O'REILLY®

Argo CD

Up & Running

A Hands-On Guide to GitOps and Kubernetes

Early Release

RAW & UNEDITED

Compliments of

AKUITY

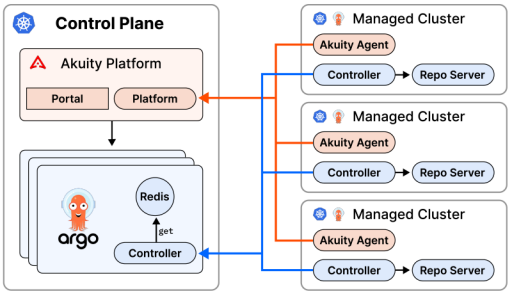
Andrew Block &
Christian Hernandez

AKUITY — The GitOps Platform for K8s

From the creators of Argo and Kargo, the Akuity Platform is the only end-to-end GitOps platform for enterprises.

> **Deploy –** Hybrid agent-based architecture for enterprise scale.

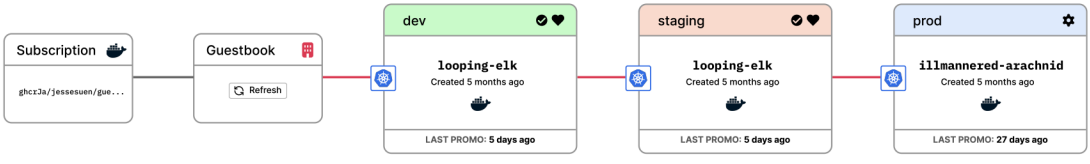
Agent Based Application Management



- **Security:** The agent-based architecture isolates secrets within the cluster, improving security.
- **Performance:** Resource updates are processed locally and only relevant changes are sent back to the control plane.
- **Scalability:** built-in and configured autoscaler automatically adjusts Argo CD resources.

> **Promote –** Leverage stage-to-stage promotion using GitOps principles without bespoke automation or CI pipeline.

- **Engineering Efficiency:** Reduced MTTR, fewer manual approvals, fewer deployment failures.
- **Deployment Speed:** Faster lead time, increased deployment frequency.
- **Risk Reduction:** Fewer security vulnerabilities, better compliance.
- **Developer Experience:** More self-service deployments, higher K8s adoption.



> **Monitor –** Real-time monitoring & comprehensive K8s dashboards across all cloud providers, regions, on-prem and edge locations.

Book a demo today!
https://akuity.io/get-in-touch

Argo CD: Up and Running

A Hands-On Guide to GitOps and Kubernetes

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Andrew Block and Christian Hernandez

O'REILLY®

Argo CD: Up and Running
by Andrew Block and Christian Hernandez

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: John Devins
Development Editor: Jill Leonard
Production Editor: Elizabeth Faerm
Copyeditor: TO COME
Proofreader: TO COME

Indexer: TO COME
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

May 2025: First Edition

Revision History for the Early Release

2023-05-04: First Release
2023-10-26: Second Release
2024-02-26: Third Release
2024-04-19: Fourth Release
2024-07-18: Fifth Release
2024-09-23: Sixth Release
2024-11-11: Seventh Release
2025-01-14: Eighth Release
2025-03-31: Ninth Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098142001* for release details.

978-1-098-14194-3

[LSI]

Table of Contents

Brief Table of Contents (*Not Yet Final*)

Preface

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the preface of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *jleonard@oreilly.com*.

Cloud Native technologies, regardless of where they reside (on the public cloud or in a private datacenter) continues to proliferate. For those running containerized applications, Kubernetes has become the de-facto solution for running and managing these applications at scale and as a result, several different architectural patterns have emerged over time. GitOps is one such pattern that describes a set of processes for managing infrastructure and applications within source code stored within a Git repository. While GitOps is not exclusive to Kubernetes, it has strong ties to Kubernetes as the practices and principles have become the cornerstone for managing the platform.

While GitOps provides a framework that defines how to align Infrastructure as Code (IaC) concepts for managing resources using content stored within source code management tools, there is still a need for a tool that can realize these goals and the declarative nature of the content. In the world of Kubernetes, Argo CD has become one of the most popular tools for implementing GitOps paradigms. Given its broad adoption within the Kubernetes community for use by both infrastructure and application teams, having an understanding how it can be used effectively is essential.

Who Should Read This Book

This book is primarily written for Kubernetes administrators and developers who want to utilize GitOps practices to improve the user experience around Cloud Native technologies along with those looking to operationalize Argo CD using the full set of features provided by the tool. However, since many Development teams are also leveraging Argo CD to deploy and manage their own Applications, those resources will also find most of the content applicable for their use as well. Upon the completion of thisBy book, you will be better equipped to implement Argo CD within your organization in a manner that supports production use.

Whether you just started your Argo CD journey or are a seasoned power user, we wrote this book to be applicable for all levels of experience. By including key topics and a set of relatable examples, this book will become a reference that you can use from day 1 and beyond

Why We Wrote This Book

Argo CD is one of the most popular tool sets in the Cloud Native Computing Foundation (CNCF) and is quickly becoming the de facto standard in GitOps implementation. Even with its popularity, best practices and getting started guides are sparse and scattered throughout the ecosystem. We wrote this book as a central place for those looking into operationalizing Argo CD, without having to scour the Internet for the information. Both of us have spent a large amount of time in the Open Source Community as well as various enterprise organizations assisting in the implementation of Argo CD in their own environment. We've collected our shared experiences and seek to be able to share them broadly so that others, like yourself, can become successful in your Argo CD journey.

Navigating This Book

The adoption of Cloud Native concepts is a journey. The following is a glimpse of what you can expect as you make your way through this book:

- Chapters 1-3 covers everything that you need for beginning to be productive working with Argo CD including the goals the project seeks to achieve, the installation methods, and common methods for interacting with the platform
- Chapters 3-5 places an emphasis on one of the most important topics within Argo CD: Applications. As the primary vehicle for managing resources in Kubernetes using GitOps, an in-depth overview of Applications will be provided including the tools that can be used to define Kubernetes manifests, the content source for these manifests, and how and when they are applied to Kubernetes clusters.

