufficient privileges to create and modify objects into the cluster

```

    > serviceAccountName: gitops-serviceaccount
    containers:
      - name: gitops-operator
        image: gitopsbook/example-operator:v1.0
        command: [sh, -e, -c]
        args:
          - git clone https://github.com/gitopsbook/sample-app-
            deployment.git /tmp/example &&
            find /tmp/example -name '*.yaml' -exec kubectl apply -f {} \;
  
```

The Docker image that has the git, find, and kubectl binaries preloaded into it

The command and args fields contain the actual logic of the GitOps reconciliation loop.

The job template spec contains the meat of the operator logic. The CronJob `gitops-cron` contains the control loop logic that deploys manifests from Git to the cluster on a regularly scheduled basis. The `schedule` field is a cron expression, which in this example will result in the job being executed every five minutes. Setting the `concurrencyPolicy` to `Forbid` prevents concurrent executions of the job, allowing the current execution to complete before attempting to start a second. Note that this will only happen if a single execution takes longer than five minutes.

The `jobTemplate` is a Kubernetes Job template spec. The Job template spec contains a Pod template spec (`jobTemplate.spec.template.spec`), which is the same spec that you may be familiar with from writing Kubernetes manifests for Deployments, Pods, Jobs, and so on. The `backoffLimit` specifies the number of retries before considering a Job as failed. A value of zero means that it will not retry. Since this is a recurring CronJob, retries happen naturally, so there is no need to retry immediately. A `restartPolicy` of `Never` is required to prevent the Job from restarting the container when it completes, which is a container's normal behavior. The `serviceAccountName` field references a Kubernetes Service account with sufficient privileges to create and modify objects in the cluster. Since this operator could potentially deploy any type of resource, the `gitops-operator` Service account should be bound to an admin-level ClusterRole.

The `command` and `args` fields contain the actual logic of the GitOps reconciliation loop. It consists of only two commands:

- `git clone`—Clones the latest repository to local storage
- `find`—Discovers YAML files in the repo, and for each YAML file located, executes the `kubectl apply` command

To use this, simply apply the CronJob to the cluster. Note that you would first need to apply the following supporting resources.

#### Listing 1.7 CronJob GitOps resources (<http://mng.bz/KMln>)

```

apiVersion: v1
kind: Namespace
  
```

← Namespace `gitops` is where the CronJob and ServiceAccount will live.

```

metadata:
  name: gitops

---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: gitops-serviceaccount
  Namespace: gitops

---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: gitops-operator
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: admin
subjects:
- kind: ServiceAccount
  name: gitops-serviceaccount
  Namespace: gitops

```

← **ServiceAccount `gitops-serviceaccount` is the Kubernetes Service account that will have privileges to deploy to the cluster.**

← **ClusterRoleBinding `gitops-operator` binds/grants cluster admin-level privileges to the ServiceAccount, `gitops-serviceaccount`.**

**MULTIRESOURCE YAML FILES** Management of multiple resources can be simplified by grouping them in the same file (separated by `---` in YAML). Listing 1.7 is an example of a single YAML file defining multiple related resources.

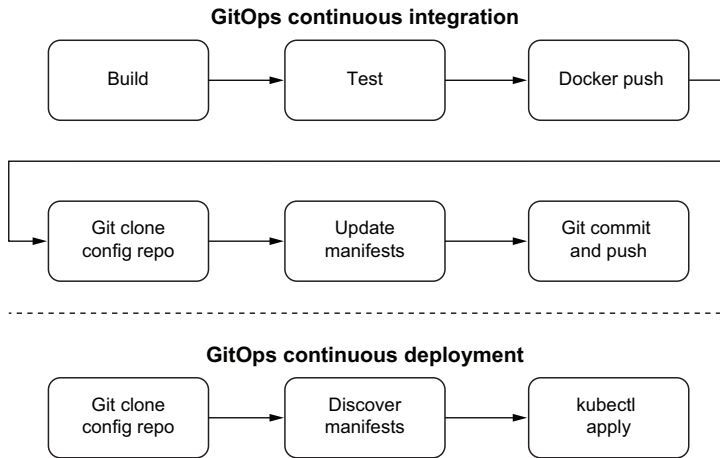
This example is primitive, meant to illustrate the fundamental concepts of a GitOps continuous delivery operator. It is not meant for any real production use since it lacks many features needed in a real-world production environment. For example, it cannot prune any resources that are no longer defined in Git. Another limitation is that it does not deal with any credentials required to connect to the Git repository.

#### **Exercise 1.5**

Modify the CronJob to point to your own GitHub repository. Apply the new CronJob, and add YAML files to your repository. Verify that the corresponding Kubernetes resources are created.

### **1.5.2 Continuous integration pipeline**

In the previous section, we implemented a basic GitOps CD mechanism that continuously delivers manifests in a Git repository to the cluster. The next step is to integrate this process with a CI pipeline, which publishes new container images and updates the Kubernetes manifests with the new image. GitOps integrates well with any CI system, as the process is more or less the same as a typical build pipeline. The main difference is that instead of the CI pipeline communicating directly to the Kubernetes API server, it commits the desired change into Git and trusts that sometime later, the new changes will be detected by the GitOps operator and applied.



**Figure 1.9** A GitOps CI pipeline is similar to a typical CI pipeline. The code is built and tested, and then the artifact (a tagged Docker image) is pushed to the image registry. The additional step is the GitOps CI pipeline also updates the manifests in the configuration repo with the latest image tag. This update may trigger a GitOps CD job to apply the updated manifests to the cluster.

The goal of a GitOps CI pipeline is to

- Build your application and run unit testing as necessary
- Publish a new container image to a container registry
- Update the Kubernetes manifests in Git to reflect the new image

The following example is a typical series of commands that would be executed in a CI pipeline to achieve this.

#### Listing 1.8 Example GitOps CI (<http://mng.bz/9M18>)

```

export VERSION=$(git rev-parse HEAD | cut -c1-7)
make build
make test

export NEW_IMAGE="gitopsbook/sample-app:${VERSION}"
docker build -t ${NEW_IMAGE} .
docker push ${NEW_IMAGE}

git clone http://github.com/gitopsbook/sample-app-deployment.git
cd sample-app-deployment

kubectl patch \

```

**Uses the first seven characters of the current commit-SHA as the version to uniquely identify the artifacts from this build**

**Builds and tests your application's binaries as you usually would**

**Builds the container image, pushes it to a container registry, and incorporates the unique version as part of the container image tag**

**Clones the Git deployment repo containing the Kubernetes manifests**

**Updates the manifests with the new image**

```

--local \
-o yaml \
-f deployment.yaml \
-p "spec:
  template:
    spec:
      containers:
        - name: sample-app
          image: ${NEW_IMAGE}" \
> /tmp/newdeployment.yaml
mv /tmp/newdeployment.yaml deployment.yaml

git commit deployment.yaml -m "Update sample-app image to ${NEW_IMAGE}"
git push

```

**Commits and pushes the manifest changes to the deployment configuration repo**

This example pipeline is one way that a GitOps CI pipeline may look. There are some important points to highlight regarding the different choices you might make that would better suit your needs.

### IMAGE TAGS AND THE TRAP OF THE LATEST TAG

Notice in the first two steps of the example pipeline, the current Git commit-SHA of the application's Git repository is used as a version variable, which is then incorporated as part of the container's image tag. A resulting container image in the example pipeline might look like `gitopsbook/sample-app:cc52a36`, where `cc52a36` is the commit-SHA at the time of the build.

It is important to use a unique version string (like a commit-SHA) that is different in each build since the version is incorporated as part of the container image tag. A common mistake that people make is to use `latest` as their image tag (such as `gitopsbook/sample-app:latest`) or reuse the same image tag from build to build. A naive pipeline might make the following mistake:

```

make build
docker build -t gitopsbook/sample-app:latest .
docker push gitopsbook/sample-app:latest

```

Reusing image tags from build to build is a terrible practice for several reasons.

The first reason why container tags should not be reused is that when container image tags are reused, Kubernetes will not deploy the new version to the cluster. This is because the second time the manifests are attempted to be applied, Kubernetes will not detect any change in the manifests, and the second `kubectl apply` will have zero effect. For example, say build #1 publishes the image `gitopsbook/sample-app:latest` and deploys it to the cluster. The Deployment manifest for this might look something like this.

#### Listing 1.9 Sample app deployment (<http://mng.bz/j4m9>)

```

apiVersion: apps/v1
kind: Deployment
metadata:

```

```

name: sample-app
spec:
  replicas: 1
  revisionHistoryLimit: 3
  selector:
    matchLabels:
      app: sample-app
  template:
    metadata:
      labels:
        app: sample-app
    spec:
      containers:
        - image: gitopsbook/sample-app:latest
          name: sample-app
          command:
            - /app/sample-app
          ports:
            - containerPort: 8080

```

When build #2 runs, even though a new container image for `gitopsbook/sample-app:latest` has been pushed to the container registry, the Kubernetes Deployment YAML for the application is the same as it was in build #1. The Deployment specs are the same from the perspective of Kubernetes; there is no difference between what is being applied in build #1 versus build #2. Kubernetes treats the second apply as a no-op (no operation) and does nothing. For Kubernetes to redeploy, something needs to be different in the Deployment spec from the first build to the second. Using unique container image tags ensures there is a difference.

Another reason for incorporating a unique version into the image tag is that it enables traceability. By incorporating something like the application's Git commit-SHA into the tag, there is never any question about what version of the software is currently running in the cluster. For example, you could run the following `kubectl` command, which outputs the images of all deployments in the Namespace:

```

$ kubectl get deploy -o wide | awk '{print $1,$7}' | column -t
NAME          IMAGES
sample-app    gitopsbook/sample-app:508d3df

```

By using the convention of tying container image tags to Git commit-SHAs of your application repository, you can trace the currently running version of the `sample-app` to commit `508d3df`. From there, you have full knowledge of exactly what version of your application is running in the cluster.

The third and possibly most important reason for not reusing image tags such as `latest` is that rollback to the older version becomes impossible. When you reuse image tags, you are overriding or rewriting the meaning of that overwritten image. Imagine the following sequence of events:

- 1 Build #1 publishes the container image `gitopsbook/sample-app:latest` and deploys it to the cluster.

- 2 Build #2 republishes the container image `gitopsbook/sample-app:latest`, overwriting the image tag deployed in build #1. It redeploys this image to the cluster.
- 3 Sometime after build #2 is deployed, it is discovered that a severe bug exists in the latest version of the code, and immediate rollback is necessary to the version created in build #1.

There is no easy way to redeploy the version of the `sample-app` created during build #1 because there is no image tag representing that version of the software. The second build overwrote the `latest` image tag, effectively making the original image unreachable (at least not without extreme measures).

