# 10

# OpenShift GitOps – Argo CD

In the previous chapter, we learned how to create and run a pipeline using Tekton to build and deploy an application. While Tekton is great for building and performing other actions that are usually related to **continuous integration (CI)**, **GitOps** is becoming the norm for **continuous deployment (CD)** regarding Kubernetes-native applications. In this chapter, we will dive into GitOps and talk about one of the best tools for CD: **Argo CD**.

In this chapter, we will cover the following topics:

- What is GitOps?
- What is Argo CD?
- Application delivery model
- Installing OpenShift GitOps
- Configuring Argo CD against multiple clusters
- Argo CD definitions and challenges
- Argo CD main objects
- Deploying an application using GitOps
- Deploying to multiple clusters

Let's dive in!

*NOTE*

*The source code used in this chapter is available at* **https://github.com/PacktPublishing/OpenShift-Multi-Cluster-Management-Handbook/tree/main/chapter10**.

## What is GitOps?

The term GitOps was first described by *Lexis Richardson*, CEO of *Weaveworks*, in 2017. At that time, he presented the four principles of GitOps, which are as follows:

- **The entire system is described declaratively**: This means that any configuration of your application and infrastructure needs to be treated as code, but not as a set of instructions, as you would with scripts or automation code. Instead, you must use a set of facts that describes the desired state of your system. These declaration files are versioned in Git, which is your single source of truth. The great benefit of this principle is that you can easily deploy or roll back your applications and,

more importantly, restore your environment quickly if a disaster occurs.

- **The canonical desired system state is versioned in Git**: Git is your source of truth. It needs to be the single place that triggers all the changes in your systems. Ideally, nothing should be done directly on the systems, but through configuration changes on Git that will be applied automatically using a tool such as Argo CD.
- **Approved changes are automatically applied to the system**: Since you have the desired state of your system stored in Git, any changes can be automatically applied to the system as they are pushed to the repository.
- **Software agents ensure correctness and alert you about divergence**: It is crucial to have tools in place that will ensure that your system is in the desired state, as described in Git. If any drift is detected, the tool needs to be able to self-heal the application and get it back to its desired state.

GitOps became the norm for Kubernetes and cloud-native applications due to the following benefits:

- **Standard process and tools**: Git workflows allow teams to work collaboratively and in a reproducible manner, avoiding issues regarding human-repetitive tasks.
- **Robust and secure process**: By working with **pull requests** (**PRs**) in Git, all the changes need to be reviewed and approved. You can also trace all changes in Git and revert them if needed.
- **Auditable changes**: All changes are tracked and easily auditable in Git history.
- **Consistency**: You can deploy the same application in multiple different clusters consistently:

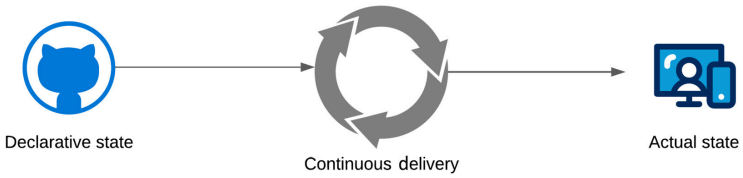


Figure 10.1 – GitOps workflow

Now that you have a better understanding of what GitOps is, let's start learning how to put GitOps workflows into practice.

## What is Argo CD?

In theory, it is possible to adopt GitOps without the need to use any specific tool. You could implement scripts and automation to deploy and manage your applications using declarative files from Git that describe your systems. However, that would be costly and time-consuming. The good news is that there are some great open source options for Kubernetes that are stable and work well. At the time of writing, the main tools for Kubernetes are **Argo CD** and **Flux CD**; both are great tools but in this book, we will explore Argo CD, which comes at *no additional cost with a Red Hat OpenShift subscription*.

In a nutshell, Argo CD is a tool that is capable of *reading* a set of *Kubernetes manifests, Helm charts, or Jsonnet files* stored in a Git repository and *applying* them to a Kubernetes namespace. Argo CD is not only able to apply manifests, though – it can also automate self-healing, object pruning, and other great capabilities, as we will explore in this chapter.

## Application delivery model

At this point, you may be wondering how OpenShift Pipelines (**Tekton**) and GitOps (**Argo CD**) are related. Tekton and Argo CD are complementary tools that are perfect together. While Tekton is a perfect fit for *CI* pipelines that run unit tests and build and generate container images, Argo CD is more appropriate for *continuous delivery* practice. The following diagram summarizes what a CI/CD pipeline with Tekton and Argo CD looks like:

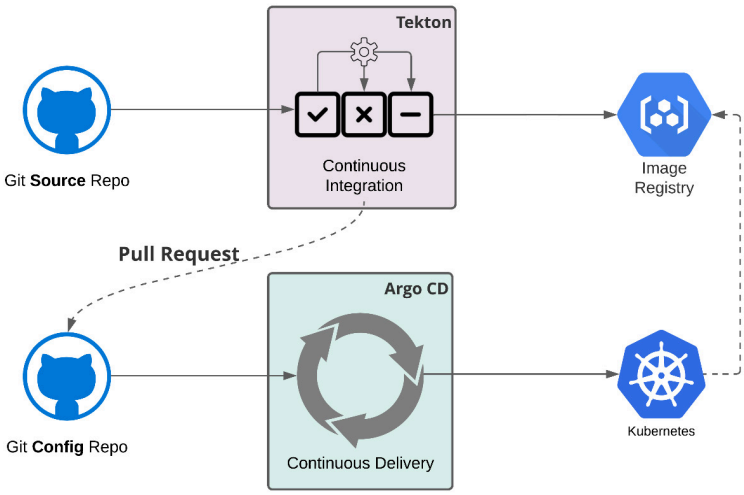


Figure 10.2 – Application delivery model using Tekton and Argo CD

CD with GitOps means that the actual state of the application should be monitored and that any changes need to be reverted to the application's desired state, as described in the Git repository:

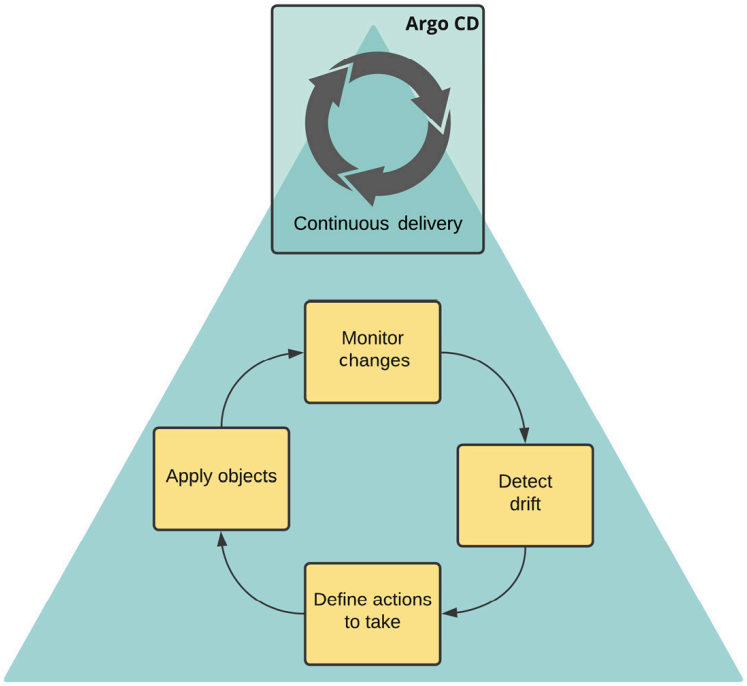


Figure 10.3 – Continuous delivery with GitOps

In this chapter, we will use our example from the previous chapter and use Argo CD to deploy the application and practice this application delivery model.