- Chapters 6-9 covers a number of topics that focus on the management of Argo CD including authentication and authorization, cluster management, multi-tenancy and security.
- Chapters 10-11 goes beyond the basics including advanced Application design and deployment patterns and extending the base functionality of Argo CD to take GitOps to new heights.
- Chapters 12-13 discusses some of the key areas that are applicable for using Argo CD within large organizations including how both the tool as well as GitOps in general can be incorporated into Continuous Integration and Continuous Delivery workflows as well as how to operationalize the platform at scale.
- Chapter 14 might appear to be the end of our journey with Argo CD. However, it is just beginning as this concluding chapter provides a number of resources for how to keep the conversation going with other members of the Argo CD community as well as areas for further exploration.

What This Book Will Not Cover

This book will focus on how to get up and running with Argo CD in a Kubernetes environment. This book will not go over how to install Kubernetes nor how to manage the lifecycle of a Kubernetes cluster. Furthermore, there are many ways to do the same thing. We will be focusing a lot on Helm in this book, however; that is not to say that using other methods aren't valid. It is impossible to go over every valid option. There are also many tools/projects that do similar things. Beyond Argo CD, a usage of a particular tool over another doesn't mean we are endorsing that tool or that we would use that particular tool all the time in every scenario. A lot of the time, we chose the tool for the sake of brevity. We will try and call out all these exceptions as we go over them.

Prerequisites

Before getting started, we will go through some of the "Prerequisites" on page xiii you might need in order to follow along in this book. We assume that you have access to an operational Kubernetes cluster, describe how to run an environment on your local machine using kind. However, we recommend that you test these out on a test system. For that, we recommend kind.

kind

Although the steps outlined in this Book should "just work" with most Kubernetes implementations; the exercises will make use of kind, a tool for running local Kubernetes clusters within container "nodes". You can get started with `kind` by visiting https://kind.sigs.k8s.io.

The `kind` website includes instructions on how to install the `kind` binary and any of the other "Prerequisites" on page xiii. Several providers are available which map to popular container runtimes, including docker, podman or nerdctl (containerd) which enables its use among a greater set of end users.

Helm

We use Helm routinely throughout the course of this book. so it will be necessary to have the Helm binary available in your `$PATH`. You can visit Helm for information about installation.

Kubernetes Client

Since we will be interacting with Kubernetes Clusters, it will be important to have the `kubectl` client available. You can follow the instructions on the official Kubernetes Documentation site: *https://kubernetes.io/docs/tasks/tools/*

Argo CD CLI Client

Argo CD comes with the `argocd` cli client that interacts with the Argo CD API server. You can follow the instructions found on the Argo CD website for installation of this client: *https://argo-cd.readthedocs.io/en/stable/cli_installation/*

YAML/JSON Processing

To make things easier, we use a lot of `jq` and `yq` to modify/update JSON/YAML in place. You can find information about these tools by visiting their respective websites:

- *https://stedolan.github.io/jq/download/*
- *https://mikefarah.gitbook.io/yq/*

If you're using Linux or a Mac, you might be able to find these utilities using their respective package manager (For example; you can run `brew install jq` on a Mac)

Companion Git Repository

Throughout this book, you will work through a series of exercises and examples as you expand your knowledge of Argo CD. These resources are available within a Git repository: https://github.com/sabre1041/argocd-up-and-running-book.

Since Git is the Source Code Management (SCM) tool for not only interacting with the companion repository, but also GitOp as a whole as well Argo CD, it is important that you also have Git installed locally on your machine too. Information related to Git including the supported installation options and platforms can be found on the GIt website.

Acknowledgments

Andy Block: They say that it takes a village to raise a child and this sentiment is certainly true for both the GitOps and Argo CD communities. It would not be possible to produce a publication, such as this book on Argo CD, without the continued support of the Open Souce community. In particular, I would like to thank Dan Garfield, who has helped shed a light into what it takes to build a business that is focused primarily on GitOps. In addition, I wanted to also thank Michael Crenshaw whose unbelievably deep knowledge of Argo CD has helped me time after time better understand all of the minute details of the project. These insights directly translated into the ongoing support that I am able to provide to community members along with material within this book.

Of course, I could not forget my colleagues at Red Hat who have both helped and supported my endeavors within the GitOps space. From Raffaele Spazzoli and our endless conversations on Helm, Kustomize and various GitOps patterns. To Gerald Nunn, and our thoughts and designs for what it takes to properly architect and operate GitOps as a platform service within some of the most regulated organizations in the world. And, to the entire OpenShift GitOps team. Thank you for making me feel like an extended member of your team where our ongoing collaboration has enabled our customers to apply GitOps principles at scale, using some of the most secure and trusted software available.

Finally, Argo CD is just one of many GitOps tools in the industry. There will never be a single GitOps tool, and we are all better because of that fact. A big thank you goes out to those in the GitOps community, including Scott Rigby, Alexis Richardson and Stacy Potter. Your continued partnership and collaboration is truly appreciated!

Christian Hernandez: The path to being a subject matter expert in a particular technology to the point where you write a book isn't a path you take alone. There have been many people in my career who have helped me get where I am. I would like to take this opportunity to give many thanks to those people.

My time at Red Hat was paramount to my development, and I couldn't have done it without the mentorship and leadership I received from Scott Cranton, Chris Morgan, and Erik Jacobs. Your leadership, mentorship, and willingness to let me grow was pivotal in my success. I cannot express my gratitude enough for everything. Also, to my "OG OpenShift TigerTeam" co-workers. We were lucky enough to work together during the best time of my career. Being able to work with experts in the field propelled me to be the best I can be. Also, a very special thanks to Chris Short who always pushed me to be the "Kelsey Hightower of GitOps".

Lastly, I would like to thank Hong Wang, Jesse Suen, and Alexander Matyushentsev. Creating the Argo Project was a bold and brave thing to do (even if you all didn't know it at the time). Growing with the Argo Project has been a privilege; and now

working with you all directly has elevated me to a level of expertise that I wouldn't have imagined. I am proud to be a part of your journey and I wouldn't be here without what you three have created.