For these reasons, it is not recommended to reuse image tags, such as `latest`, at least in production environments. With that said, in dev and test environments, continuously creating new and unique image tags (which likely never get cleaned up) could cause an excessive amount of disk usage in your container registry or become unmanageable just by the sheer number of image tags. In these scenarios, reusing image tags may be appropriate, understanding Kubernetes' behavior of not doing anything when the same specification is applied twice.

**KUBECTL ROLLOUT RESTART** Kubectl has a convenience command, `kubectl rollout restart`, which causes all the Pods of a deployment to restart (even if the image tag is the same). It is useful in dev and test scenarios where the image tag has been overwritten and redeploy is desired. It works by injecting an arbitrary timestamp into the Pod template metadata annotations. This causes the Pod spec to be different from what it was before, which causes a regular, rolling update of the Pods.

One thing to note is that our CI example uses a Git commit-SHA as the unique image tag. But instead of a Git commit-SHA, the image tag could incorporate any other unique identifier, such as a semantic version, a build number, a date/time string, or even a combination of these pieces of information.

**SEMANTIC VERSION** A *semantic version* is a versioning methodology that uses a three-digit convention (MAJOR.MINOR.PATCH) to convey the meaning of a version (such as `v2.0.1`). MAJOR is incremented when there are incompatible API changes. MINOR is incremented when functionality is added in a backward-compatible manner. PATCH is incremented when there are backward-compatible bug fixes.

## Summary

- Kubernetes is a container orchestration system for deployment, scaling, and management of containers.
- Basic Kubernetes objects are Pod, Service, and Volume.
- The Kubernetes control plane consists of `kube-apiserver`, `kube-controller-manager`, and `kube-scheduler`.

- Each Kubernetes worker node runs kubelet and kube-proxy.
- A Running Service in a Pod is accessible from your computer using `kubectl port-forward`.
- Pods can be deployed by using imperative or declarative syntax. Imperative deployment is not idempotent, and declarative deployment is idempotent. For GitOps, declarative is the preferred method.
- Controllers are the brains in Kubernetes to bring the Running state into the desired state.
- A Kubernetes operator can be implemented simply as a shell script by monitoring ConfigMap changes and updating deployment.
- Kubernetes configuration is declarative.
- GitOps complements Kubernetes due to its declarative nature.
- GitOps operators trigger deployments to your Kubernetes cluster based on changes to revision-controlled configuration files stored in Git.
- A simple GitOps operator can be implemented as a script by regularly checking the manifest Git repo for changes.
- CI pipeline can be implemented as a script with steps to build the Docker image and update the manifest with the new image tag.

# Argo CD

---

## ***This chapter covers***

- What is Argo CD?
- Deploying an application using Argo CD
- Using Argo CD enterprise features

In this chapter, you'll meet Argo CD, an open source GitOps operator designed with enterprises in mind. You'll get firsthand experience deploying an example application to Kubernetes using this important member of the Argo family of cloud-native tools, and explore the many uses and benefits it has to offer.

## **2.1 What is Argo CD?**

Argo CD is an open source GitOps operator for Kubernetes.<sup>8</sup> The project is a part of the Argo family, a set of cloud-native tools for running and managing jobs and applications on Kubernetes. Along with Argo Workflows, Rollouts, and Events, Argo CD focuses on application delivery use cases and makes it easier to combine three modes of computing: services, workflows, and event-based processing. In

---

<sup>8</sup> <https://argoproj.github.io/projects/argo-cd>.



2020, Argo CD was accepted by the Cloud Native Computing Foundation (CNCF) as an incubation-level hosted project.

**CNCF** The Cloud Native Computing Foundation is a Linux Foundation project that hosts critical components of the global technology infrastructure.

The company behind Argo CD is Intuit, the creator of TurboTax and QuickBooks. In early 2018, Intuit decided to adopt Kubernetes to speed up cloud migration. At the time, the market already had several successful continuous deployment tools for Kubernetes, but none of them fully satisfied Intuit's needs. So instead of adopting an existing solution, the company decided to invest in a new project and started working on Argo CD. What is so special about Intuit's requirements? The answer to that question explains how Argo CD is different from other Kubernetes CD tools and explains its main project use cases.

### 2.1.1 Main use cases

The importance of a GitOps methodology and benefits of representing infrastructure as code is not questionable. However, the enterprise scale demands additional requirements. Intuit is a cloud-based software-as-a-service company. With around 5,000 developers, the company successfully runs hundreds of microservices on-premises and in the cloud. Given that scale, it was unreasonable to expect that every team would run its own Kubernetes cluster. Instead, it was decided that a centralized platform team would run and maintain a set of multitenant clusters for the whole company. At the same time, end users should have the freedom and necessary tools to manage workloads in those clusters. These considerations have defined the following additional requirements on top of the decision to use GitOps.

#### **AVAILABLE AS A SERVICE**

A simple onboarding process is extremely important if you are trying to move hundreds of microservices to Kubernetes. Instead of asking every team to install, configure, and maintain the deployment operator, it should be provided by the centralized team. With several thousands of new users, SSO integration is crucial. The service must integrate with various SSO providers instead of introducing its own user management.

#### **ENABLE MULTITENANCY AND MULTICLUSTER MANAGEMENT**

In multitenant environments, users need an effective and flexible access control system. Kubernetes has a great built-in role-based access control system, but that is not enough when you have to deal with hundreds of clusters. The continuous deployment tool should provide access control on top of multiple clusters and seamlessly integrate with existing SSO providers.

#### **ENABLE OBSERVABILITY**

Last, but not least, the continuous deployment tool should provide developers insights about the state of managed applications. That assumes a user-friendly interface that quickly answers the following questions:

- Is the application configuration in sync with the configuration defined in Git?
- What exactly is not matching?
- Is the application up and running?
- What exactly is broken?

The company needed the GitOps operator ready for enterprise scale. The team evaluated several GitOps operators, but none of them satisfied all the requirements, so it was decided to implement Argo CD.

### Exercise 2.1

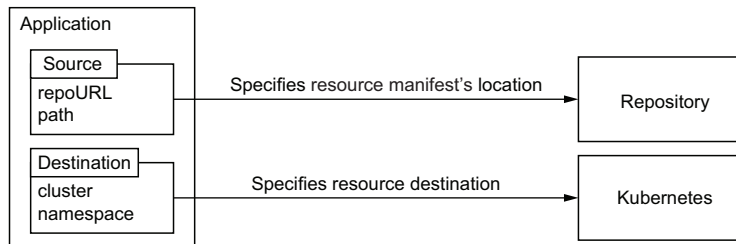
Reflect on your organization's needs and compare them to use cases that Argo CD is focused on. Try to decide if Argo CD solves the pain points your team has.

## 2.1.2 Core concepts

In order to effectively use Argo CD, we should understand two basic concepts: the Application and the Project. Let's have a closer look at the Application first.

### APPLICATION

The Application provides a logical grouping of Kubernetes resources and defines a resources manifest's source and destination.



**Figure 2.1** The main properties of the Argo CD Application are the source and destination. The source specifies a resource manifest's location in the Git repository. The destination specifies where resources should be created in the Kubernetes cluster.

The Application source includes the repository URL and the directory inside of the repository. Typically repositories include multiple directories, one per application environment such as QA and Prod. The sample directory structure of such a repository is represented here:

```

.
├── prod
│   └── deployment.yaml
└── qa
    └── deployment.yaml
  
```

Each directory does not necessarily contain plain YAML files. Argo CD does not enforce any configuration management tool and instead provides first-class support for various config management tools. So the directory might as well contain a Helm chart definition as YAML along with Kustomize overlays.

The Application destination defines where resources must be deployed and includes the API server URL of the target Kubernetes cluster, along with the cluster Namespace name. The API server URL identifies the cluster where all application manifests must be deployed. It is impossible to deploy application manifests across several clusters, but different applications might be deployed into different clusters. The Namespace name is used to identify the target Namespace of all Namespace-level application resources.

So the Argo CD Application represents an environment deployed in the Kubernetes cluster and connects it to the desired state stored in the Git repository.

### **Exercise 2.2**

Consider the real service deployed in your organization and come up with a list of Argo CD applications. Define the source repository URL, directory, and target cluster with the Namespace for one of the applications from your list.

### **2.1.3 Sync and health statuses**

In addition to the source and destination, the Argo CD application has two more important properties: sync and health statuses.

Sync status answers whether the observed application resources state deviates from the resources state stored in the Git repository. The logic behind deviation calculation is equivalent to the logic of the `kubectl diff` command. The possible values of a sync status are `in-sync` and `out-of-sync`. The `in-sync` status means that each application resource is found and fully matching to the expected resource state. The `out-of-sync` status means that at least one resource status is not matching to the expected state or not found in the target cluster.

The health status aggregates information about the observed health status of each resource that makes up the application. The health assessment logic is different for each Kubernetes resource type and results in one of the following values:

- *Healthy*—For example, the Kubernetes deployment is considered healthy if the required number of Pods is running and each Pod successfully passes both readiness and liveness probes.
- *Progressing*—Represents a resource that is not healthy yet but is still expected to reach a healthy state. The Deployment is considered progressing if it is not healthy yet but still without a time limit specified by the `progressingDeadlineSeconds`<sup>9</sup> field.

---

<sup>9</sup> <http://mng.bz/aomz>.

- *Degraded*—The antipode of a healthy status. The example is a Deployment that could not reach a healthy status within an expected timeout.
- *Missing*—Represents the resource that is stored in Git but not deployed to the target cluster.

The aggregated application status is the worst status of every application resource. The healthy status is the best, descending to progressing, degraded, and missing (the worst). So if all application resources are healthy and only one is degraded, the aggregated status is also degraded.

### **Exercise 2.3**

Consider an application consisting of two Deployments. The following information is known about the resources:

- Deployment 1 has an image that does not match the image stored in the Git repository. All Deployment Pods have failed to start for several hours while Deployment `progressingDeadlineSeconds` is set to 10 minutes.
- Deployment 2 is not fully matching the expected state and has all Pods running.

What are the application sync and health statuses?

The health and sync statuses answer the two most important questions about an application:

- Is my application working?
- Am I running what is expected?