# Installing OpenShift GitOps

The installation process is simple and is similar to what we followed in the previous chapter regarding OpenShift Pipelines.

## Prerequisites

To install OpenShift GitOps, you will need an OpenShift cluster with cluster-admin permissions.

## Installation

Follow these steps:

1. Access the **OpenShift web console** using the administrator's perspective.
2. Navigate to **Operators** | **OperatorHub**:

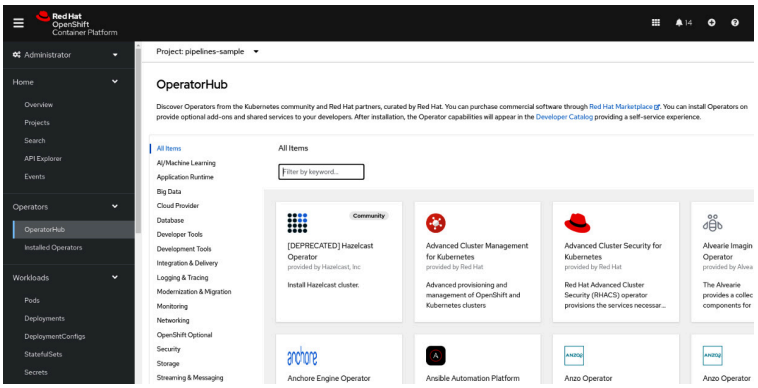


Figure 10.4 – OperatorHub

3. Search for **`OpenShift GitOps`** using the *Filter by keyword* box:

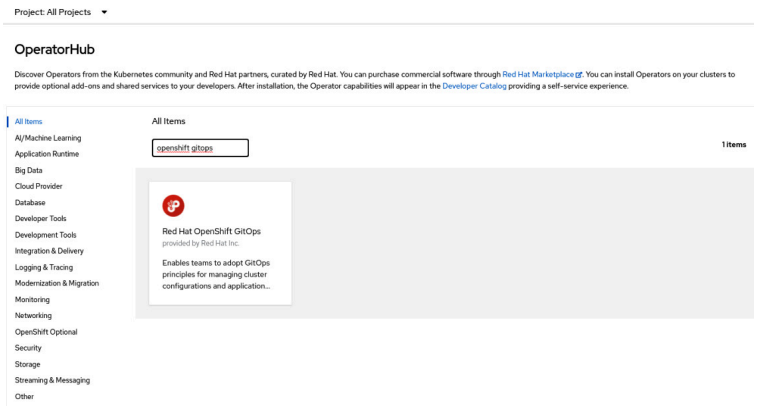


Figure 10.5 – Red Hat OpenShift GitOps on OperatorHub

4. Click on the **Red Hat OpenShift GitOps** tile and then the **Install** button to go to the **Install** screen:

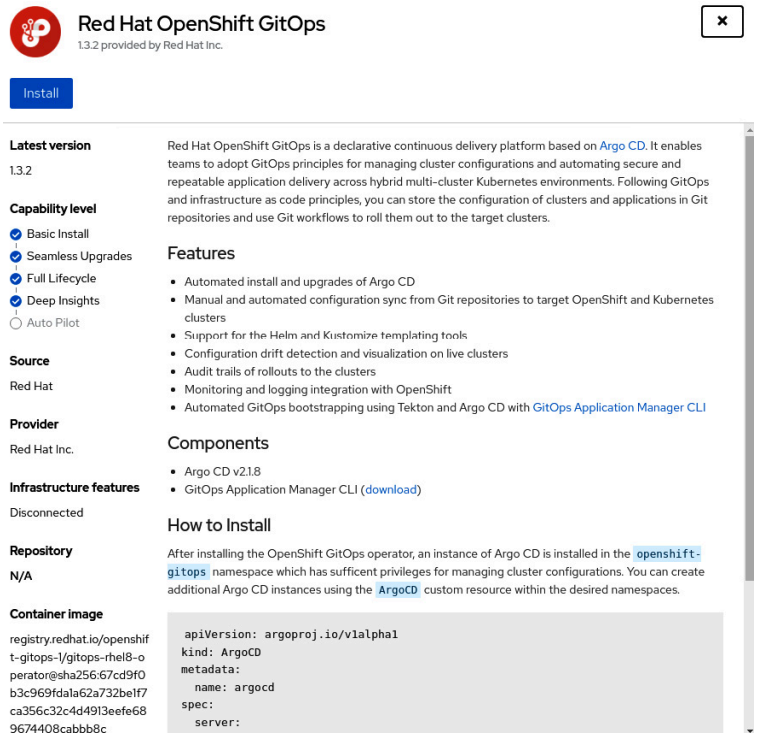


Figure 10.6 – Installing OpenShift GitOps

5. Now, select **All namespaces on the cluster (default)** for **Installation mode**. As such, the operator will be installed in the **`openshift-operators`** namespace and permits the operator to install OpenShift GitOps instances in any target namespace.
6. Select **Automatic** or **Manual** for the upgrade's **Approval Strategy**. If you go for **Automatic**, upgrades will be performed automatically by the **Operator Lifecycle Manager** (**OLM**) as soon as they are released by Red Hat, while for **Manual**, you need to approve it before it can be applied.
7. Select an **Update channel**. The **stable** channel is recommended as it contains the latest stable and *supported* version of the operator.
8. Click the **Install** button:

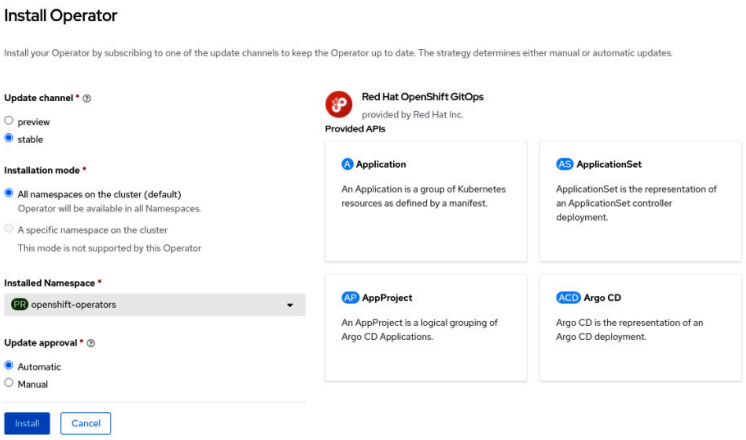


Figure 10.7 – Installing the operator

9. Wait up to 5 minutes until you see the following message:

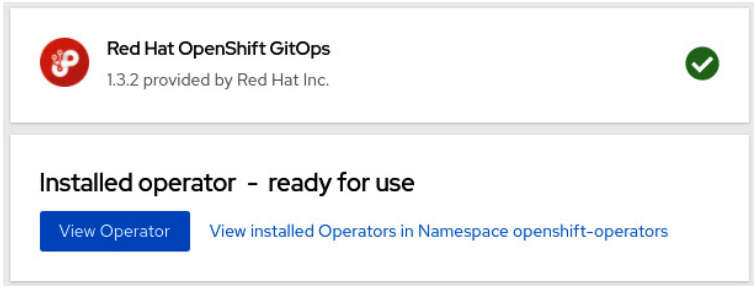


Figure 10.8 – Operator installed

OpenShift GitOps (Argo CD) also has a CLI, which helps execute common tasks, such as updating the admin's password, registering external clusters, and much more. Let's learn how to install the **`argocd`** CLI.