We are deeply grateful to the tech reviewers for their meticulous attention to detail and technical expertise, which greatly enhanced the accuracy and quality of this book. We like to thank the following:

- Vladislav Bilay
- Manuel Dewald
- Werner Dijkerman
- Nadir Doctor
- Predrag Knežević
- Jess Males
- Benjamin Muschko
- Gerald Nunn
- Lipi Deepaakshi Patnaik
- Rick Rackow

Your invaluable feedback helped us refine complex concepts, ensuring clarity and precision for readers. The insights and suggestions you provided were instrumental in strengthening the technical depth and real-world applicability of the content. We sincerely appreciate the time and effort you dedicated to reviewing, catching errors, and offering thoughtful recommendations. This book is stronger because of your contributions, and we are truly thankful for your commitment to making it the best resource possible.

CHAPTER 1

Installing Argo CD

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *jleonard@oreilly.com*.

Like most cloud native applications, Argo CD features a microservices architecture that comprises multiple components and technologies. Each Argo CD component, working together, helps support a fault tolerant and robust system that helps enable the full set of features and capabilities. Understanding how all of these services work together in concert helps provide a greater awareness of the architecture, their significance, as well as how they are incorporated into the overall system as a whole. This chapter introduces the architecture and design of Argo CD along with detailing the various ways that it can be installed to a Kubernetes environment.

Argo CD Architecture

Since Argo CD is a GitOps based solution designed for Kubernetes, the architecture emphasizes the use of as many Kubernetes primitives as possible. As introduced in Chapter 1, Argo CD sources content stored in repositories and realizes those

configurations within a Kubernetes cluster. But, what does that look like and what are the components involved?

Kubernetes Controller Pattern

One of the key benefits of using Argo CD aside from the capability to define Kubernetes resources within source repositories and applying them to a cluster is that Argo CD can be configured to enforce those configurations to stay in place, even if they are modified. This is known as *drift management*. Argo CD does this by implementing a Kubernetes concept called a Controller which executes a non-terminating control loop for managing and monitoring the desired state of at least one resource. Based on a defined configuration, the controller will ensure that the current state matches the desired state.

For example, *Deployments* are a common method for registering workloads into Kubernetes and as part of the creation of a Deployment resource, a ReplicaSet is created. One of the built-in Kubernetes controllers, the ReplicaSet controller, monitors all pods that have been created for a given ReplicaSet and ensures that the actual state of the resource(s) in the cluster match the expected and defined state. If the actual state does not match the expected state, the controller will reconcile the difference until the current state matches the defined state.

Chapter 5 will cover divergence and diffing

This controller pattern applies to not only the resources that end users manage, but are foundational for Argo CD itself. The properties that drive the core configurations of Argo CD (everything from how to connect to external entities to the various properties of the Argo CD server) are stored within ConfigMaps and Secrets. However, as one can imagine, there is a limitation of the types of properties that can be stored within the simple key/value constructs provided by not only these Kubernetes resources, but any resources that are part of a standard Kubernetes deployment.

Argo CD is not alone when it comes to solutions for adding new ways of managing resources within a Kubernetes environment. This need led to the creation of Custom Resources which are implemented through CustomResourceDefinitions (CRDs) and enable an end user to register a new resource type within Kubernetes. By defining a new resource type, not only can the properties of this resource be defined (so that consumers can become aware and comply with the acceptable fields and their rules), but a new API endpoint in the Kubernetes API server is registered to facilitate the management of these resources.

A concept similar to a Kubernetes controller, known as an operator, builds upon the primitives of a Kubernetes controller for managing the current and desired state of resources in a Kubernetes cluster and applies them to CRDs. Given that Custom Resources typically have domain specific values and meaning associated with them, an operator is built with this domain level knowledge of how to interpret those values and ensure the state of resources within Kubernetes match those defined values.

Argo CD makes use of several Custom Resources and their properties are the primary vehicle to enable end users to manage their Kubernetes resources using GitOps based principles. The use of Kubernetes Controllers and Custom Resource are fundamental to the overall Argo CD architecture.

Argo CD Architecture Overview

Given that Argo CD implements a microservices based architecture, there is no single Argo CD component, but instead multiple distributed systems that act in a coordinated fashion. Figure 1-1 depicts the overall Argo CD architecture including the relationship between each of the services and resources.