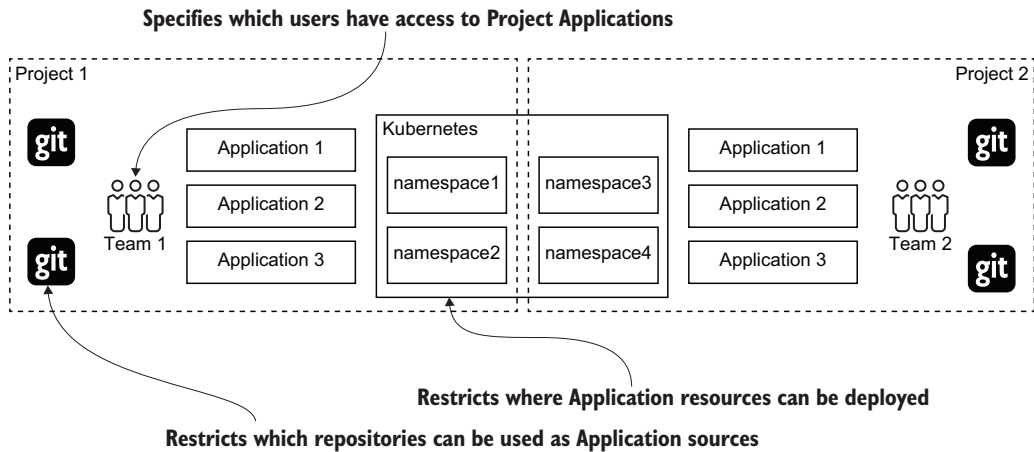
### **PROJECT**

Argo CD applications provide a very flexible way to manage different applications independently of each other. This functionality provides very useful insights to the team about each piece of infrastructure and greatly improves productivity. However, this is not enough to support multiple teams with different access levels:

- The mixed list of applications creates confusion that creates a human error possibility.
- Different teams have different access levels. A individual might use the GitOps operator to escalate their own permissions to get full cluster access.

The workaround for these issues is a separate Argo CD instance for each team. This is not a perfect solution since a separate instance means management overhead. In order to avoid management overhead, Argo CD introduces the Project abstraction. Figure 2.2 illustrates the relationship between Applications and Projects.

A Project provides a logical grouping of Applications, isolates teams from each other, and allows for fine-tuning access control in each Project.



**Figure 2.2** Demonstrates the relationship between Applications and Projects. A Project provides a logical grouping of Applications, isolating teams from each other and enabling using Argo CD in multitenant environments.

In addition to separating sets of applications, a Project provides the following set of features:

- Restricts which Kubernetes clusters and Git repositories might be used by Project Applications
- Restricts which Kubernetes resources can be deployed by each Application within a Project

#### **Exercise 2.4**

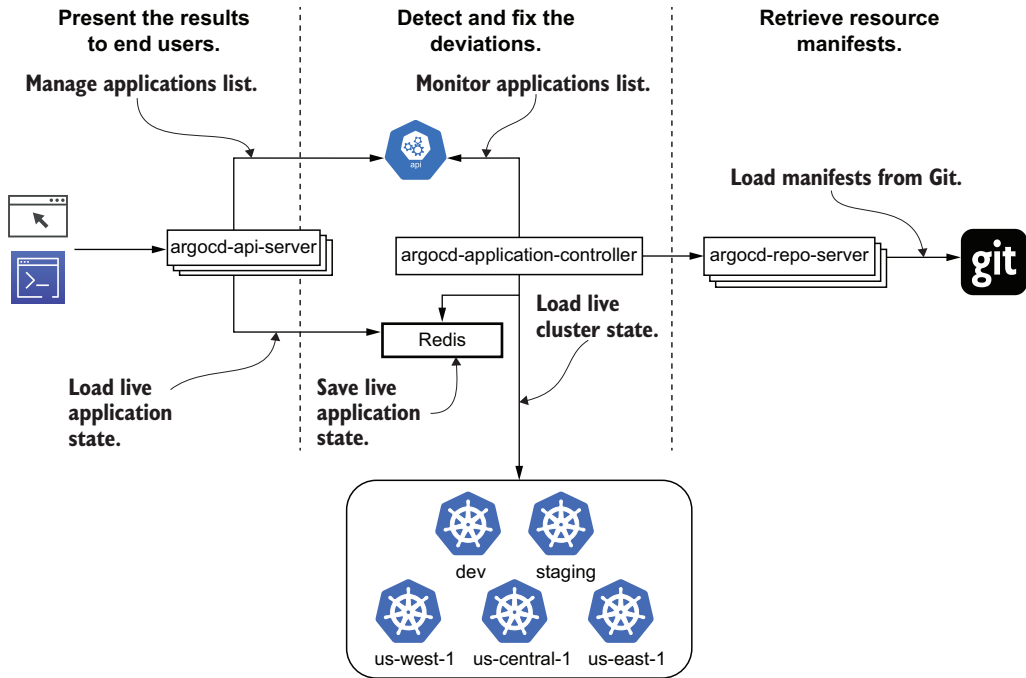
Try to come up with a list of projects in your organization. Using Projects, you can restrict what kind of resource users can deploy, source repositories, and destination clusters available within the Project. Which restrictions would you configure for your projects?

### **2.1.4 Architecture**

At first glance, the implementation of the GitOps operator does not look too complex. In theory, all you need is to clone the Git repository with manifests and use `kubectl diff` and `kubectl apply` to detect and handle config drifts. This is true until you are trying to automate this process for multiple teams and manage the configuration of dozens of clusters simultaneously. Logically this process is split into three phases, and each phase has its own challenges:

- Retrieve resource manifests.
- Detect and fix the deviations.
- Present the results to end users.

Each phase consumes different resources, and the implementation of each phase has to scale differently. A separate Argo CD component is responsible for each phase.



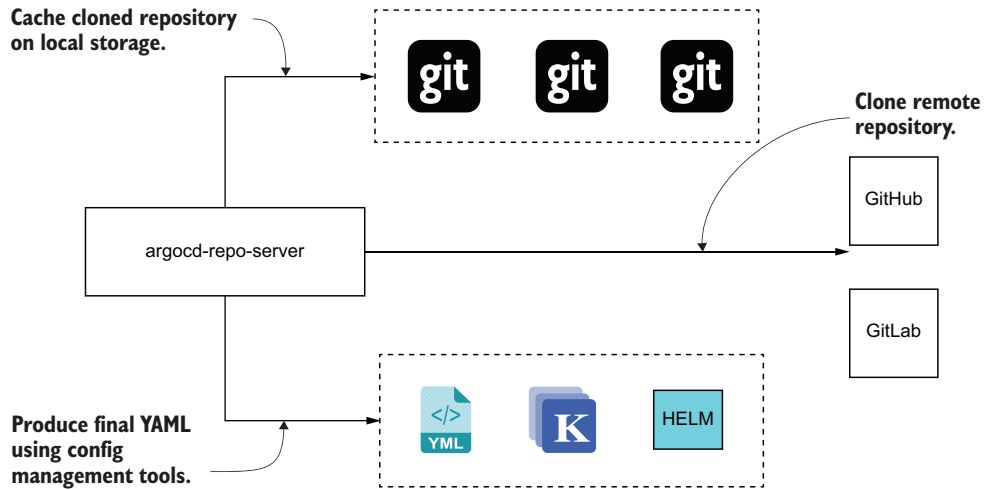
**Figure 2.3** Argo CD consists of three main components that implement GitOps reconciliation cycle phases. The `argocd-repo-server` retrieves manifests from Git. The `argocd-application-controller` compares manifests from Git with resources in the Kubernetes cluster. The `argocd-api-server` presents reconciliation results to the user.

Let's go through each phase and the corresponding Argo CD component implementation details.

### RETRIEVE RESOURCE MANIFESTS

The manifest generation in Argo CD is implemented by the `argocd-repo-server` component. This phase presents a whole set of challenges.

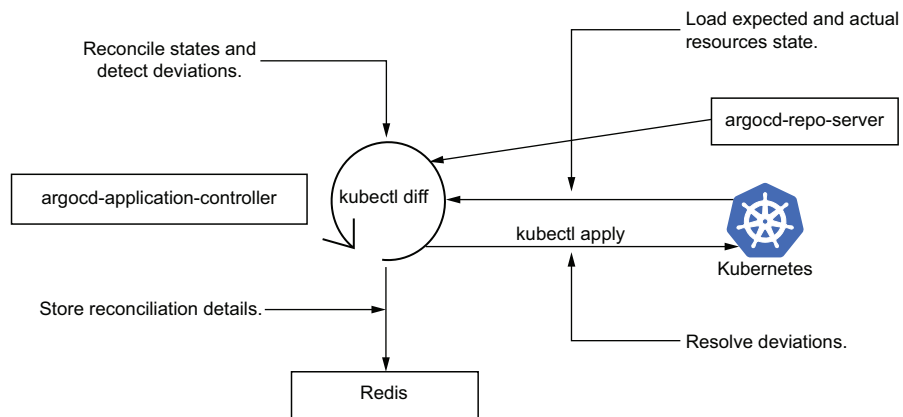
Manifest generation requires you to download Git repository content and produce ready-to-use manifest YAML. First of all, it is too time consuming to download the whole repository content every time you need to retrieve expected resource manifests. Argo CD solves this by caching the repository content on local disk and using the `git fetch` command to download only recent changes from the remote Git repository. The next challenge is related to memory usage. In real life, resource manifests are rarely stored as plain YAML files. In most cases, developers prefer to use a config management tool such as Helm or Kustomize. Every tool invocation causes a spike in memory usage. To handle the memory usage issues, Argo CD allows the user to limit the number of parallel manifest generations and scale up the number of `argocd-repo-server` instances to improve performance.



**Figure 2.4** `argocd-repo-server` caches the cloned repository on local storage and encapsulates interaction with the config management tool that is required to produce final resource manifests.

### DETECT AND FIX THE DEVIATIONS

The reconciliation phase is implemented by the `argocd-application-controller` component. The controller loads the live Kubernetes cluster state, compares it with the expected manifests provided by the `argocd-repo-server`, and patches deviated resources. This phase is probably the most challenging one. In order to correctly detect deviations, the GitOps operator needs to know about each resource in the cluster, and compare and update thousands of resources.



**Figure 2.5** `argocd-application-controller` performs resource reconciliation. The controller leverages the `argocd-repo-server` component to retrieve expected manifests and compare manifests with the lightweight in-memory Kubernetes cluster state cache.

The controller maintains a lightweight cache of each managed cluster and updates it in the background using the Kubernetes watch API. This allows the controller to perform reconciliation on an application within a fraction of a second and empowers it to scale and manage dozens of clusters simultaneously. After each reconciliation, the controller has exhaustive information about each application resource, including the sync and health status. The controller saves that information into the Redis cluster so it can be presented to the end user later.

#### **PRESENT THE RESULTS TO END USERS**

Finally, the reconciliation results must be presented to end users. This task is performed by the `argocd-server` component. While the heavy lifting was already done by the `argocd-repo-server` and `argocd-application-controller`, this last phase has the highest resiliency requirements. The `argocd-server` is a stateless web application that loads the information about reconciliation results and powers the web user interface.

The architecture design allows Argo CD to serve GitOps operations for large enterprises with minimal maintenance overhead.

#### **Exercise 2.5**

Which components serve user requests and require multiple replicas for resiliency? Which components might require a lot of memory to scale?