## Installing the argocd CLI

The **`argocd`** CLI makes it easier to work with Argo CD. Through it, you can manage Argo CD projects, applications, cluster credentials, and more.

To install the **`argocd`** CLI, follow these steps:

1. Download the latest Argo CD binary file from **https://github.com/argoproj/argo-cd/releases/latest**.
2. If you are using Linux, download the CLI and add it to your path:

```
$ sudo curl -sSL -o /usr/local/bin/argocd https://github.com/argoproj/argo-cd/releases/
```

```
$ sudo chmod +x /usr/local/bin/argocd
```

3. If everything went well, you will see the following output by running the **`argocd version`** command. Ignore the error message you see in the last line; it is an expected message as we haven't logged in to any OpenShift cluster yet:

```
$ argocd version
```

```
argocd: v2.2.1+122ecef
```

```
  BuildDate: 2021-12-17T01:31:40Z
```

```
  GitCommit: 122ecefc3abfe8b691a08d9f3cecf9a170cc8c37
```

```
  GitTreeState: clean
```

```
  GoVersion: go1.16.11
```

```
  Compiler: gc
```

```
  Platform: linux/amd64
```

```
FATA[0000] Argo CD server address unspecified
```

Now, let's learn how to configure Argo CD to deploy applications against multiple clusters.

## Configuring Argo CD against multiple clusters

If you are planning to use Argo CD to deploy applications to *external clusters*, you need to add the new cluster's credentials using the **`argocd`** CLI. You can skip this step if you want to deploy applications in the same clus-

ter where Argo CD is installed (the **kubernetes.default.svc** file already exists and should be used in this case).

To register new clusters, perform the following steps using the **argocd** CLI you installed previously:

1. Log into the new cluster we want to register:

```
$ oc login -u <user> https://<api-newcluster>:6443
```

2. Now, log into the cluster where Argo CD is installed using **oc login**:

```
$ oc login -u <user> https://<api-argocluster>:6443
```

3. At this point, you should have both clusters in your **kubeconfig** file:

```
$ oc config get-contexts
```

```
CURRENT NAME         CLUSTER    AUTHINFO NAMESPACE
```

```
        newcluster   newcluster admin
```

```
*       argocluster  argocluster
```

4. Set a different context for the new cluster:

```
$ oc config set-context prd-cluster --cluster=newcluster --user=admin
```

```
Context "prd-cluster" created.
```

5. Get the Argo CD public URL from the **openshift-gitops** namespace:

```
$ oc get route openshift-gitops-server -n openshift-gitops -o jsonpath='{.spec.host}'
```

6. Get the administrator password:

```
$ oc extract secret/openshift-gitops-cluster -n openshift-gitops --to=-
```

7. Log in using **argocd**:

```
$ argocd login --insecure openshift-gitops-server-openshift-gitops.apps.example.com
```

```
Username: admin
```

```
Password:
```

```
'admin:login' logged in successfully
```

```
Context ' openshift-gitops-server-openshift-gitops.apps.example.com' updated
```

8. Now, add the new cluster to Argo CD:

```
argocd cluster add prd-cluster -y
```

```
INFO[0000] ServiceAccount "argocd-manager" created in namespace "kube-system"
```

```
INFO[0000] ClusterRole "argocd-manager-role" created
```

```
INFO[0001] ClusterRoleBinding "argocd-manager-role-binding" created
```

```
Cluster 'h https://<api-newcluster>:6443' added
```

With that, you are ready to deploy applications either into your local or remote cluster using Argo CD! But before we dive into application deployment, let's check out some of the important aspects related to Argo CD.

# Argo CD definitions and challenges

Before we walk through the application deployment process, we need to discuss some important challenges related to GitOps, decisions, and standards.

## GitHub repository structure

The first important question that always comes up with GitOps is about the GitHub repository's structure. Should I only use one repository for source code and the Kubernetes manifests? How should I deal with differ-

ent configuration files for different environments, such as development, QA, and production?

Well, there are no right or wrong answers to these questions, as each option has its pros and cons. You need to find out which approach works best for you. My advice here is: try it! There is nothing better than practical experience, so use each model and find out which one fits best for your applications and teams. In the following sections, we'll look at some of the most popular repository structures out there for GitOps-oriented applications.

### Mono-repository

In this structure, you will have one repository for all your Kubernetes manifests and infrastructure-related files. Although there is not a single standard for this structure, you will probably have a repository similar to the following:

```
├── config #[1]
├── environments #[2]
│   ├── dev #[3]
│   │   ├── apps #[4]
│   │   │   └── app-1
│   │   └── env #[5]
│   └── qa #[6]
│   │   ├── apps
│   │   │   └── app-1
│   │   └── env
(...)
```

Let's look at this code in more detail:

- **#[1]**: This folder contains the CI/CD pipelines, Argo CD, and other related configuration files that are common for any environment
- **#[2]**: This folder contains the manifests that are specific to each environment, such as development, QA, and production
- **#[3]**: These are the manifest files that are specific to the development environment
- **#[4]**: Here, you have the Kubernetes manifests to deploy the applications that are tracked and released in this repository
- **#[5]**: These are the cluster or infrastructure-related manifests for the development environment, such as **`RoleBinding`** permissions, **`Namespace`**, and so on
- **#[6]**: These are the manifest files that are specific to the QA environment

The main benefit of this approach is its **simplicity**: in this approach, you only need to manage one repository for one or more applications, which makes it easier to manage branches, tags, **PRs**, and anything related to the application's manifests repository. However, the major con of this strategy is that all the contributors can read and make changes to the production manifests. That said, it might be *hard to detect unintentional changes to production*, especially with large PRs.

This leads us to the next approach, in which you have a different repository per environment.

### Repository per environment

With this strategy, you will have multiple repositories, one for each environment. In other words, you will have one repository for *development* manifests, another one for *QA*, and so on. In this strategy, you will likely use PRs to promote changes between each environment and have a granular review process, which leads to a less error-prone process. In this strategy, you can also manage Git permissions according to each environment:

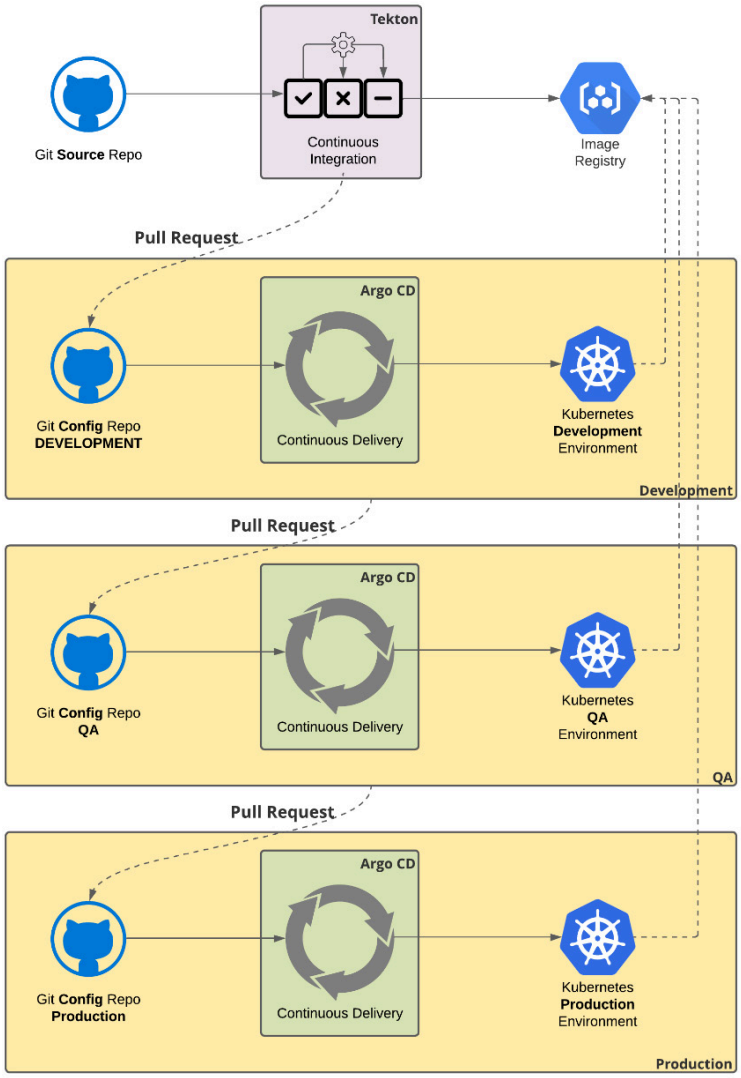


Figure 10.9 – One repository per environment

In this chapter, we will be using a mono-repository strategy and use Git push requests and PRs with multiple branches to mitigate the risk of unintentional changes.

Next, we will discuss another important aspect related to GitOps on Kubernetes: templating YAML files and avoiding duplication.

## Templating Kubernetes manifests

Whatever repository structure you decide to go for, one thing is certain: you will need to have separate files and folders for each environment you manage. So, how can you avoid duplicating YAML manifest files everywhere and turning your GitOps process into a nightmare?

Currently, the most popular options to do this are as follows:

- **Helm**: Rely on Helm Charts and Helm Templates to package and deliver Kubernetes applications. Through Helm Templates, you can combine values with templates and generate valid Kubernetes manifest files as a result.
- **Kustomize**: With Kustomize, you can reuse existing manifest files using a patch strategy. It uses a hierarchical structure so that you can flexibly reuse shared configurations and create layers of configurations with only environment-specific parameters that will be overloaded with base parameters.

While Helm is a great package tool, we are going to focus on Kustomize in this chapter due to the following reasons:

- Kustomize runs natively on Kubernetes and the OpenShift CLI (**`kubectl`**/**`oc`**)
- It is declarative, which is an important factor for GitOps, as we mentioned previously
- You can use a remote base in a repository as the starter set of the manifest and have the overlays stored in different repositories

Let's take a closer look at Kustomize.

### Kustomize

Kustomize is composed of hierarchical layers of manifest files:

- **`Base`**: This is a directory that contains the resources that are always reused as the base manifest files. These describe the application and objects declaratively.
- **`Overlays`**: This is a directory that only contains the configurations that are specific for each overlay. For instance, it is common to have an overlay for the development environment, another for QA, and so on. The configurations that reside in the **`overlay`** directories replace the values that are in the **`base`** directory.

You can have multiple layers of bases and overlays – as many as you want. However, to maintain the legibility and maintainability of your application manifest files, it is not recommended to use several layers of manifest files. The following diagram shows an example of a base and two overlays that might be used with Kustomize:

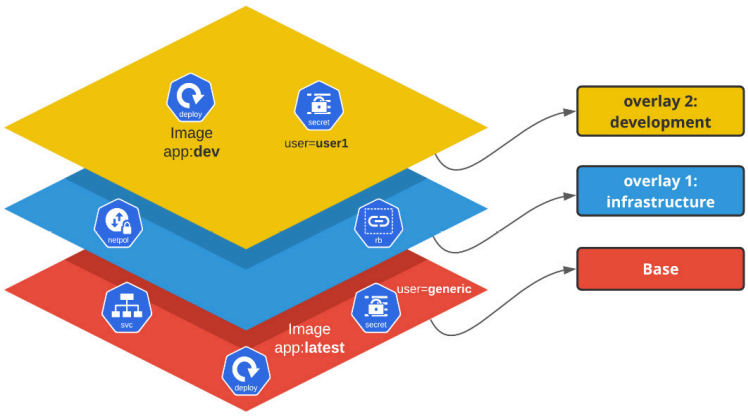


Figure 10.10 – Kustomize layers

The following is a typical folder structure you'll see when you're using Kustomize:

```
├── base
│   ├── deployment.yaml
│   ├── kustomization.yaml
│   ├── route.yaml
│   └── service.yaml
└── overlays
    ├── dev
    │   ├── deployment-patch.yaml
    │   ├── kustomization.yaml
    │   └── namespace.yaml
    ├── prod
    │   ├── deployment-patch.yaml
    │   ├── kustomization.yaml
    │   └── namespace.yaml
    └── qa
        ├── deployment-patch.yaml
        ├── kustomization.yaml
        └── namespace.yaml
```

We will practice using Kustomize a bit more in this chapter when we deploy our example.

## Managing secrets

Last, but not least, the real challenge with Kubernetes is managing sensitive data using secrets. While secrets are somewhat safe, depending on how the permissions are set among users, they are not encrypted. This is a real problem when we think about storing those secrets in a GitHub repository. So, how can we handle secrets securely?

There are two ways to handle secrets:

- Use an external vault to store the secrets securely outside Git and the cluster
- Encrypt the secret before saving it in Git using a sealed secret tool such as Bitnami Sealed Secrets

*NOTES*

*1. Secrets are presented in Base64 encoding. The following command, for instance, decrypts a secret named* **`db-root-password`** *that contains a password field:*

```
oc get secret db-root-password -o jsonpath="{.data.password}" | base64 -d
```

*2. Bitnami Sealed Secrets allows you to encrypt your secret into a* **`SealedSecret`** *object and store it securely, even in a public GitHub repository, since it is encrypted using a public/private certificate. To learn more, check out the link in the Further reading section.*

With that, we have discussed the main points that you need to think about regarding GitOps. They are important topics we decided to bring to you before our practical example but don't get too worried about that yet – you will find out what works best for you and your team by practicing and learning from it. In the next section, we will introduce some of the main objects you will work with in Argo CD.

# Argo CD main objects

In this section, we will look at some of the main Argo CD objects you need to know about. Argo CD is quite simple and most of what you will do can be summarized in two objects: **AppProject** and **Application**.

In this section, we will not mention all the different objects Argo CD has since that is not the main focus of this book. Check out the *Further reading* section to learn more.

## AppProject

Projects allow you to group applications and structure them according to any group logic you need. Using projects, you can do the following:

- Limit the Git *source repositories* that can be used to deploy applications
- Restrict the *clusters and namespaces destination* that the applications can be deployed to
- Limit the type of objects that can be deployed (for example, Deployments, Secrets, DaemonSets, and so on)
- Set roles to limit the permissions that are allowed by groups and/or JWTs

When Argo CD is installed, it comes with a **`default`** project. If you don't specify a project in your Argo CD application, the **`default`** option will be used. Creating additional projects is optional as you could use Argo CD's **`default`** project instead. However, it is recommended to create additional projects to help you organize your Argo CD applications.