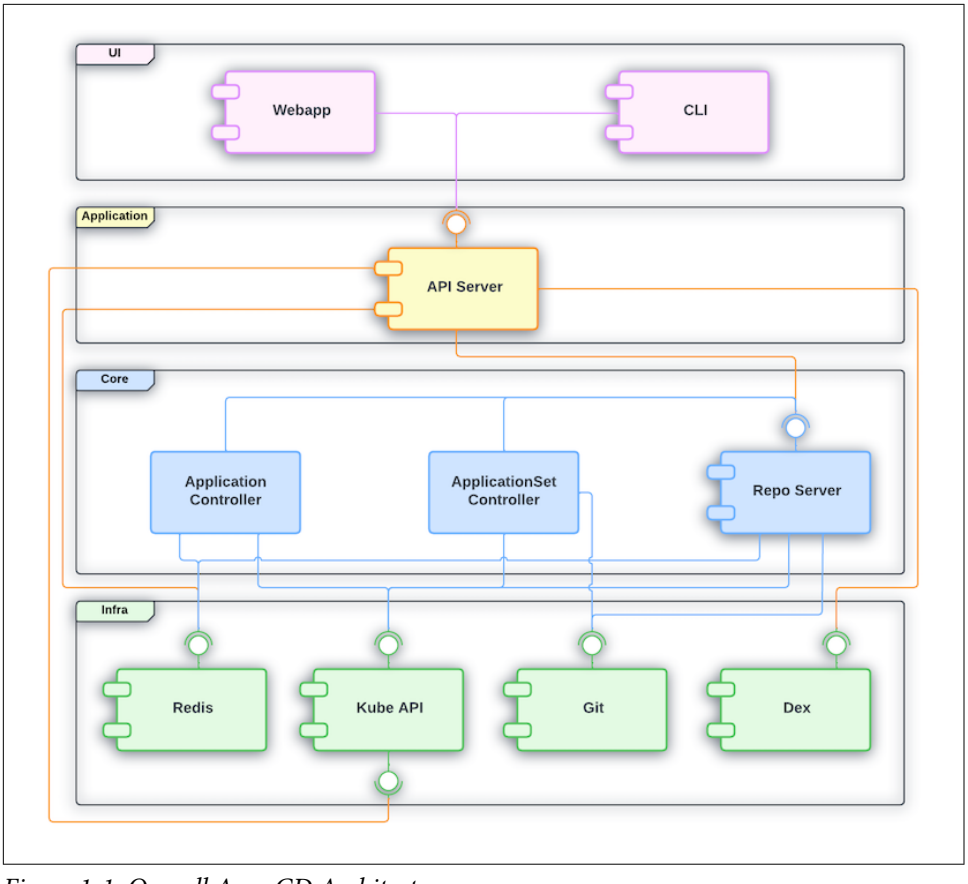


Figure 1-1. Overall Argo CD Architecture

Custom Resources

Argo CD makes use of several Custom Resources to declaratively define business logic and API's to implement GitOps management capabilities. Three Custom Resources are provided with each installation of Argo CD:

- `Applications`
- `AppProjects`
- `ApplicationSets`

The purpose of each Custom Resource will be described in more detail throughout the course of this book. However, it is important to note that Argo CD interacts with the Kubernetes cluster using these CRDs. This, effectively, makes these CRDs your interface for managing your Kubernetes cluster/clusters.

The Application Controller and ApplicationSet Controller are both Kubernetes Operators (and by definition, also controllers) that continuously monitor the state of `Application` and `ApplicationSet` resources and compare the live state in the Kubernetes cluster to the desired state from source repositories. In addition to strictly monitoring for Custom Resources, they are also responsible for performing lifecycle events associated with the content that they are reconciling.

Repository Server

The Repository Server maintains a local cache of the remote content source that will be translated into Kubernetes manifests. It is responsible for generating resources based on parameters including:

- Repository type
- Repository source location
- Path within the repository
- Template tool specific parameters

In addition, Custom plugins are also executed within this component as they can influence the generation of the Kubernetes resources.

API Server

The API server is a gRPC/REST based server that exposes services for managing key configurations that are integral to the platform including:

- Application management and status reporting
- Invocation of Application operations including syncing, rollback and additional user defined actions
- Cluster and repository management
- RBAC enforcement

Several other components within the Argo CD ecosystem heavily rely on this asset for their normal operation including the User Interface, CLI, and external CI/CD systems.

In addition to acting as an API server, a web User Interface is also exposed which provides a method for visualizing Application activity as well as supporting the management and configuration of Argo CD.

Redis

Redis is an in-memory database and provides local caching capabilities to reduce the dependency on external systems. While its primary purpose is to cache the contents

of remote repositories, it also supports the state of Kubernetes resources as well as connection status of remote repositories and clusters. The content of the cache is not persisted and is always rebuilt at startup.

Command Line Interface (CLI)

Command line based utility for interacting with Argo CD. Support is available to manage the configuration of the platform itself as well as the lifecycle of Applications. Communicates via the Argo CD API and includes a superset of the capabilities that are provided by the Argo CD user interface.

Single Sign On (SSO)

Argo CD provides user management capabilities for interacting with the platform. These users can be defined locally within Argo CD or can be sourced from an external source. When integrating with an external source, OpenID Connect (OIDC) based authentication is supported. For external identity providers that do not provide a direct OIDC integration, an instance of the Dex identity server is provided to act as a bridge between Argo CD and the remote identity provider.

Notifications

Included as part of a standard installation of Argo CD starting in version 2.3. This feature provides a mechanism for monitoring and triggering notifications to external systems based on the lifecycle of Applications through the use of templating capabilities and a catalog of included triggers. Argo CD Notifications can be configured to send information to (but not limited to) Slack, Email, and can also invoke other webhooks. Understanding the current state of systems and environments is key when running production systems and Argo CD notifications will be covered in greater detail within Chapter 13.

Argo CD Key Themes

As may be evident by this point, now that the foundational architecture has been introduced, Argo CD makes use of several key patterns and their significance will become even more apparent as each topic is described.

First, is an emphasis of defining resources in a declarative fashion; whether they be one of the provided Custom Resources, or a core configuration of the Argo CD server itself that is stored in a ConfigMap or Secret. Not only does trait implement one of the most important concepts in GitOps, it also enables the configuration of Argo CD itself to be managed via GitOps and Argo CD.

Building upon the first theme where each resource is managed in a declarative fashion, Argo CD also makes use of a stateless architecture, meaning that configurations are the state of the system that can be rebuilt at any time. This approach to a stateless

architecture is achieved by using etcd, the key/value datastore of Kuberntes to act as the persistent store for the aforementioned resources. If there is either a desire or need for state to be tracked against a particular resource, the `status` field, a standard property and method found on many Kubernetes resources, can be used. In addition, while Redis is included as a caching mechanism within the Argo CD architecture, it is used as a volatile cache without any long term persistence.

Finally, Argo CD enables extensibility. Not only are there multiple repository types which GitOps related content can be sourced from, but there is built in support for templating resources using a number of popular tools including Kustomize, Helm and Jsonnet. Additional user defined tools can also be added to not only integrate with additional external resources, but enhance how assets are rendered.

Now that we've covered the basics of the Argo CD architecture including the primary components, let's shift gears to the methods that are supported for installing Argo CD.

Installing Argo CD

Just as Argo CD supports the use of multiple methods and tools, such as Kustomize and Helm, to generate resources that can be applied to a Kubernetes cluster, many of these same tools and approaches can be used to install Argo CD itself. The determination of the particular approach depends largely on user preference as well as if there are any specific requirements or constraints, such as team or organizational guidance or restrictions. In addition to the tool that is used to facilitate the execution of the installation, Argo CD also supports several installation types which influences the resources that are included in the deployment as well as the configuration of the deployed resources. Some of these topics will be expanded upon in subsequent chapters.