## **2.2 Deploy your first application**

While Argo CD is an enterprise-ready, complex distributed system, it is still lightweight and can easily run on minikube. The installation is trivial and includes a few simple steps. To install Argo CD, follow the official Argo CD instructions.<sup>10</sup>

### **2.2.1 Deploying the first application**

As soon as Argo CD is running, we are ready to deploy our first application. As it's been mentioned before, to deploy an Argo CD application, we need to specify the Git repository that contains deployment manifests and target the Kubernetes cluster and Namespace. To create the Git repository for this exercise, open the following GitHub repository and create a repository fork:<sup>11</sup>

`https://github.com/gitopsbook/sample-app-deployment`

Argo CD can deploy into the external cluster as well as into the same cluster where it is installed. Let's use the second option and deploy our application into the default Namespace of our minikube cluster.

**RESET YOUR FORK** Have you already forked the deployment repository while working on previous chapters? Please make sure to revert changes for the best experience. The simplest way is to delete the previously forked repository and fork it again.

---

<sup>10</sup> [https://argoproj.github.io/argo-cd/getting\\_started/](https://argoproj.github.io/argo-cd/getting_started/).

<sup>11</sup> <https://help.github.com/en/github/getting-started-with-github/fork-a-repo>.



The application might be created using the web user interface, using the CLI, or even programmatically using the REST or gRPC APIs. Since we already have Argo CD CLI installed and configured, let's use it to deploy an application. Go ahead and execute the following command to create an application:

```
$ argocd app create sample-app \
  --repo https://github.com/<username>/sample-app-deployment \
  --path . \
  --dest-server https://kubernetes.default.svc \
  --dest-namespace default
```

Unique application name  
 Git repository URL  
 Directory path within the Git repository  
 The Kubernetes Namespace name  
 The Kubernetes API server URL. The <https://kubernetes.default.svc/> is the API server URL that is available inside of every Kubernetes cluster.

As soon as the application is created, we can use the Argo CD CLI to get the information about the application state. Use the following command to get the information about the `sample-app` application state:

```
argocd app get sample-app
Name:                sample-app
Project:             default
Server:              https://kubernetes.default.svc
Namespace:          default
URL:                 https://<host>:<port>/applications/sample-app
Repo:                https://github.com/<username>/sample-app-deployment
Target:              .
Path:                .
SyncWindow:         Sync Allowed
Sync Policy:         <none>
Sync Status:        OutOfSync from (09d6663)
Health Status:      Missing
```

CLI command that returns the information about an application state  
 Application sync status that answers whether the application state matches the expected state or not  
 Application aggregated health status

GROUP	KIND	NAMESPACE	NAME	STATUS	HEALTH	HOOK	MESSAGE
	Service	default	sample-app	OutOfSync	Missing		
apps	Deployment	default	sample-app	OutOfSync	Missing		

As we can see from the command output, the application is out of sync and not healthy. By default, Argo CD does not push resources defined in the Git repository into the cluster if it detects a deviation. In addition to the high-level summary, we can see the details of every application resource. Argo CD detected that the application is supposed to have a Deployment and a Service, but both resources are missing. To deploy the resources, we need to either configure automated application syncing

using the sync policy<sup>12</sup> or trigger syncing manually. To trigger the sync and deploy the resources, use the following command:

```

$ argocd app sync sample-app

```

TIMESTAMP	HEALTH	HOOK	MESSAGE	GROUP	KIND	NAMESPACE	NAME	STATUS
2020-03-17T23:16:50-07:00	OutOfSync	Missing		Service	default	sample-app	sample-app	Missing
2020-03-17T23:16:50-07:00	OutOfSync	Missing		apps	Deployment	default	sample-app	Missing

**CLI command that triggers application sync**

**Initial application state before the sync operation**

```

Name:                sample-app
Project:             default
Server:              https://kubernetes.default.svc
Namespace:          default
URL:                 https://<host>:<port>/applications/sample-app
Repo:                https://github.com/<username>/sample-app-deployment
Target:
Path:                .
SyncWindow:         Sync Allowed
Sync Policy:         <none>
Sync Status:         OutOfSync from (09d6663)
Health Status:      Missing

Operation:           Sync
Sync Revision:       09d6663dcfa0f39b1a47c66a88f0225a1c3380bc
Phase:               Succeeded
Start:               2020-03-17 23:17:12 -0700 PDT
Finished:            2020-03-17 23:17:21 -0700 PDT
Duration:            9s
Message:             successfully synced (all tasks run)

```

**Final application state after the sync is completed**

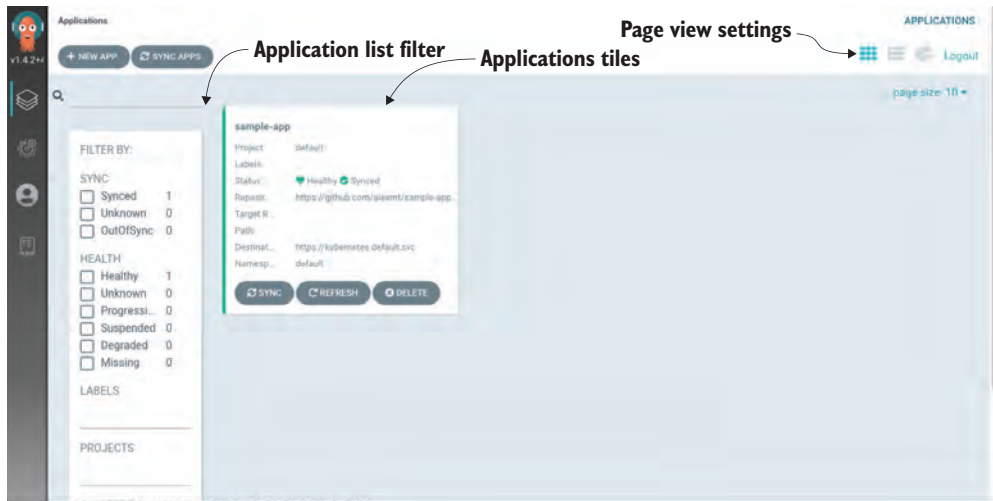
GROUP	KIND	NAMESPACE	NAME	STATUS	HEALTH	HOOK	MESSAGE
	Service	default	sample-app	Synced	Healthy		
	service/sample-app created						
apps	Deployment	default	sample-app	Synced	Progressing		deployment
	.apps/sample-app created						

As soon as the sync is triggered, Argo CD pushes the manifests stored in Git into the Kubernetes cluster and then reevaluates the application state. The final application state is printed to the console when the synchronization completes. The `sample-app` application was successfully synced, and each result matches the expected state.

## 2.2.2 *Inspect the application using the user interface*

In addition to the CLI and API, Argo CD provides a user-friendly web interface. Using the web interface, you might get the high-level view of all your applications deployed across multiple clusters as well as very detailed information about every application resource. Open the `https://<host>:<port>` URL to see the applications list in the Argo CD user interface.

<sup>12</sup> [https://argoproj.github.io/argo-cd/user-guide/auto\\_sync/](https://argoproj.github.io/argo-cd/user-guide/auto_sync/).



**Figure 2.6** Application list page showing available Argo CD applications. The page provides high-level information about each application, such as sync and health status.

The application list page provides high-level information about all deployed applications, including health and synchronization status. Using this page, you can quickly find if any of your applications have degraded or have configuration drift. The user interface is designed for large enterprises and able to handle hundreds of applications. You can use search and various filters to quickly find the desired applications.

### Exercise 2.6

Experiment with the filters and page view settings to learn which other features are available in the applications list page.

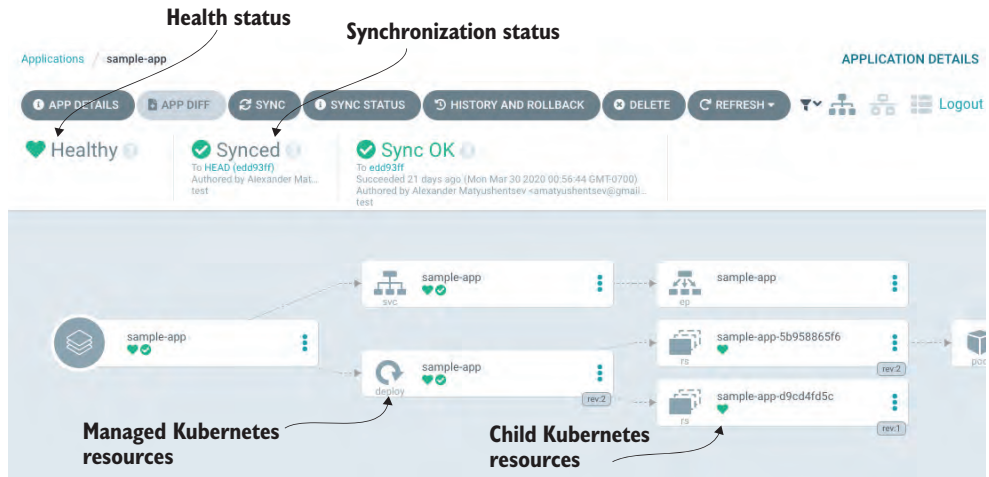
### APPLICATION DETAILS PAGE

The additional information about the application is available on the application details page. Navigate to the application details page by clicking on the “sample app” application tile.

The application details page visualizes the application resources hierarchy and provides additional details about synchronization and health status. Let’s take a closer look at the application resource tree and learn which features it provides.

The root element of the resource tree is the application itself. The next level consists of managed resources. The managed resources are resources that the manifest defined in Git and are controlled by Argo CD explicitly. As we’ve learned in chapter 1, Kubernetes controllers often leverage delegation and create child resources to delegate the work. The third and deeper levels represent such resources. That provides complete information about every application element and makes the application details page an extremely powerful Kubernetes dashboard.

In addition to this information, the user interface allows executing various actions against each resource. It is possible to delete any resource, re-create it by running sync



**Figure 2.7** The application details page provides information about the application resource hierarchy as well as detailed information about each resource.

actions, update the resource definition using a built-in YAML editor, and even run resource-specific actions such as Deployment restart.

### **Exercise 2.7**

Go ahead, use the application details page to inspect your application. Try to find how to view the resource manifests, locate Pods, and see the live logs.

## **2.3 Deep dive into Argo CD features**

So far, we've learned how to deploy new applications using Argo CD and get detailed application information using the CLI and the user interface. Next, let's learn how to deploy a new application version using GitOps and Argo CD.