*NOTE*

*There is no relationship between Argo CD's default project and OpenShift's default namespace. Although they have the same name, they are unrelated.*

A typical **`AppProject`** specification looks as follows:

```
apiVersion: argoproj.io/v1alpha1
kind: AppProject
metadata:
  name: clouds-api #[1]
  namespace: openshift-gitops
spec:
  sourceRepos: #[2]
    - '*'
  destinations: #[3]
    - namespace: '*'
      server: '*'
  clusterResourceWhitelist: #[4]
    - group: '*'
      kind: '*'
```

Let's look at this code in more detail:

- **[1]**: The name of the project.
- **[2]**: The Git source repositories that are allowed. In this case, any source repository is allowed
- **[3]**: The destination clusters and namespaces that are allowed. In this case, any combination of clusters and namespaces is allowed
- **[4]**: The objects that can be deployed (for example, Deployments, Secrets, DaemonSets, and so on). In this case, there is no limitation

Properly adjust the code pointed out to achieve manifest file.

## Applications

Applications represent application instances that have been deployed and managed by Argo CD. The specification of an application is composed of **`source`** and **`destination`**. **`source`** is where the Kubernetes manifests (Git repository) that specify the desired state of the application reside, while **`destination`** specifies the cluster and namespace where the application will be deployed. Besides that, you can also specify the synchronization policies you want Argo CD to apply.

The following is an example of an **`Application`** specification:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: clouds-app-dev #[1]
  namespace: openshift-gitops #[2]
spec:
  project: clouds-api #[3]
  source: #[4]
    repoURL: `https://github.com/PacktPublishing/OpenShift-Multi-Cluster-Management-Handbo
    path: chapter10/clouds-api-gitops/overlays/dev
```

```
    targetRevision: dev
  destination: #[5]
    server: 'https://kubernetes.default.svc'
    namespace: default
  syncPolicy: #[6]
    automated:
      selfHeal: true
```

Let's look at this code in more detail:

- **[1]**: Argo CD **`Application`** name.
- **[2]**: This is the namespace where Argo CD is installed. The default project for the OpenShift GitOps operator is **`openshift-gitops`**.
- **[3]**: This is the Argo CD project that you create using the **`AppProject`** object. Do not get it confused with the OpenShift project; they are unrelated.
- **[4]**: Git source repository information about where the Kubernetes manifests reside.
- **[5]**: The cluster and namespace where the application will be deployed.
- **[6]**: The synchronization policies that Argo CD will use. We will learn more about these policies in the next section.

*IMPORTANT NOTE*

*Argo CD's namespace (***`openshift-gitops`***) has special privileges within the cluster to perform all the necessary activities. Due to that, you must protect access to this namespace to avoid unwanted deployments or changes.*

### Syncing policies

You can configure Argo CD to automatically synchronize your application when there is any drift between the desired state specified in the manifests in Git and the actual application state. You have the following options with Argo CD:

- **Self-Healing**: When you set this option to **`true`**, Argo CD will automatically sync when it detects any differences between the manifests in Git and the actual state. By default, this flag is **`false`**.
- **Pruning**: As a precaution, automated sync will never delete a resource when it no longer exists in Git. Argo CD only prunes those resources with a manual sync. However, if you want to allow Argo CD to automatically prune objects that no longer exist in Git, you can set the **`prune`** flag to **`true`**.

### Syncing the order

For standard Kubernetes manifests, Argo CD already knows the correct order that needs to be applied to avoid precedence issues. For instance, consider an application that contains three manifests for namespace creation, deployment, and role bindings. In such a case, Argo CD will always apply the objects in the following order:

1. Namespace

2. Role bindings
3. Deployment

That said, this is the kind of thing you don't need to be worried about as Argo CD is smart enough to apply them in the correct order.

However, there are some other specific cases where you may need to specify objects' precedence. Let's say that you want to deploy an application composed of one StatefulSet to deploy a database and a deployment for an application that uses the database. In this case, you can use **resource hooks** to specify the correct order to apply the objects.

The following types of resource hooks can be used:

- **`PreSync`**: Objects marked with **`PreSync`** are executed before any other manifests.
- **`Sync`**: This runs after **`PreSync`** is complete. You can also use **`sync-wave`** to set the sync precedence of the objects in the **`Sync`** phase.
- **`PostSync`**: Runs after all the **`Sync`** objects have been applied and are in a **`Healthy`** state.
- **`SyncFail`**: The manifests with this annotation will only be executed when a sync operation fails.

The following is an example of a resource hook specification:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    argocd.argoproj.io/hook: Sync
    argocd.argoproj.io/sync-wave: "1"
(.. omitted ..)
```

There are different types of annotations that you can include in your manifests to perform more complex tasks. Check out the *Further reading* section to learn more.

With that, we have covered the most important concepts and theories behind GitOps and Argo CD. Now, without further ado, let's look at our example and practice what we have discussed so far!

# Deploying an application using GitOps

In this practical exercise, we will build and deploy our sample application in three different namespaces to simulate an application life cycle composed of development, QA, and production environments. The following diagram shows the delivery model we will use in this exercise to practice Argo CD deployments. Use it as much as you want as a starting point to build a comprehensive and complex ALM workflow that's suitable for your needs:

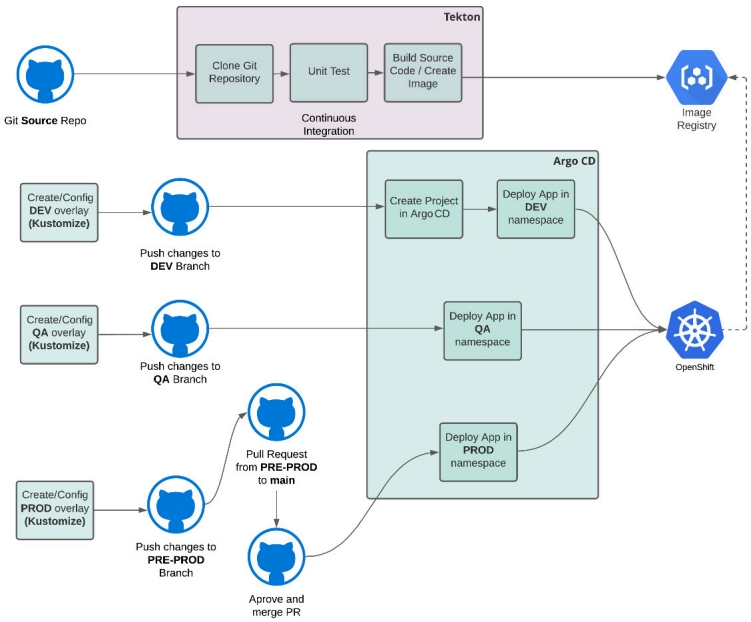


Figure 10.11 – Application delivery model using Tekton, Argo CD, and Git

Once again, we are going to use the content we have prepared in this book's GitHub repository. To do this, you must *fork this repository to your GitHub account*: **https://github.com/PacktPublishing/OpenShift-Multi-Cluster-Management-Handbook**. Once you have forked it, follow the instructions in this section to put this workflow into practice.