Installation Types

Argo CD as a GitOps tool, similar to many other tools in this space, is utilized by a variety of personas who each have their own set of business domains and goals. Since there are a multitude of use cases and requirements that may be desired, Argo CD supports multiple installation configurations and the determination of a particular configuration depends on the answer to three key decision points:

- Who are the users and consumers of the platform?
- What is the scope Argo CD should manage?
- Is high availability a concern?

These options are illustrated in Figure 1-2:

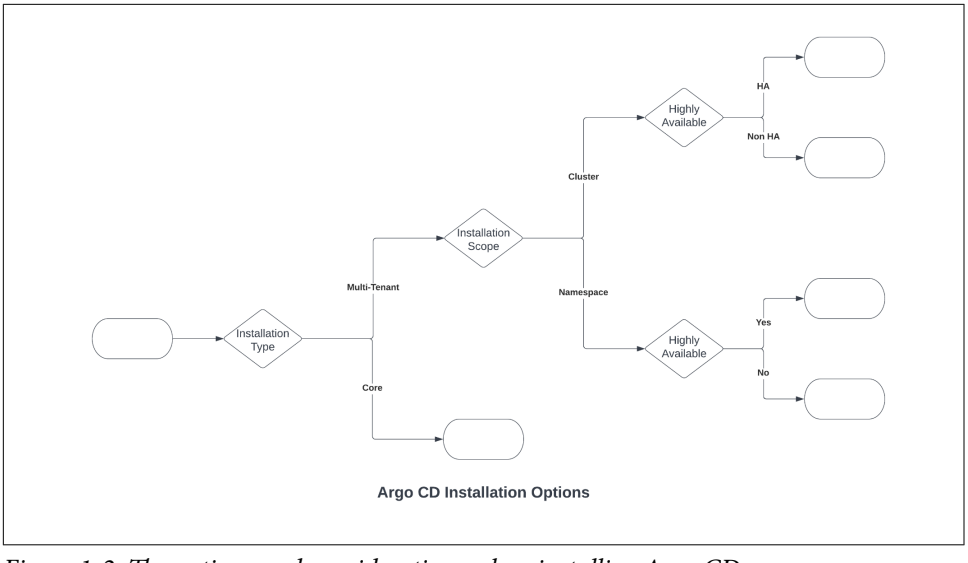


Figure 1-2. The options and considerations when installing Argo CD

The first decision point is the type of installation that Argo CD should serve. In most cases, Argo CD will be consumed by multiple individuals that may span across multiple teams within an organization. Additionally, most organizations desire to make use of the full set of features that are provided as part of a standard deployment of Argo CD (we covered these in the previous section). This is known as a *multi-tenant* type of installation and it is most commonly utilized as it provides the full set of capabilities provided by Argo CD.

Alternatively, an option is available to perform an installation that includes only the minimal set of components to support normal operation. This approach does not include the API server or user interface, SSO or notification features. In addition, each component is also optimized to consume a minimum amount of resources in a non-highly available configuration (more on the topic of high availability later in this chapter). While a core deployment is not intended to appease the masses, this approach is beneficial for individual users that manage Argo CD from both an administrative and end user perspective where there is not a desire to leverage the full featureset of Argo CD, but there is still a desire to take advantage of the primary GitOps capabilities.

The next decision point that must be addressed is the scope that Argo CD should manage. By default, Argo CD has the authority to control resources across an entire Kubernetes cluster. This is the preferred option, especially when Argo CD is being used by Kubernetes cluster administrators. However, another approach, known as "namespaced" mode, that is available is to deploy Argo CD within a specific namespace and to allow Argo CD to only manage resources within specific namespaces.

This option is used in multi-tenant environments where individual application teams are given the autonomy to operate their own instance of Argo CD, but are not granted access to manage cluster scoped resources. An in depth look into the use of a namespaced deployment of Argo CD and its use case will be discussed in Chapter 8.

Finally, to support production environments, each of Argo CD's components can be configured in an optimized manner to ensure greater resiliency and performance needs. This approach is accomplished through a combination of increasing the replica count as well as enabling tunable parameters within each component. However, there are certain considerations that must be followed so that Argo CD can operate in an optimized fashion as merely increasing the replica count of all components uniformly can actually cause a performance degradation. Fortunately, Argo CD provides manifests supporting both clustered and namespace scoped deployments illustrate the types of configurations necessary to enable a highly available deployment.

Now that both an overview of the Argo CD architecture as well as an understanding of the deployment approaches have been addressed, it's time to see Argo CD in action by working through the first hands-on activity..

Deploying Argo CD

In due course throughout the remainder of this book, most of the installation types and approaches will be realized. However, to get started, let's start off by performing a basic installation of Argo CD to our `kind` environment.

We've covered multiple approaches for performing an installation of Argo CD. Each of these options are detailed within the installation section of the Argo CD documentation.

Deploying Argo CD Using YAML Manifests

The simplest and most straightforward option is to use one of the raw YAML formatted manifests that include all of the resources and configurations within a single document, and in particular, a non-highly available, multi-tenant based deployment of Argo CD.

First, ensure that a fresh `kind` cluster is running.

```
"kind" on page xiii create cluster
```

Note: By default, the name of the cluster that the "kind" on page xiii tool creates is called "kind" on page xiii. You are free to change the default behavior by specifying an alternate name using the `--name` parameter of the "kind" on page xiii `create cluster` command or by setting the environment variable `KIND_CLUSTER_NAME` with the desired name.

Once the cluster has started, your `kubectl` context will be automatically updated and ready to utilize the newly created cluster. Execute the following commands using `kubectl` to create a new namespace called `argocd` and to deploy Argo CD in the previously described configuration:

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

After a few moments to allow for the associated images to be downloaded to the `kind` cluster, the pods in the `argocd` namespace can be queried with a result similar to the following:

```
kubectl get pods -n argocd


NAME                                                READY   STATUS
RESTARTS   AGE
argocd-application-controller-0                     1/1     Running
0          46s
argocd-applicationset-controller-74575b6959-8dc7l   1/1     Running
0          46s
argocd-dex-server-64897989f8-qg8pm                  1/1     Running
0          46s
argocd-notifications-controller-566bc99494-7vj82    1/1     Running
0          46s
argocd-redis-79c755c747-867nk                       1/1     Running
0          46s
argocd-repo-server-bc9c646dc-6sd86                  1/1     Running
0          46s

argocd-server-757fddb4d7-xgdxh                      1/1     Running
0          46s
```

The standard deployment of Argo CD depicted here contains each of the primary components that are included with Argo CD, so it is an ideal baseline to work from.