### **2.3.1 GitOps-driven deployment**

In order to perform GitOps deployment, we need to update resource manifests and let the GitOps operator push changes into the Kubernetes cluster. As a first step, clone the Deployment repository using the following command:

```
$ git clone git@github.com:<username>/sample-app-deployment.git
$ cd sample-app-deployment
```

Next, use the following command to change the image version of the Deployment resource:

```
$ sed -i '' 's/sample-app:v.*/sample-app:v0.2/' deployment.yaml
```

Use the `git diff` command to make sure that your Git repository has the expected changes:

```
$ git diff
diff --git a/deployment.yaml b/deployment.yaml
index 5fc3833..397d058 100644
--- a/deployment.yaml
+++ b/deployment.yaml
@@ -16,7 +16,7 @@ spec:
   containers:
   - command:
     - /app/sample-app
     image: gitopsbook/sample-app:v0.1
+   image: gitopsbook/sample-app:v0.2
   name: sample-app
   ports:
   - containerPort: 8080
```

Finally, use `git commit` and `git push` to push changes to the remote Git repository:

```
$ git commit -am "update deployment image"
$ git push
```

Let's use the Argo CD CLI to make sure that Argo CD correctly detected manifest changes in Git and then triggered a synchronization process to push the changes into the Kubernetes cluster:

```
$ argocd app diff sample-app --refresh
===== apps/Deployment default/sample-app =====
21c21
<       image: gitopsbook/sample-app:v0.1
---
>       image: gitopsbook/sample-app:v0.2
```

### Exercise 2.8

Open the Argo CD UI and use the application details page to check the application sync status and inspect the managed resources status.

Use the `argocd sync` command to trigger the synchronization process:

```
$ argocd app sync sample-app
```

Great, you just performed GitOps deployment using Argo CD!

## 2.3.2 Resource hooks

Resource manifest syncing is just the basic use case. In real life, we often need to execute additional steps before and after actual deployment. For example, set the maintenance page, execute database migration before the new version deployment, and finally remove the maintenance page.

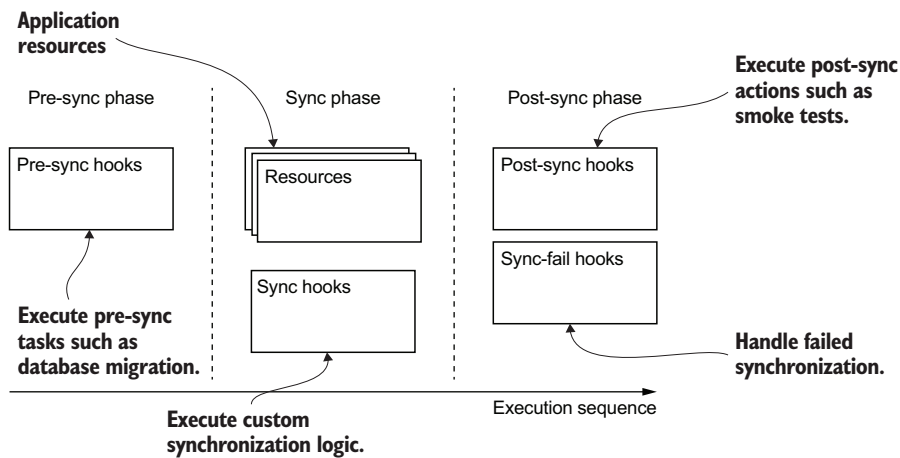
Traditionally these deployment steps are scripted in the CI pipeline. However, this again requires production access from the CI server, which involves a security threat. To solve that problem, Argo CD provides a feature called *resource hooks*. These hooks allow running custom scripts, typically packaged into a Pod or a Job, inside of the Kubernetes cluster during the synchronization process.

The hook is a Kubernetes resource manifest stored in the Git repository and annotated with the `argocd.argoproj.io/hook` annotation. The annotation value contains

a comma-separated list of phases when the hook is supposed to be executed. The following phases are supported:

- *Pre-sync*—Executes prior to the applying of the manifests
- *Sync*—Executes after all pre-sync hooks completed and were successful, at the same time as the apply of the manifests
- *Skip*—Indicates to Argo CD to skip the apply of the manifest
- *Post-sync*—Executes after all sync hooks completed and were successful, a successful apply, and all resources in a healthy state
- *Sync-fai*—Executes when the sync operation fails

The hooks are executed inside of the cluster, so there is no need to access the cluster from the CI pipeline. The ability to specify the sync phase provides the necessary flexibility and allows a mechanism to solve the majority of real-life deployment use cases.



**Figure 2.8** The synchronization process includes three main phases. The pre-sync phase is used to execute preparation tasks such as database migration. The sync phase includes the synchronization of application resources. Finally, the post-sync phase runs postprocessing tasks, such as email notifications.

It is time to see the hooks feature in action! Add the hook definition into the sample app deployment repository and push changes to the remote repository:

```
$ git add pre-sync.yaml
$ git commit -am 'Add pre-sync hook'
$ git push
```

**Listing 2.1** <http://mng.bz/go7l>

```
apiVersion: batch/v1
kind: Job
metadata:
```

```

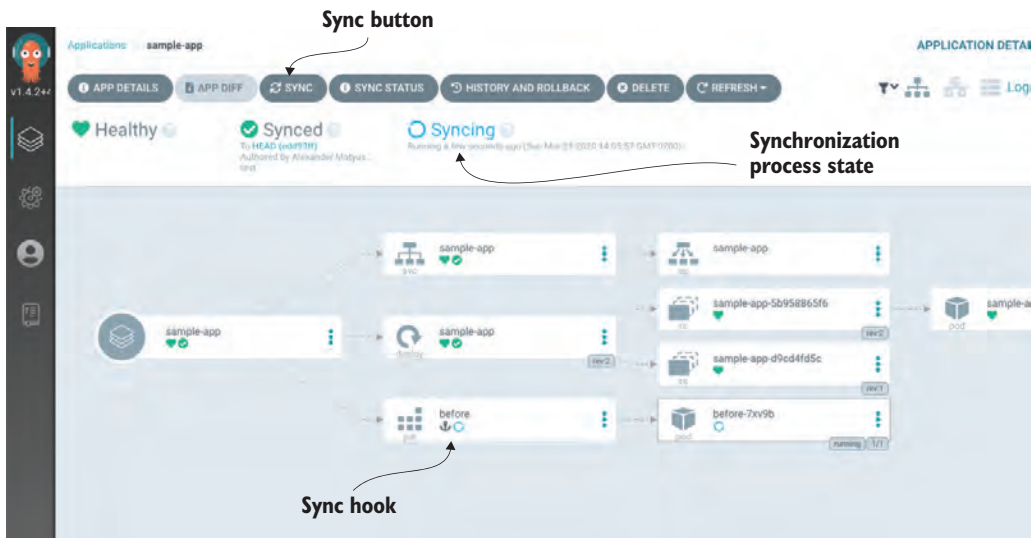
name: before
annotations:
  argocd.argoproj.io/hook: PreSync
spec:
  template:
    spec:
      containers:
      - name: sleep
        image: alpine:latest
        command: ["echo", "pre-sync"]
        restartPolicy: Never
      backoffLimit: 0

```

The Argo CD user interface provides much better visualization of a dynamic process than the CLI. Let's use it to better understand how hooks work. Open the Argo CD UI using the following command:

```
$ minikube service argocd-server -n argocd --url
```

Navigate to the `sample-app` details page and trigger the synchronization process using the Sync button. The syncing process is represented in figure 2.9.



**Figure 2.9** The application detail page allows the user to trigger the synchronization as well as view detailed information about the synchronization progress, including synchronization hooks.

As soon as the sync is started, the application details page shows live process status in the top-right corner. The status includes information about operation start time and duration. You can view the syncing status panel with detailed information, including sync hook results, by clicking the Sync Status icon.

The hooks are stored as the regular resource manifests in the Git repository and also visualized as regular resources in the Application resource tree. You can see the real-time status of the “before” job and use the Argo CD user interface to inspect child Pods.

In addition to phases, you might customize the hook deletion policy. The deletion policy allows automating hook resources deletion that will save you a lot of manual work.

### **Exercise 2.9**

Read more details in the Argo CD documentation<sup>13</sup> and change the “before” job deletion policy. Use the Argo CD user interface to observe how various deletion policies affect hook behavior. Synchronize the application and observe how hook resources got created and deleted by Argo CD.

### **2.3.3 Postdeployment verification**

Resource hooks allow encapsulating the application synchronization logic, so we don’t have to use scripts and continuous integration tools. However, some of such use cases naturally belong to continuous integration processes, and it is still preferable to use tools like Jenkins.

One such use case is postdeployment verification. The challenge here is that GitOps deployment is asynchronous by nature. After the commit is pushed to the Git repository, we still need to make sure that changes are propagated to the Kubernetes cluster. Even after changes are propagated, it is not safe to start running tests. In most cases, the update of a Kubernetes resource is not instant, either. For example, the Deployment resource update triggers the rolling-update process. The rolling update might take several minutes or even fail if the new application version has an issue. So if you start tests too early, you might end up testing the previously deployed application version.

Argo CD makes this issue trivial by providing tools that help to monitor application status. The `argocd app wait` command monitors the application and exits after the application reaches a synced and healthy state. As soon as the command exits, you can assume that all changes are successfully rolled out, and it is safe to start postdeployment verification. The `argocd app wait` command is often used in conjunction with `argocd app sync`. Use the following command to synchronize your application and wait until the change is fully rolled out, and the application is ready for testing:

```
$ argocd app sync sample-app && argocd app wait sample-app
```

## **2.4 Enterprise features**

Argo CD is pretty lightweight, and it is really easy to start using it. At the same time, it scales well for a large enterprise and is able to accommodate the needs of multiple teams. The enterprise features can be configured as you go. If you are rolling out an Argo CD for your organization, then the first question is how to configure the end user and effectively manage access control.

---

<sup>13</sup> <http://mng.bz/e5Ez>.



### 2.4.1 Single sign-on

Instead of introducing its own user management system, Argo CD provides integration with multiple SSO services. The list includes Okta, Google OAuth, Azure AD, and many more.

**SSO** SSO is a session and user authentication service that allows a user to use one set of login credentials to access multiple applications.

The SSO integration is great because it saves you a lot of management overhead, and end users don't have to remember another set of login credentials. There are several open standards for exchanging authentication and authorization data. The most popular ones are SAML, OAuth, and OpenID Connect (OIDC). Of the three, SAML and OIDC satisfy the best requirements of a typical enterprise and can be used to implement SSO. Argo CD decided to go ahead with OIDC because of its power and simplicity.