## Building a new image version

In this section, we will build a new container image version, 1.0, and push it to the OpenShift internal registry, as shown in the following diagram:

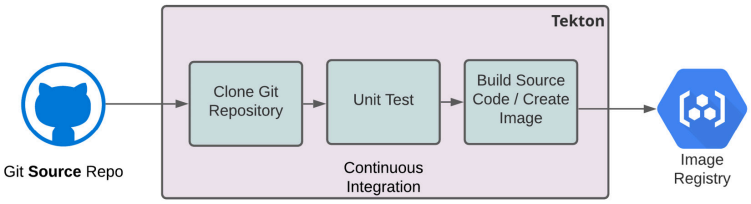


Figure 10.12 – Building a new image version

To do so, perform the following steps:

1. Clone the repository in your machine:

```
$ GITHUB_USER=<your_user>
```

```
$ git clone https://github.com/PacktPublishing/OpenShift-Multi-Cluster-Management-Handb
```

2. Run the following script and follow the instructions to change the references from the original repository (**PacktPublishing**) to your forked repository:

```
$ cd OpenShift-Multi-Cluster-Management-Handbook/chapter10
```

```
$ ./change-repo-urls.sh
```

```
# Go back to the root dir
```

```
$ cd ..
```

3. Create a new branch for development:

```
$ git checkout -b dev
```

4. Open the **`./sample-go-app/clouds-api/clouds.go`** file with your preferred text editor and change line 147 by adding **`version=1.0`** to it:

```
$ vim ./sample-go-app/clouds-api/clouds.go
```

```
func homePage(w http.ResponseWriter, r *http.Request) {
```

```
fmt.Fprintf(w, "Welcome to the HomePage! Version=1.0")
```

```
fmt.Println("Endpoint Hit: homePage")
```

```
}
```

5. Commit and push change to the **`dev`** branch:

```
$ git add ./sample-go-app/clouds-api/clouds.go
```

```
$ git commit -m 'Version 1.0 changes'
```

```
$ git push -u origin dev
```

6. Run the following command to deploy the required prerequisites and the pipeline that builds image version 1.0, which we will deploy in the development namespace shortly. Make sure that you are already logged into the OpenShift cluster (by using the **`oc login`** command):

```
$ oc apply -k ./chapter10/config/cicd
```

7. Now, run the pipeline and check the logs:

```
$ oc apply -f ./chapter10/config/cicd/pipelinerun/build-v1.yaml -n cicd
```

```
$ tkn pipelinerun logs build-v1-pipelinerun -f -n cicd
```

```
[fetch-repository : clone] + '[' false = true ']'
```

```
[fetch-repository : clone] + '[' false = true ']'
```

```
[fetch-repository : clone] + CHECKOUT_DIR=/workspace/output/
```

```
[fetch-repository : clone] + '[' true = true ']'
```

```
(.. omitted ..)
```

```
[build-image : push] Writing manifest to image destination
```

```
[build-image : push] Storing signatures
```

```
 [build-image : digest-to-results] + cat /workspace/source/image-digest
```

```
[build-image : digest-to-results] + tee /tekton/results/IMAGE_DIGEST
```

```
[build-image : digest-to-results] sha256:5cc65974414ff904f28f92a0deda96b08f4ec5a98a09c5
```

With that, you have built the **clouds-api:v1.0** container image and pushed it to OpenShift's internal registry. Now, let's deploy this image using **Kustomize** and **Argo CD**.

## Deploying in development

In this section, we are going to use Kustomize to overwrite the image tag of the deployment YAML file so that it uses **v1.0**, which we built in the previous section. We will also create a new namespace for the development branch named **clouds-api-dev**.

The following diagram shows the steps we will perform:

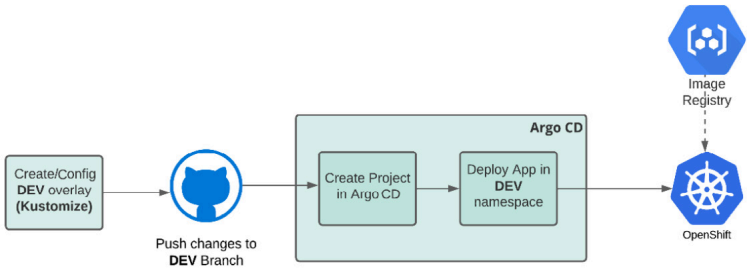


Figure 10.13 – Deploying in development

Perform the following steps:

1. Change the image version of our development **kustomization.yaml** file. To do so, change line 18 from **changeme** to **v1.0**:

```
$ vim ./chapter10/clouds-api-gitops/overlays/dev/kustomization.yaml
```

```
apiVersion: kustomize.config.k8s.io/v1beta1
```

```
kind: Kustomization
```

```
commonLabels:
```

```
  environment: dev
```

```
namespace: clouds-api-dev
```

```
bases:
```

```
  - ../../base

resources:

  - namespace.yaml

images:

- name: quay.io/gfontana/clouds-api

  newName: image-registry.openshift-image-registry.svc:5000/cicd/clouds-api

  newTag: v1.0 # Change this line
```

2. Alternatively, you may use the **sed** command to replace this line:

```
sed -i 's/changeme/v1.0/' ./chapter10/clouds-api-gitops/overlays/dev/kustomization.yaml
```

3. Now, push this change to the **dev** branch:

```
$ git add chapter10/clouds-api-gitops/overlays/dev/kustomization.yaml

$ git commit -m 'updating kustomization file for v1.0'

$ git push -u origin dev
```

4. Now, let's create a new Argo CD project:

```
$ oc apply -f ./chapter10/config/argocd/argocd-project.yaml
```

5. Create a new Argo CD application that will deploy the application in the development namespace:

```
$ oc apply -f ./chapter10/config/argocd/argocd-app-dev.yaml
```

6. Get Argo CD's URL and admin passwords using the following commands:

```
# Get the Argo CD url:

$ echo "$(oc  get route openshift-gitops-server -n openshift-gitops --template='https:/

# Get the Admin password

$ oc extract secret/openshift-gitops-cluster -n openshift-gitops --to=-
```

7. Access the Argo CD UI using the URL and admin user provided previously. You should see a new application there named **`clouds-app-dev`**:

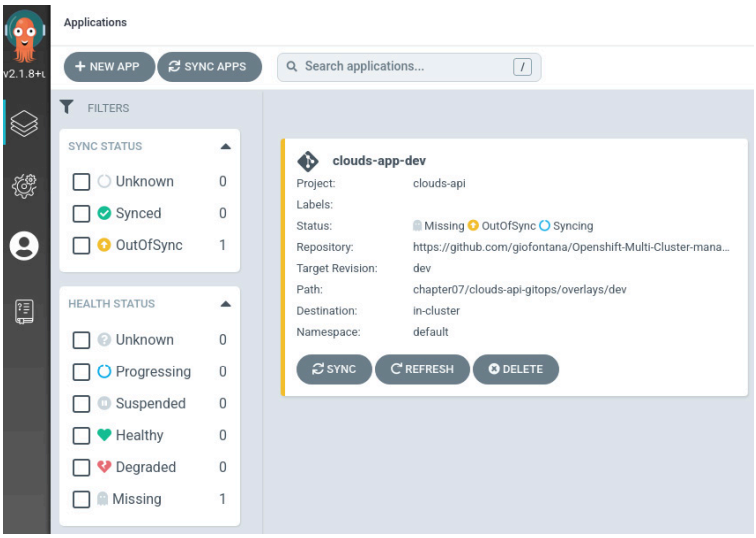


Figure 10.14 – The Argo CD application to deploy in development

8. Click **clouds-app-dev** to learn more about the application:

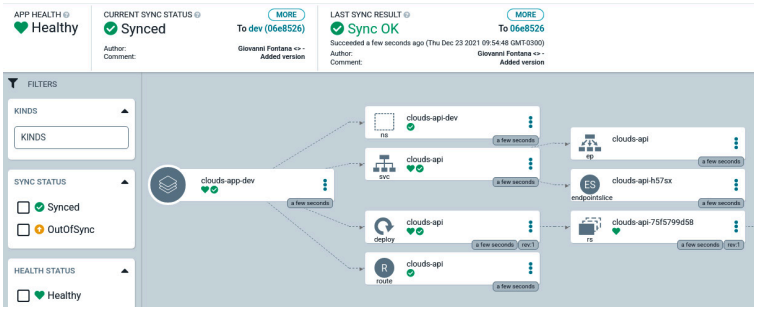


Figure 10.15 – The Argo CD application

9. Run the **`curl`** command to check that version 1.0 of the application is running and has been successfully deployed by Argo CD:

```
$ curl $(oc get route clouds-api -n clouds-api-dev --template='http://{{.spec.host}}')
```

You should see the following response:

```
Welcome to the HomePage! Version=1.0
```

With that, our sample application is running in the development namespace. Now, let's learn how to promote this application to the next stage: QA.