The User Interface is one of the key features that sets Argo CD apart from other GitOps solutions. Before moving on, confirm the successful installation of Argo CD by accessing the User Interface.

By default, the set of resources that were applied to the Kuberntes cluster did not include any configurations or resources to expose access to Argo CD outside the cluster. While there are several approaches that can be used to access Argo CD externally, such as creating a `LoadBalancer` service type or using an Ingress, to demonstrate baseline functionality, the port forwarding capability of the `kubectl` CLI can be used to connect to Argo CD without any additional actions.

Next, execute the following command to initiate the forwarding of port 8080 from the local machine to the Argo CD server service which will expose access to the User Interface.

```
kubectl port-forward svc/argocd-server -n argocd 8080:443
```

The command will establish a tunnel to facilitate the connection and block additional commands from being entered while the tunnel is established. If additional commands need to be executed while ports are forwarded, launch another terminal.

With access to the Argo CD user interface available due to the `port-forward` tunnel, navigate to *https://localhost:8080*

By default, Argo CD generates a self signed TLS certificate to enable secure transmission between itself and the browser. Since this certificate is not trusted by the browser, a warning is displayed. Depending on the browser being used, there will be an option to proceed even though the certificate is not trusted and then the Argo CD login page will be displayed.

To login, `admin` is the username of the Argo CD administrator and the password is a secret with the name `argocd-initial-admin-secret`. Obtain the password by executing the following command:

```
kubectl -n argocd get secret argocd-initial-admin-secret -o json-
path="{.data.password}" | base64 -d; echo
```

Login using `admin` as the username and the password that was obtained from the prior command. Upon successful login, the Argo CD dashboard is displayed, as shown in Figure 1-3.

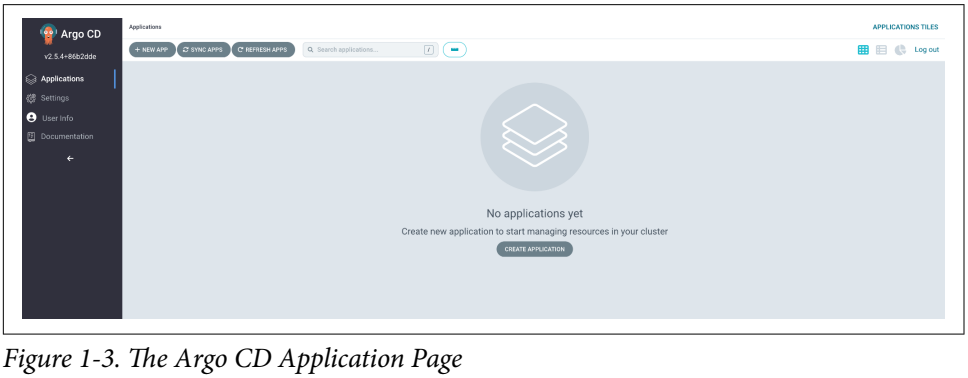


Figure 1-3. The Argo CD Application Page

The dashboard contains a list of the current *Applications* that have been registered to Argo CD and their current status. Since this instance does not have any Applications registered, the dashboard is empty. Feel free to navigate around the user interface as you see fit. However, a more in depth overview of the user interface will be covered in Chapter 2.

High Availability. The standard deployment of Argo CD is ideal for getting started, but is not suitable for production environments due to the fact that there is only a single replica for each component. In case one of the components fails (due to an error or issue with the underlying infrastructure), it will cause a degradation of functionality as one or more of the resources will become unavailable. To mitigate these concerns, an alternate set of YAML definitions are available for both cluster and namespaced modes of operation. The key difference between these sets of resources and those that were deployed previously is that not only have additional tuning options been implemented, but multiple replicas of each service have also been defined. This means that if a failure does occur to one of the services, the remaining replica will be able to take on requests and continue normal operation in a degraded state until the original replica returns to normal operation.

Given that the topic of high availability is just one of the many traits of a production system, this topic will be expounded upon in Chapter 13 as part of a discussion on the considerations for operating Argo CD at scale.

Deploying Argo CD Using Helm

Argo CD can also be installed using a Helm chart. A Helm based installation approach has advantages over YAML manifests as the resources that are installed can be customized using the dynamic templating capabilities provided by Helm. For example, entire components can be enabled or disabled as well as specific properties can be tailored whereas these options would not be possible using the YAML based manifest approach.

To install Argo CD using Helm, first be sure that your `kind` cluster does not have any previously created resources deployed. If Argo CD is still running from the prior section, the `kind` cluster can be deleted and recreated or the contents from the prior section can be removed.

To delete and recreate the "kind" on page xiii cluster, using the following commands:

```
"kind" on page xiii delete cluster
"kind" on page xiii create cluster
```

Alternatively, instead of needing to recreate the entire `kind` cluster, the YAML based manifest installation of Argo CD can be uninstalled by removing the resources from the same manifest and then deleting the `argocd` namespace:

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

    kubectl delete namespace argocd
```

With a fresh `kind` cluster available, proceed to deploy Argo CD using Helm.

First, add the Argo CD Helm repository:

```
helm repo add argo https://argoproj.github.io/argo-helm
```

Install the Helm chart using the default configuration and create a new namespace called argocd using the following command:

```
helm upgrade -i argo-cd argo/argo-cd -n argocd --create-namespace
```

Either the helm install or helm upgrade command can be used to install the Argo CD chart. When the helm upgrade command is used with the -i parameter, Helm will check if there is an existing release found. If a release is not found, the chart will be installed instead of upgraded. The benefit of using helm upgrade in this situation is that the same command can be issued regardless of installing a chart for the first time, or upgrading an existing release. The helm install command can only be used when installing a chart for the first time.

Another benefit of Helm is that chart creators can include additional information that is displayed whenever a chart is installed or upgraded, known as NOTES. After executing the helm upgrade command previously, the contents of the NOTES document in the chart was displayed which provided a set of next steps, including how to access the Argo CD user interface and how to obtain the password for the Argo CD admin user.

Query the running pods from the argocd namespace and take note that the set of resources are available as they were using the YAML manifest approach (albeit with slightly different names as the Helm chart prefixes each resource with the name of the Helm release).