The number of steps required to configure an OIDC integration depends on your OIDC provider. The Argo CD community already contributed a number of instructions for popular OIDC providers such as Okta and Azure AD. After performing the configuration on the OIDC provider side, you need to add the corresponding configuration to the `argocd-cm` ConfigMap. The following snippet represents the sample Okta configuration:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: argocd-cm
  namespace: argocd
  labels:
    app.kubernetes.io/name: argocd-cm
    app.kubernetes.io/part-of: argocd
data:
  url: https://<myargocdhost>
  oidc.config: |
    name: Okta
    issuer: https://yourorganization.oktapreview.com
    clientID: <your client id>
    clientSecret: <your client secret>
    requestedScopes: ["openid", "profile", "email", "groups"]
    requestedIDTokenClaims: {"groups": {"essential": true}}
```

The externally facing base URL Argo CD URL

OIDC configuration that includes Okta application client id and secret

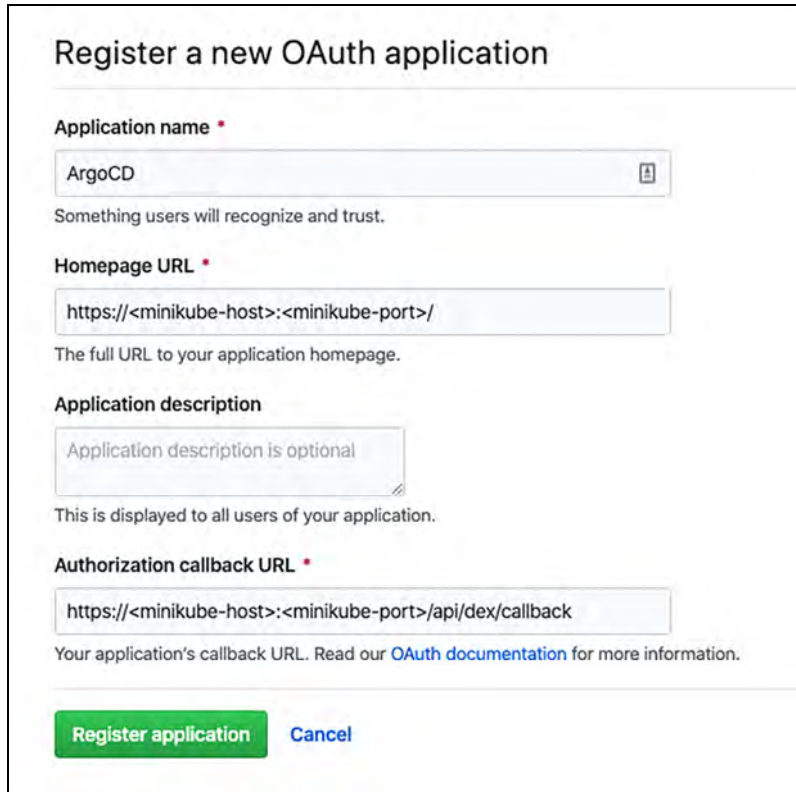
What if your organization does not have an OIDC-compatible SSO service? In this case, you can use a federated OIDC provider, Dex,<sup>14</sup> which is bundled into the Argo CD by default. Dex acts as a proxy to other identity providers and allows establishing integration with SAML, LDAP providers, or even services like GitHub and Active Directory.

GitHub often is a very attractive option, especially if it is already used by developers in your organization. Additionally, organizations and teams configured in GitHub nat-

<sup>14</sup> <https://github.com/dexidp/dex>.

usually fit the access control model required to organize cluster access. As you are going to learn soon, it is very easy to model Argo CD access using the GitHub team membership. Let's use GitHub to enhance our Argo CD installation and enable SSO integration.

First of all, we need to create a GitHub OAuth application. Navigate to <https://github.com/settings/applications/new> and configure the application settings as represented in figure 2.10.



**Register a new OAuth application**

**Application name \***

  
Something users will recognize and trust.

**Homepage URL \***

  
The full URL to your application homepage.

**Application description**

  
This is displayed to all users of your application.

**Authorization callback URL \***

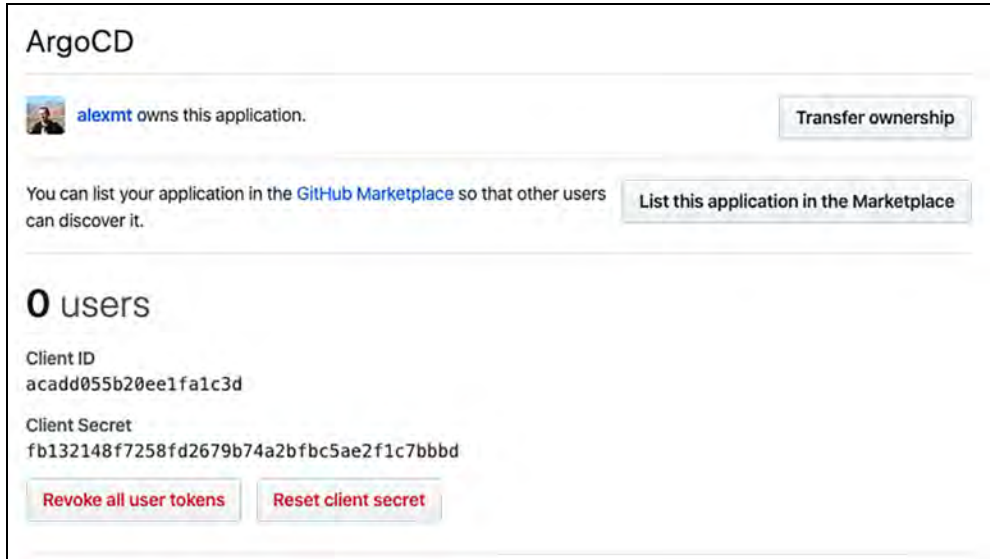
  
Your application's callback URL. Read our [OAuth documentation](#) for more information.

**Register application** **Cancel**

**Figure 2.10** New GitHub OAuth application settings include the application name and description, home page URL, and, most importantly, the authorization callback URL.

Specify the application name of your choice and the home page URL that matches the Argo CD web user interface URL. The most important application setting is the callback URL. The callback URL value is the Argo CD web user interface URL plus the `/api/dex/callback` path. The sample URL with minikube might be `http://192.168.64.2:32638/api/dex/callback`.

After creating the application, you will be redirected to the OAuth application settings page. Copy the application Client ID and Client Secret. These values will be used to configure the Argo CD settings.



**Figure 2.11** The GitHub OAuth application settings page displays the Client ID and Client Secret values, which are required to configure the SSO integration.

Substitute the placeholder values in the `argocd-cm.yaml` file with your environment values.

**Listing 2.2** <http://mng.bz/pV1G>

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: argocd-cm
  labels:
    app.kubernetes.io/name: argocd-cm
    app.kubernetes.io/part-of: argocd
data:
  url: https://<minikube-host>:<minikube-port>
  dex.config: |
    connectors:
      - type: github
        id: github
        name: GitHub

```

← The externally facing base URL Argo CD URL

```

config:
  clientID: <client-id>
  clientSecret: <client-secret>
  loadAllGroups: true

```

← GitHub OAuth application client ID

← GitHub OAuth application client secret

Update the Argo CD ConfigMap using the `kubectl apply` command:

```
$ kubectl apply -f ./argocd-cm.yaml -n argocd
```

You are ready to go! Open the Argo CD user interface in the browser and use the Login Via GitHub button.

## 2.4.2 Access control

You might notice that after a successful login using GitHub SSO integration, the application list page is empty. If you try creating a new application, you will see a “permission denied” error. This behavior is expected because we have not given any permission to the new SSO user yet. In order to provide the user with appropriate access, we need to update the Argo CD access control settings.

Argo CD provides a flexible role-based access control (RBAC) system whose implementation is based on Casbin,<sup>15</sup> a powerful open source access control library. Casbin provides a very solid foundation and allows configuring various access control rules.

The RBAC Argo CD settings are configured using `argocd-rbac-cm` ConfigMap. To quickly dive into the configuration details, let’s update the ConfigMap fields and then go together through each change.

Substitute the `<username>` placeholder with your GitHub account username in the `argocd-rbac-cm.yaml` file.

### Listing 2.3 <http://mng.bz/OEPn>

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: argocd-rbac-cm
  labels:
    app.kubernetes.io/name: argocd-rbac-cm
    app.kubernetes.io/part-of: argocd
data:
  policy.csv: |
    p, role:developer, applications, *, **/, allow
    g, role:developer, role:readonly

    g, <username>, role:developer

  scopes: '[groups, preferred_username]'

```

← The policy.csv contains role-based access rules.

← The scopes setting specifies which JWT claim is used to infer user groups.

<sup>15</sup> <https://github.com/casbin/casbin>.

Apply the RBAC changes using the `kubectl apply` command:

```
$ kubectl apply -f ./argocd-rbac-cm.yaml -n argocd
```

The `policy.csv` field in this configuration defines a role named `role:developer` with full permissions on Argo CD applications and read-only permissions over Argo CD system settings. The role is granted to any user that belongs to a group whose name matches your GitHub account username. As soon as changes are applied, refresh the applications list page and try syncing the `sample-app` application.

We've introduced quite a few new terms. Let's step back and discuss what roles, groups, and claims are and how they work together.

### ROLE

The role allows or denies a set of actions on an Argo CD object to a particular subject. The role is defined in the form

```
p, subject, resource, action, object, effect
```

where

- `p` indicates the RBAC policy line.
- `subject` is a group.
- `resource` is one of the Argo CD resource types. Argo CD supports the following resources: "clusters", "projects", "applications", "repositories", "certificates", "accounts".
- `action` is an action name that might be executed against a resource. All Argo CD resources support the following actions: "get", "create", "update", "delete". The "\*" value matches any action.
- `object` is a pattern that identifies a particular resource instance. The "\*" value matches any instance.
- `effect` defines whether the role grants or denies the action.

The `role:developer` role from this example allows any action against any Argo CD application:

```
p, role:developer, applications, *, */*, allow
```

### GROUP

A group provides the ability to identify a set of users and works in conjunction with OIDC integration. After performing the successful OIDC authentication, the end user receives a JWT token that verifies the user identity as well as provides additional metadata stored in the token claims.

**JWT TOKEN** A JWT token is an internet standard for creating JSON-based access tokens that assert some number of claims.<sup>16</sup>

---

<sup>16</sup> [https://en.wikipedia.org/wiki/JSON\\_Web\\_Token](https://en.wikipedia.org/wiki/JSON_Web_Token).

The token is supplied with every Argo CD request. The Argo CD extracts the list of groups that a user belongs to from a configured list of token claims and uses it to verify user permissions.

Following is a token claims example generated by Dex:

```
{
  "iss": "https://192.168.64.2:32638/api/dex",
  "sub": "CgY0MjY0MzcSBmdpdGh1Yg",
  "aud": "argo-cd",
  "exp": 1585646367,
  "iat": 1585559967,
  "at_hash": "rAz6dDHs1BWvU6PiWj_o9g",
  "email": "AMatyushentsev@gmail.com",
  "email_verified": true,
  "groups": [
    "gitopsbook"
  ],
  "name": "Alexander Matyushentsev",
  "preferred_username": "alexmt"
}
```

The token contains two claims that might be useful for authorization:

- `groups` includes a list of GitHub organizations and teams the user belongs to.
- `preferred_username` is the GitHub account username.

By default, Argo CD uses `groups` to retrieve user groups from the JWT token. We've added the `preferred_username` claim using the `scopes` setting to allow identifying GitHub users by name.

### **Exercise 2.10**

Update the `argocd-rbac-cm` ConfigMap to provide admin access to the GitHub user based on their email.

**NOTE** This chapter covers important foundations of Argo CD and gets you ready for further learning. Explore the Argo CD documentation to learn about diffing logic customization, fine-tuning config management tools, advanced security features such as auth tokens, and much more. The project keeps evolving and getting new features in every release. Check out the Argo CD blog to stay up to date with the changes, and don't hesitate to ask questions in the Argoproj slack channel.

### **2.4.3 Declarative management**

As you might've noticed, Argo CD provides a lot of configuration settings. The RBAC policies, SSO settings, Applications, and Projects—all of those are settings that have to be managed by someone. The good news is that you can leverage GitOps and use Argo CD to manage itself!

All Argo CD settings are persisted in Kubernetes resources. The SSO and RBAC settings stored in ConfigMap and Applications and Projects are stored in custom

resources, so you can store these resource manifests in a Git repository and configure Argo CD to use it as a source of truth. This technique is very powerful and allows us to manage configuration settings as well as seamlessly upgrade the Argo CD version.

As a first step, let's demonstrate how to convert the SSO and RBAC changes we've just made imperatively into a declarative configuration. To do so we would need to create a Git repository that stores manifest definitions of every Argo CD component. Instead of starting from scratch, you can just use the code listings in the repository at <https://github.com/gitopsbook/resources> as a starting point. Navigate to the repository GitHub URL and create your personal fork so you can store settings specific to your environment.

The required manifest files are located in the chapter-09 directory, and the first file we should look at is represented in the following listing.

**Listing 2.4** <http://mng.bz/YqRN>

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
resources:
- https://raw.githubusercontent.com/argoproj/argo-
  cd/stable/manifests/install.yaml

patchesStrategicMerge:
- argocd-cm.yaml
- argocd-rbac-cm.yaml
- argocd-server.yaml
```

The remote file URL containing default Argo CD manifests

The file path that contains argocd-cm ConfigMap modifications

The file path that contains argocd-rbac-cm ConfigMap modifications

The file path that contains argocd-server Service modifications

The kustomization.yaml file contains references to the default Argo CD manifests and files with the environment-specific changes.

The next step is to move your environment-specific changes into Git and push them into the remote Git repository. Clone the forked Git repository:

```
$ git clone git@github.com:<USERNAME>/resources.git
```

Repeat the changes to the argocd-cm.yaml and argocd-rbac-cm.yaml files described in sections 2.4.1 and 2.4.2. Add SSO configuration to the ConfigMap manifest in argocd-cm.yaml. Update the RBAC policy in the argocd-rbac-cm.yaml file. Once the files are updated, commit and push the changes back to the remote repository:

```
$ git commit -am "Update Argo CD configuration"
$ git push
```

The hardest part is done! Argo CD config changes are not version controlled and can be managed using GitOps methodology. The last step is to create an Argo CD application that deploys Kustomize-based manifests from your Git repository into the argocd Namespace:

```
$ argocd app create argocd \
--repo https://github.com/<USERNAME>/resources.git \
--path chapter-09 \
--dest-server https://kubernetes.default.svc \
--dest-namespace argocd \
--sync-policy auto
application 'argocd' created
```

As soon as the application is created, Argo CD should detect already deployed resources and visualize the detected deviations.

So how about managing applications and projects? Both are represented by the Kubernetes custom resource and might be managed using GitOps as well. The manifest in the next listing represents the declarative definition of the `sample-app` Argo CD application that we created manually earlier in the chapter. In order to start managing `sample-app` declaratively, add the `sample-app.yaml` into the resources section of `kustomization.yaml` and push the change back to your repository fork.

### Listing 2.5 <http://mng.bz/Gx9q>

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: sample-app
spec:
  destination:
    namespace: default
    server: https://kubernetes.default.svc
  project: default
  source:
    path: .
    repoURL: https://github.com/<username>/sample-app-deployment
```

As you can see, you don't have to choose between declarative and imperative management styles. Argo CD supports using both simultaneously so that some settings are managed using GitOps and some are managed using imperative commands.

## Summary

- Argo CD is designed with enterprises in mind and can be offered as a centralized service to support multitenancy and multiclustering for large enterprises.
- As a continuous deployment tool, Argo CD also provides detail diff among Git, target Kubernetes clusters, and running states for observability.
- Argo CD automates three phases in deployment:
  - Retrieve resource manifests.
  - Detect and fix the deviations.
  - Present the results to end users.
- Argo CD provides CLI for configuring Application deployment and can be incorporated into CI solutions through scripting.



- Argo CD's CLI and web interface can be used to inspect applications' sync and health statuses.
- Argo CD provides resource hooks to enable additional customization of the deployment life cycle.
- Argo CD also provides support to ensure deployment completion and application readiness.
- Argo CD supports both SSO and RBAC integration for enterprise-level SSO and access control.

# appendix A

## Set up a test Kubernetes cluster

---

A full production-capable Kubernetes cluster is a very complex system consisting of multiple components that must be installed and configured based on your particular needs. How to deploy and maintain Kubernetes in production goes way beyond the focus of this book and is covered elsewhere.

Luckily for us, there are several projects which handle the configuration complexity and allow running Kubernetes locally with a single CLI command. Running a Kubernetes locally on your laptop is useful to get your hands dirty with Kubernetes and prepare you for completing the exercises in the remainder of this book. To the extent possible, all remaining exercises will utilize a cluster running on your laptop using an application called minikube. However, if you prefer to use your own cluster running on a cloud provider (or even on-premises) the exercises will work there as well.

**MINIKUBE** Minikube is an official tool maintained by the Kubernetes community to create a single-node Kubernetes cluster inside a VM on your laptop and supports macOS, Linux, and Windows. In addition to actually running the cluster, minikube provides features which simplify accessing services inside Kubernetes, volume management and many more.

Other projects you can consider to use are

- *Docker for desktop*—If you are using Docker on your laptop you might already have Kubernetes installed! Starting with version 18.6.0 both Windows and Mac Docker-for-Desktop comes with bundled Kubernetes binaries and developer productivity features.
- *K3S*—As the name implies, K3S is a lightweight Kubernetes deployment. According to authors K3S is five less than eight so K8S minus five is K3S. Besides the funny name K3S is indeed extremely lightweight, fast, and a

great choice if you need to run Kubernetes as part of CI job or on a hardware with limited resources. Installation instructions are available at <https://k3s.io>.

- **KIND**—Another tool developed by the Kubernetes community. KIND was developed by maintainers for Kubernetes v1.11+ conformance testing. Installation instructions are available at <https://kind.sigs.k8s.io/>.

While all listed tools simplify Kubernetes deployment to a single CLI command and provide great experience, minikube is still the safest choice to get started with Kubernetes. With all platforms support, thanks to virtualization, and a great set of developer productivity features this is a great tool for both beginners and experts. All exercises and samples in this book rely on minikube.

## A.1 Prerequisites for working with Kubernetes

The following tools and utilities are needed to work with Kubernetes.

In addition to Kubernetes itself, we need to install kubectl. The kubectl is a command line utility which allows interacting with Kubernetes control plane and allows doing virtually anything with Kubernetes.

### A.1.1 Configure kubectl

To get started go ahead and install minikube as described at <https://kubernetes.io/docs/tasks/tools/install-kubectl/>. If you are a macOS or Linux user you can complete the installation process in one step using homebrew package manager using the following command:

```
$ brew install kubectl
```

**HOME BREW** Homebrew is a free and open source software package management system that simplifies the installation of software on Apple's macOS operating system and Linux. More information is available at <https://brew.sh/>.

## A.2 Install minikube and create a cluster

minikube is an application that allows you to run a single-node Kubernetes cluster on your desktop or laptop machine. Installation instructions are available at <https://kubernetes.io/docs/tasks/tools/install-minikube/>.

Most (but not all) exercises in this book can be completed using a local minikube cluster.

### A.2.1 Configure minikube

The next step is to install and start the minikube cluster. The installation process is described at <https://minikube.sigs.k8s.io/docs/start/>. The minikube package is also available via Homebrew:

```
$ brew install minikube
```

If everything goes smoothly so far, we are ready to start minikube and configure our first deployment:

```
$ minikube start
(minikube/default)
🐳 minikube v1.1.1 on darwin (amd64)
👉 Creating virtualbox VM (CPUs=2, Memory=2048MB, Disk=20000MB) ...
👉 Configuring environment for Kubernetes v1.14.3 on Docker 18.09.6
👉 Pulling images ...
🚀 Launching Kubernetes ...
🕒 Verifying: apiserver proxy etcd scheduler controller dns
👉 Done! kubectl is now configured to use "minikube"
$ kubectl create -f nginx-Pod.yaml
Pod/nginx created
```

### **A.3 Create a GKE cluster in GCP**

The Google Cloud Platform (GCP) offers the Google Kubernetes Engine (GKE) as part of their free tier:

<https://cloud.google.com/free/>.

You can create a Kubernetes GKE cluster to run the exercises in this book:

<https://cloud.google.com/kubernetes-engine/>

Keep in mind that while GKE itself is free, you may be charged for other GCP resources that are created by Kubernetes. It is recommended that you delete your test cluster after completing each exercise in order to avoid unexpected costs.

### **A.4 Create an EKS cluster in AWS**

Amazon Web Services (AWS) offers a managed Kubernetes service called Elastic Kubernetes Service (EKS). You can create a free AWS account and create an EKS cluster to run the exercises in this book. However, while relatively inexpensive, EKS is not a free service (it costs \$0.20/hour at the time of this writing) and you may also be charged for other resources created by Kubernetes. It is recommended that you delete your test cluster after completing each exercise in order to avoid unexpected costs.

There is a tool called eksctl by Weaveworks which allows you to easily create an EKS Kubernetes cluster in your AWS account:

<https://github.com/weaveworks/eksctl/blob/master/README.md>

# appendix B

## Set up GitOps tools

---

This appendix will go over the step by step instruction to set up tools required for the tutorials in part 3.

### B.1 Install Argo CD

Argo CD supports several installation methods. You might use the official Kustomize based installation, the community maintained Helm chart,<sup>17</sup> or even the Argo CD operator<sup>18</sup> to manage the Argo CD deployments. The simplest possible installation method requires only to use a single YAML file. Please go ahead and use the following commands to install Argo CD into your minikube cluster:

```
$ kubectl create namespace argocd
$ kubectl apply -n argocd -f
    https://raw.githubusercontent.com/argoproj/argo-
    cd/stable/manifests/install.yaml
```

The command above installs all Argo CD components with the default settings that work for most users out of the box. For security reasons, the Argo CD UI and API are not exposed outside of the cluster by default. It is totally safe to open full access on minikube.

Enable load balancer access<sup>19</sup> in your minikube cluster by running the following command in a separate terminal:

```
$ minikube tunnel
```

Use the following command to open access to the “argocd-server” service and get the access URL:

```
$ kubectl patch svc argocd-server -n argocd -p '{"spec": {"type":
    "LoadBalancer"}}'
```

---

<sup>17</sup> <https://github.com/argoproj/argo-helm/tree/master/charts/argo-cd>

<sup>18</sup> <https://github.com/argoproj-labs/argocd-operator>

<sup>19</sup> <https://minikube.sigs.k8s.io/docs/handbook/accessing/#loadbalancer-access>

Argo CD provides both a web-based user interface and a command-line interface (CLI). To simplify the instructions, we are going to use the CLI tool in this tutorial. Let's go ahead and install the CLI tool. You might use the following command to install Argo CD CLI on Mac or follow the official getting started instructions<sup>20</sup> to install the CLI on your platform:

```
$ brew tap argoproj/tap
$ brew install argoproj/tap/argocd
```

As soon as Argo CD is installed, it has a preconfigured admin user. The initial admin password is auto-generated to be the pod name of the Argo CD API server that can be retrieved using the command below:

```
$ kubectl get pods -n argocd -l app.kubernetes.io/name=argocd-server -o name
| cut -d '/' -f 2
```

Use the following command to get the Argo CD server URL and update generated password using the Argo CD CLI:

```
$ argocd login <ARGOCD_SERVER-HOSTNAME>:<PORT>
$ argocd account update-password
```

The <ARGOCD\_SERVER-HOSTNAME>:<PORT> is a minikube API and service port that should be obtained from the Argo CD URL. The URL might be retrieved using the following command:

```
minikube service argocd-server -n argocd --url
```

The command returns the HTTP service URL. Make sure to remove `http://` and use only hostname and the port to login using Argo CD CLI.

Finally, login to the Argo CD user interface. Please open the Argo CD URL in the browser and login using the admin username and your password. You are ready to go!

## **B.2** *Install Jenkins X*

Jenkins X CLI depends on `kubectl`<sup>21</sup> and `Helm`<sup>22</sup> and will do its best to install those tools. However, the number of permutations of what we have on our laptops are close to infinite, so you're better off installing those tools first yourself.

**NOTE** At the time of this writing, February 2020, Jenkins X does not yet support Helm v3+. Please make sure that you're using Helm CLI v2+.

---

<sup>20</sup> [https://argoproj.github.io/argo-cd/cli\\_installation/#download-with-curl](https://argoproj.github.io/argo-cd/cli_installation/#download-with-curl)

<sup>21</sup> <https://kubernetes.io/docs/tasks/tools/install-kubectl/>

<sup>22</sup> [https://docs.helm.sh/using\\_helm/#installing-helm](https://docs.helm.sh/using_helm/#installing-helm)

### B.2.1 Prerequisites

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 poses two significant issues. You might prefer to practice locally using Minikube or Docker for desktop, but 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, or from anywhere else. Finally, we'll perform some GitHub operations using the command `hub`. Install it if you don't have it already.

**NOTE** Please refer to appendix A for more information on configuring AWS or GCP Kubernetes cluster.

For your convenience, the list of all the tools we'll use is as follows:

- `Git`
- `kubectl`
- `Helm`<sup>23</sup>
- `AWS CLI`
- `eksctl`<sup>24</sup> (if using AWS EKS)
- `gcloud` (if using Google GKE)
- `hub`<sup>25</sup>

Now let's install Jenkins X CLI:

```
$ brew tap jenkins-x/jx
$ brew install jx
```

### B.2.2 Installing Jenkins X In Kubernetes cluster

How can we install Jenkins X in a better way than what we're used to installing software? 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.

**NOTE** You can also refer to the Jenkins X site<sup>26</sup> for setting up Jenkins X in your Kubernetes cluster.

---

<sup>23</sup> [https://docs.helm.sh/using\\_helm/#installing-helm](https://docs.helm.sh/using_helm/#installing-helm)

<sup>24</sup> <https://github.com/weaveworks/eksctl>

<sup>25</sup> <https://hub.github.com/>

<sup>26</sup> <https://jenkins-x.io/docs/getting-started/setup/>

Let's take a look at the repository:

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

Once you see the repo in your browser, you will first create a fork under your GitHub account. We'll explore the files in the repo a bit later.

Next, we'll define a variable `CLUSTER_NAME` that will, as you can guess, hold the name of the cluster we created a short while ago. In the commands that follow, please replace the first occurrence of [...] with the name of the cluster and the second with your GitHub user:

```
$ export CLUSTER_NAME=[...]
$ export GH_USER=[...]
```

After 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:

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

```
$ cd environment- $\$CLUSTER\_NAME$ -dev
$ cat jx-requirements.yml
cluster:
  clusterName: ""
  environmentGitOwner: ""
  project: ""
  provider: gke
  zone: ""
gitops: true
environments:
- key: dev
- key: staging
- key: production
ingress:
  domain: ""
  externalDNS: false
  tls:
    email: ""
    enabled: false
    production: false
kaniko: true
secretStorage: local
storage:
  logs:
    enabled: false
    url: ""
  reports:
    enabled: false
    url: ""
```



```

repository:
  enabled: false
  url: ""
versionStream:
  ref: "master"
  url: https://github.com/jenkins-x/jenkins-x-versions.git
webhook: prow

```

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

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).
- 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 things may have changed since the writing of this chapter, and your provider is indeed supported now.
- Set `cluster.zone` to whichever zone your cluster is running in. If you're running a regional cluster (as you should), then the value should be the region, not the zone. If, for example, you used my Gist to create a GKE cluster, the value should be `us-east1-b`. Similarly, the one for EKS is `us-east-1`.

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

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 `local`. The whole platform will be defined in this repository, except for secrets (such as 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 secrets 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. If you have a vault setup, you can set the value of `secretStorage` to `vault`. Otherwise, you can leave the default value `'local'`.

Below the `secretStorage` value is the whole section that defines storage for logs, reports, and repositories. 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, and so on.

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`.

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 control of the Jenkins X versioning and that you're ready to maintain it.

As you already know, Prow only supports GitHub. If that's not your Git provider, Prow is a no-go. As an alternative, we could set it up in 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 has 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 (such as 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).

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

```
$ export IAM_USER=[...] # such as jx-boot
echo "vault:
  aws:
    autoCreate: true
    iamUserName: \"\$IAM_USER\" \" \" \
    | tee -a jx-requirements.yml
```

Please only execute the following commands if you are using EKS. The jx-requirements.yml file contains zone entry, and for AWS we need a region. That command will replace one with the other:

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

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

```
$ cat jx-requirements.yml
cluster:
  clusterName: "jx-boot"
  environmentGitOwner: "vfarctic"
  project: "devops-26"
  provider: gke
  zone: "us-east1"
gitops: true
environments:
- key: dev
- key: staging
- key: production
ingress:
  domain: ""
  externalDNS: false
  tls:
    email: ""
    enabled: false
    production: false
kaniko: true
secretStorage: vault
storage:
  logs:
```

```

    enabled: true
    url: ""
  reports:
    enabled: true
    url: ""
  repository:
    enabled: true
    url: ""
  versionStream:
    ref: "master"
    url: https://github.com/jenkins-x/jenkins-x-versions.git
  webhook: prow

```

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 the 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 Boot create the resources that are missing and thus convert the unknowns into known.

Let's install Jenkins X:

```
$ jx boot
```

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

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

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 done. All we have to do is repeat the process but, this time, with the latest version of jx:

```
$ jx boot
```

The process started again. We'll skip commenting on the first few questions from jx boot and 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. You can use your GitHub username and email for the first two questions. As for the token,<sup>27</sup> you'll need to create a new one in GitHub and *grant full repo access*. Finally, the next 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 the case of GKE, that would be GCR, for EKS that's ECR. In any case, by not configuring an external Docker Registry, the Boot will use whatever makes the most sense for a given provider:

```
? Jenkins X Admin Username admin
? Jenkins X Admin Password [?] for help] *****
? The Git user that will perform git operations inside a pipeline. It should
  be a user within the Git organisation/own? Pipeline bot Git username
  vfarci
? Pipeline bot Git email address vfarci@gmail.com
? A token for the Git user that will perform git operations inside a
  pipeline. This includes environment repository creation, and so this
  token should have full repository permissions. To create a token go to
  https://github.com/settings/tokens/new?scopes=repo,read:user,read:org,us
```

---

<sup>27</sup> <https://docs.github.com/en/github/authenticating-to-github/creating-a-personal-access-token>

```

er:email,write:repo_hook,delete_repo then enter a name, click Generate
token, and copy and paste the token into this prompt.
? Pipeline bot Git token *****
Generated token bb65edc3f137e598c55a17f90bac549b80fefbcaf, to use it press
enter.
This is the only time you will be shown it so remember to save it
? HMAC token, used to validate incoming webhooks. Press enter to use the
generated token [? for help]
? Do you want to configure non default Docker Registry? No

```

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. We have the whole definition of the platform with complete configuration (except for secrets) stored in a Git repository:

```

verifying the Jenkins X installation in namespace jx
verifying pods
Checking pod statuses
POD                                STATUS
jenkins-x-chartmuseum-774f8b95b-bdxfh    Running
jenkins-x-controllerbuild-66cbf7b74-twkbp  Running
jenkins-x-controllerrole-7d76b8f449-5f5xx  Running
jenkins-x-gcactivities-1594872000-w6gns    Succeeded
jenkins-x-gcpods-1594872000-m7kgq         Succeeded
jenkins-x-heapster-679ff46bf4-94w5f       Running
jenkins-x-nexus-555999cf9c-s8hnn          Running
lighthouse-foghorn-599b6c9c87-bvpct       Running
lighthouse-gc-jobs-1594872000-wllsp       Succeeded
lighthouse-keeper-7c47467555-c87bz        Running
lighthouse-webhooks-679cc6bbbd-fxw7z      Running
lighthouse-webhooks-679cc6bbbd-zl4bw      Running
tekton-pipelines-controller-5c4d79bb75-75hvj Running
Verifying the git config
Verifying username billyy at git server github at https://github.com
Found 2 organisations in git server https://github.com: IntuitDeveloper,
    intuit
Validated pipeline user billyy on git server https://github.com
Git tokens seem to be setup correctly
Installation is currently looking: GOOD
Using namespace 'jx' from context named 'gke_hazel-charter-283301_us-east1-
    b_cluster-1' on server 'https://34.73.66.41'.

```

### **B.3 Install Flux**

Flux consists of a CLI client and daemon that runs inside of the managed Kubernetes cluster. This document explains how to install the Flux CLI only. The daemon installation requires you to specify the Git repository with access credentials and covered in chapter 11.

### **B.3.1 Install CLI client**

The Flux distribution includes the CLI client named `fluxctl`. The `fluxctl` automates Flux daemon installation and allows to get the information about Kubernetes resources controlled by the Flux daemon.

Use one of following commands to install the `fluxctl` in Mac, Linux, and Windows:

#### **Mac OS**

```
brew install fluxctl
```

#### **Linux**

```
sudo snap install fluxctl
```

#### **Windows**

```
choco install fluxctl
```

Find more information about `fluxctl` installation details in the official installation instructions: <https://docs.fluxcd.io/en/latest/references/fluxctl/>.