## Promoting to QA

We have application version 1.0 running in development. Now, let's use Kustomize and Argo CD once more to deploy it in a new namespace that's dedicated to QA, as shown in the following diagram:

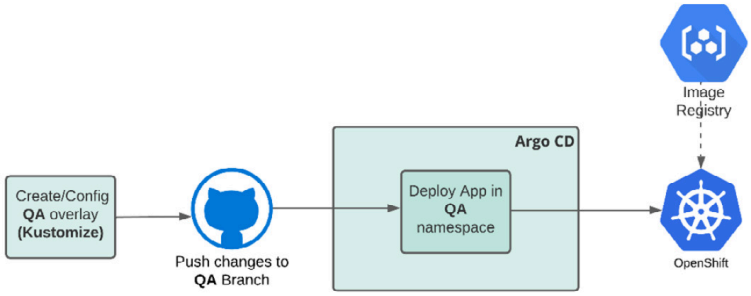


Figure 10.16 – Promoting to QA

Follow these steps:

1. Create a new branch for QA:

```
$ git checkout -b qa
```

2. Create an overlay for QA by copying the **dev** overlay:

```
$ cp -r ./chapter10/clouds-api-gitops/overlays/dev/ ./chapter10/clouds-api-gitops/overl
```

3. Replace the references to **dev** with **qa**:

```
$ sed -i 's/dev/qa/' ./chapter10/clouds-api-gitops/overlays/qa/namespace.yaml ./chapter
```

4. Push the changes to Git:

```
$ git add ./chapter10/clouds-api-gitops/overlays/qa
```

```
$ git commit -m 'Promoting v1.0 to QA'
```

```
$ git push -u origin qa
```

5. Deploy the manifest file to promote the environment using Argo CD:

```
$ oc apply -f ./chapter10/config/argocd/argocd-app-qa.yaml
```

6. Access the Argo CD UI again. At this point, you should have two applications on Argo CD:

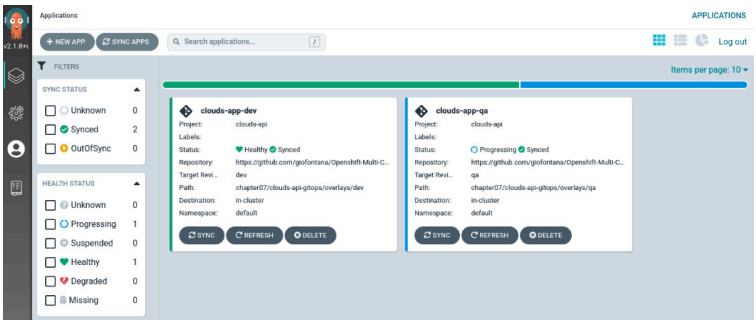


Figure 10.17 – Argo CD applications

7. Let's access the application that is running in the QA namespace:

```
$ curl $(oc get route clouds-api -n clouds-api-qa --template='http://{{.spec.host}}')
```

You should see the same response that you saw previously:

```
Welcome to the HomePage! Version=1.0
```

With that, we have promoted our application to QA! Now, let's learn how to move it to the last stage, which is the production environment.

## Promoting to production

For production, we are going to use a different approach – we are going to use PRs instead of simple Git pushes. We will use a temporary branch named **`pre-prod`** to commit the overlay manifests that will be used for production, as shown in the following diagram:

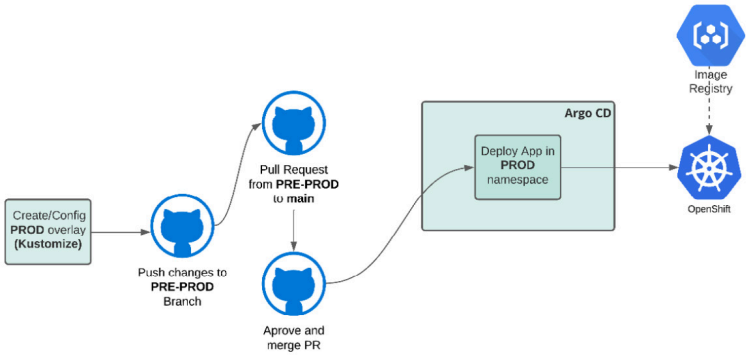


Figure 10.18 – Promoting to production

Follow these steps to promote version 1.0 of our application to production:

1. Create a new branch to prepare for production:

```
$ git checkout -b pre-prod
```

2. Create an overlay for production, similar to what you did with QA:

```
$ cp -r chapter10/clouds-api-gitops/overlays/dev/ chapter10/clouds-api-gitops/overlays/
```

```
$ sed -i 's/dev/prod/' ./chapter10/clouds-api-gitops/overlays/prod/namespace.yaml ./cha
```

3. Push the changes to the **`pre-prod`** branch:

```
$ git add ./chapter10/clouds-api-gitops/overlays/prod
```

```
$ git commit -m 'Promoting v1.0 to Prod'
```

```
$ git push -u origin pre-prod
```

4. Now, create a PR on GitHub and merge it with the main branch. Access the **Pull requests** tab of your GitHub repository and click the **New pull request** button:

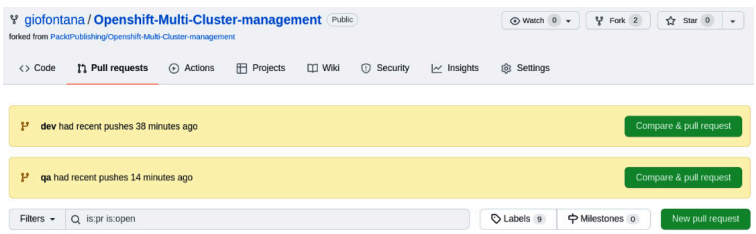


Figure 10.19 – Creating a PR

5. Since you are working in the forked repository, GitHub suggests that you create a PR for the source repository (in this case, from **`PacktPublishing`**). We want to create a PR that goes from our **`pre-prod`** branch to the **`main`** branch, both in our forked repository. So, change the base repository to our forked repository:

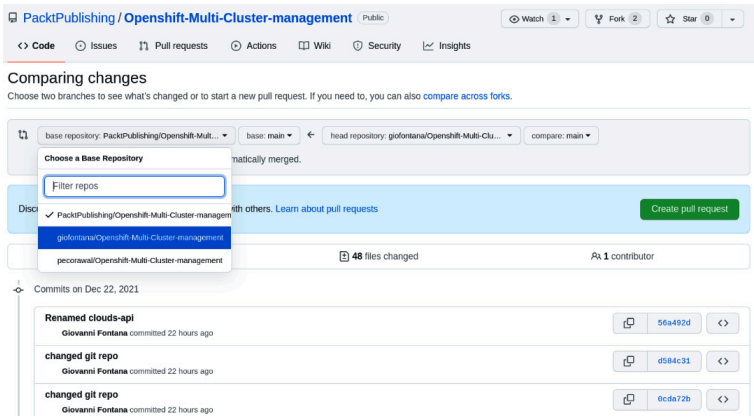


Figure 10.20 – Creating a PR

6. Then, select **`pre-prod`** in the **compare** field:

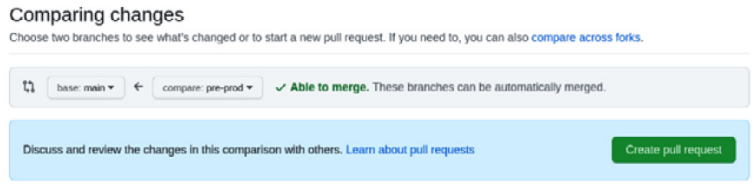


Figure 10.21 – Creating a PR

7. Now, fill out the form and click **Create pull request**:

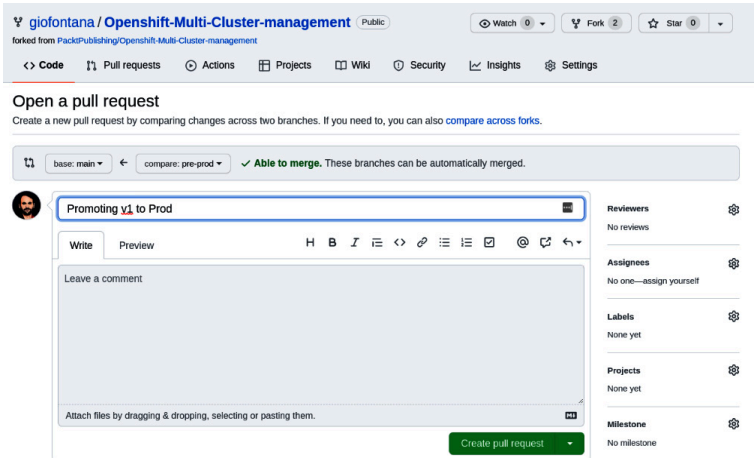


Figure 10.22 – Creating a PR

8. In a real-life scenario, this PR would be reviewed, approved by peers, and then merged. We are still practicing at the moment, so let's go ahead and click the **Merge pull request** button:

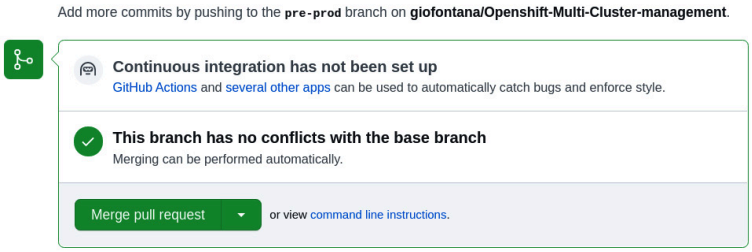


Figure 10.23 – Approving the PR

9. The overlay manifests for version 1.0 of the production environment are already in the **`main`** branch of our Git repository. This means we can deploy it using Argo CD:

```
$ git checkout main
```

```
$ oc apply -f ./chapter10/config/argocd/argocd-app-prod.yaml
```

10. At this point, you should have three applications on Argo CD:

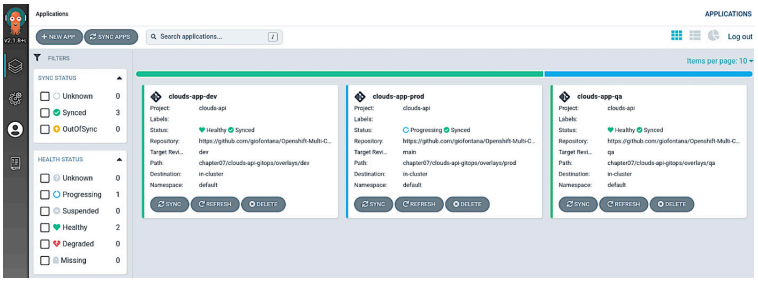


Figure 10.24 – Argo CD applications

11. Let's access the application to see version 1.0 of our application running in production:

```
$ curl $(oc get route clouds-api -n clouds-api-prod --template='http://{{.spec.host}}')
```

You should see the same response that you saw previously:

```
Welcome to the HomePage! Version=1.0
```

Congratulations! We have deployed our application using Argo CD into three different namespaces, each one representing a different environment: development, QA, and production. Since this book is intended to be about *multi-cluster*, we must learn how to do the same process but deploy into multiple clusters, instead of only one. In the next section, you will see that the process is the same, except you must change one parameter in Argo CD's **`Application`** object.

# Deploying to multiple clusters

We learned how to register external clusters in the *Configuring Argo CD against multiple clusters* section. As soon as you have multiple external clusters registered to Argo CD, deploying an application to one of them is simple – you only need to refer to the external cluster you registered in the **`destination`** field of Argo CD's **`Application`**. An example of this can be seen in the following manifest:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: clouds-app-dev-external-cluster
  namespace: openshift-gitops
spec:
  project: clouds-api
  source:
    repoURL: `https://github.com/PacktPublishing/OpenShift-Multi-Cluster-Management-Handbo
    path: chapter10/clouds-api-gitops/overlays/dev
    targetRevision: dev
  destination:
    server: 'https://api.<external-cluster>:6443'
    namespace: default
  syncPolicy:
    automated:
      selfHeal: true
```

You can create as many **`Application`** objects as you need, deploying only to the local cluster or including multiple external clusters. As you have seen, the deployment process itself is similar, regardless of whether you are deploying to a local or external cluster.

*IMPORTANT NOTE*

*When you work with multiple clusters, you need to pay special attention to the* **container image registry**. *The OpenShift internal registry, as its name suggests, should only be used internally in a single cluster; it is not suitable for multiple clusters. In such a case, an enterprise container image registry is recommended. There are multiple options on the market, such as Nexus, Quay, Harbor, and many others. In this book, we will cover Quay in* ***Chapter 13****, OpenShift Plus – a Multi-Cluster Enterprise-Ready Solution.*

# Summary

In this chapter, you learned about various concepts related to **GitOps**. You also learned about **Argo CD** and how to install it on OpenShift and use it. You also built and deployed a sample application to three different namespaces to simulate the *development*, *QA*, and *production* environments. Finally, you learned that deploying to the local or external cluster is a similar process – you only need to change the destination server field.

Argo CD allows you to establish an efficient and robust application delivery model using GitOps, in which you ensure *consistency*, *auditable changes*, and a *secure process*, no matter where you are deploying your applications. And the best part is that there is no additional cost to use it since it is included in Red Hat OpenShift's subscription. That said, if you

are deploying containerized applications on OpenShift, I strongly recommend that you try OpenShift GitOps and use the concepts we explored in this chapter.

In the next chapter, we will explore a great tool that will help you deploy and manage several OpenShift clusters from a single unified interface – **Red Hat Advanced Cluster Management**. This tool allows you to monitor, manage, define, and enforce policies and deploy applications to several clusters.

Let's move on and take a deep dive into Red Hat Advanced Cluster Management!

## Further reading

To find out more about the topics that were covered in this chapter, take a look at the following resources:

- *A History of GitOps:* **https://www.weave.works/blog/the-history-of-gitops**
- *Argo CD official documentation:* **https://argo-cd.readthedocs.io/**
- *Great tutorial about* **Kustomize**: **https://blog.stack-labs.com/code/kustomize-101/**
- *Bitnami's Sealed Secrets overview:* **https://github.com/bitnami-labs/sealed-secret**.

Previous chapter
‹ Chapter 9: OpenShift Pipelines – Tekton

Next chapter
Chapter 11: OpenShift Multi-Cluster GitOps and Management ›