```
kubectl get pods -n argocd
```

If desired, the Argo CD user interface can be accessed in a similar manner as described in the previous section and the exact steps in this instance can be found within the provided Helm NOTES output.

While only a basic deployment of Argo CD was described in this section, the full set of tunable parameters provided by the Argo CD Helm chart can be viewed by listing the available chart *Values*:

```
helm show values argo/argo-cd
```

The use of helm values within the Argo CD Helm chart enables a greater level of customization and simplifies the initial configuration when deploying Argo CD.

Summary

This chapter provided an overview of the architecture and components that are included as part of a deployment of Argo CD. In addition, two of the most common approaches for installing Argo CD, YAML manifests and Helm charts were introduced and used to deploy Argo CD to a Helm cluster. Finally, the Argo CD was accessed using the user interface to confirm a successful installation. The next chapter expands upon the use of the user interface and describes the various different methods available that can be used to manage and interact with Argo CD.

CHAPTER 2

Interacting with Argo CD

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *jleonard@oreilly.com*.

Argo CD includes a fully declarative configuration model which supports a hands-off approach to GitOps and the management of a GitOps server. However, in most cases, more direct methods will be needed for interacting with the Argo CD server. In the previous chapter, we covered the Argo CD user interface which is one method for interacting with the platform since it provides a visual approach to the current state of GitOps. While the user interface may be one of the most common methods for utilizing Argo CD, there are additional mechanisms to choose from. This chapter builds upon the foundational concepts established in Chapter 1 for accessing and configuring Argo CD along with introducing several additional approaches that can be employed depending on the use case or preference.

The User Interface In Depth

In Chapter 1, the TypeScript based user interface was used as a way to access Argo CD. However, when deployed to the `kind` cluster, it required establishing a connec-

tion to the server component using the `kubectl port-forward` command. While this was acceptable for initial testing and validation, it is by no means how one should utilize a service long term. A more robust approach should be undertaken to provide a more reliable exposure of services.

One of the most common methods for exposing services and gaining access to resources within a Kubernetes cluster is to leverage an Ingress resource. An Ingress provides a means for exposing services outside of a cluster and they are enabled by the use of an Ingress Controller which will map the incoming request to the backend service. There are many options as it relates to the available Ingress controllers where some have additional features and integrations with the operating environment, such as a cloud provider.

NGINX is one such popular ingress controller and there is support for deploying it to a `kind` cluster. "kind" on page xiii clusters can be customized to include advanced configurations, such as setting options for kubeadm, the tool for deploying Kubernetes clusters, and to deploy multiple "nodes" to support simulating high availability scenarios.

Another available option is to forward local ports to the `kind` node – a capability to enable ingress into the Kubernetes cluster and in this case, the NGINX ingress controller.

Create a new "kind" on page xiii cluster and pass an inline definition of a `kind` configuration:

```
cat <<EOF | "kind" on page xiii create cluster --config=-
"kind" on page xiii: Cluster
apiVersion: "kind" on page xiii.x-k8s.io/v1alpha4
nodes:
- role: control-plane
kubeadmConfigPatches:
- |
"kind" on page xiii: InitConfiguration
nodeRegistration:
  kubeletExtraArgs:
    node-labels: "ingress-ready=true"
extraPortMappings:
- containerPort: 80
hostPort: 80
protocol: TCP
- containerPort: 443
hostPort: 443
protocol: TCP
EOF
```

Alternatively, the configuration definition can be placed into a file and referenced using the same `--config` parameter when creating the cluster.

Once the cluster has started, deploy the NGINX ingress controller using Helm.

First, add the NGINX Ingress Controller Helm repository and install the NGINX ingress controller chart:

```
helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
helm repo update
```

Once the repository has been added, create a new file called `values-ingress-nginx.yaml` to contain customized Helm values for the NGINX ingress controller with the following content:

```
controller:
service:
type: NodePort
hostPort:
enabled: true
updateStrategy:
type: Recreate
```

Install the Helm chart for the NGINX ingress controller using the customized values created previously using the following command:

```
helm -n ingress-nginx install ingress-nginx ingress-nginx/ingress-nginx --
create-namespace -f values-ingress-nginx.yaml
```

Wait until the ingress controller is ready.

```
kubectl wait --namespace ingress-nginx \
--for=condition=ready pod \
--selector=app.kubernetes.io/component=controller \
--timeout=90s
```

Query the pods and services in the `ingress-nginx` namespace to view the resources that were just deployed

```
kubectl get pods -n ingress-nginx

NAME                                        READY   STATUS      RESTARTS   AGE
ingress-nginx-controller-56f6595fc8-74t7s   1/1     Running     0          3m33s

kubectl get svc -n ingress-nginx

NAME                                 TYPE        CLUSTER-IP     EXTERNAL-IP
PORT(S)                      AGE
ingress-nginx-controller             NodePort    10.96.33.103   <none>
80:30579/TCP,443:31111/TCP   4m13s
ingress-nginx-controller-admission   ClusterIP   10.96.168.33   <none>
443/TCP                      4m13s
```

Since the `kind` cluster was created binding port 80 and 443 of the local machine to the `kind` node, invoking the `curl` command against port 80 should verify communication to the ingress controller

```
curl http://127.0.0.1

<html>
<head><title>404 Not Found</title></head>
<body>
<center><h1>404 Not Found</h1></center>
<hr><center>nginx</center>
</body>
</html>
```

While the 404 error may appear to be a failure, given that no Ingress resource has yet to be created, the fact that a response was provided and that it included nginx in the response body confirms the ingress controller has been deployed and is operating correctly.

Now that access to the ingress controller has been confirmed, an Ingress resource must be created so that access can be achieved through the ingress controller. Fortunately, the Argo CD Helm chart includes functionality for configuring this task. When utilizing an ingress resource, one of the first steps that must be completed is to determine the hostname that is associated to the service.

Since NGINX is a OSI Layer 7 load balancer, routing is performed using the Host header in the request. As requests are received, NGINX will inspect this header, determine if any of the defined Ingress resource matches the request, and if so, route the request to the associated backend service.

The hostname that will need to be specific for the Argo CD instance will most likely not have an associated value in a publicly accessible DNS server. To solve this challenge, two options are available:

- Modify the contents of the `/etc/hosts` file on the local machine
- Use a hosted wildcard DNS service, such as nip.io

While using the hosted service eliminates the need to make modifications on the local machine, it potentially introduces an unnecessary dependency on an external. As such, the manual modification approach will be demonstrated here.

`argocd.upandrunning.local` is the hostname that will be used to refer to the Argo CD instance deployed within the "kind" on page xiii cluster. Modify the `/etc/hosts` file to add the loopback address of the local host and hostname so that queries are resolved and routed appropriately.

Append the following to the end of the `/etc/hosts` file

```
127.0.0.1 argocd.upandrunning.local
```

With the ingress controller and hostname "Prerequisites" on page xiii complete, the Argo CD Helm chart has the capabilities available to support generating the necessary manifests to enable Argo CD to be accessed via an Ingress resource.

Create a file called `values-argocd-ingress.yaml` with the following content:

```
---

server:
ingress:
enabled: true
hostname: argocd.upandrunning.local
ingressClassName: nginx
extraArgs:
- --insecure
```

Deploy the Helm chart using the values file created previously.

```
helm upgrade -i argo-cd argo/argo-cd --namespace argocd --create-namespace -f
values-argocd-ingress.yaml
```

This Helm release will appear similar to the release that was completed in Chapter 1. However, by specifying the appropriate values, a new Ingress resource was created which can be verified by executing the following command:

```
kubectl get ingress -n argocd

NAME                    CLASS   HOSTS                       ADDRESS     PORTS
AGE
argo-cd-argocd-server   nginx   argocd.upandrunning.local   localhost   80
53s
```

Given that all of the pieces are in place in order to access Argo CD using an Ingress resource, open a web browser and navigate to *http://argocd.upandrunning.local* which should display the Argo CD User Interface login page.

Argo CD can also be accessed securely through the ingress controller by using the analogous https:// address Similar to when the user interface was accessed in Chpater 2, a warning will be displayed signifying that a connection is attempting to be established to an endpoint whose certificates are not trusted by the browser. One key difference is that the untrusted certificates are related to the ingress controller and not Argo CD. The default ingress configuration the Helm chart establishes terminates TLS traffic at the ingress controller instead of at Argo CD itself. A more in depth discussion on the TLS options that can be configured in Argo CD will be discussed in Chapter 9.

Once again obtain the password for the Argo CD admin user from the `argocd-initial-admin-secret` secret and login. Let's take an opportunity to explore the various configuration options that are available within the user interface.

The default landing page upon login contains the list of Applications that have been registered to Argo CD. An Application is a source of GitOps content that targets a particular destination environment and is the primary focus for end users when using Argo CD. The user interface enables the creation, management and synchronization of Application resources in a visual, user friendly manner.

Aside from managing applications, the other primary purpose of the user interface is to facilitate the management of the Argo CD server itself – from within the Settings page.

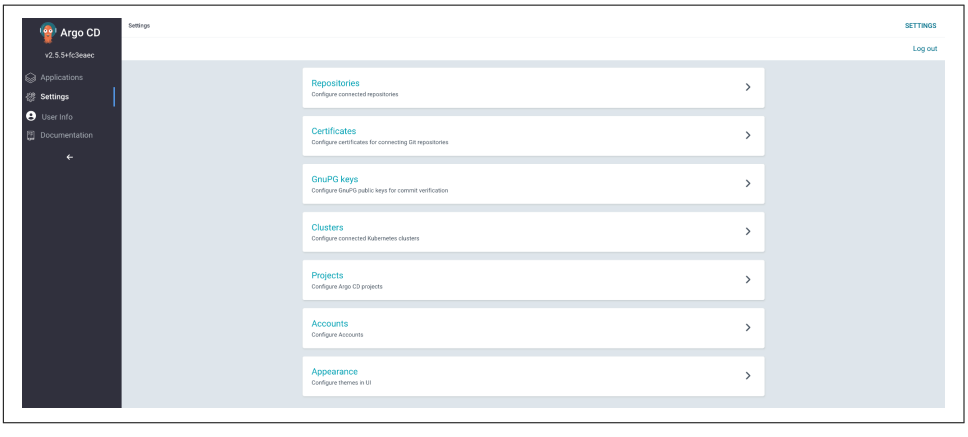


Figure 2-1. Argo CD Settings Page

The following table details the configurable options that are made available from the Settings page of the user interface:

Table 2-1. Options available within the Argo CD Settings Page

Setting	Description
Repositories	Configure remote locations containing resources that will be translated into Kubernetes manifests
Certificates	Management of transport mechanisms to facilitate secure connectivity to remote repositories
GnuPG keys	Key management to enable the verification of source control content
Clusters	Kubernetes environments that have content from source repositories applied
Projects	Logical groupings of Applications with common configurations and permissions
Accounts	Management of local accounts stored within the Argo CD server
Appearance	Configure the look and feel of the user interface

In addition to being able to manage Application and server settings, information related to the current authenticated user is available from the User Info page which is helpful for associating identity details to enable Role Based Access permissions that are used to manage access to Argo CD resources.

Even with all of the parameters and settings that can be configured within the Argo CD user interface, there are still a large number of properties that either cannot be managed using the user interface or their values are read only. When those situations do arise, the solution can be typically facilitated by using the Argo CD CLI. The next section introduces the capabilities included with the Argo CD CLI along with applying the appropriate settings to enable the management of the `kind` Argo CD environment.

The Argo CD Command Line Interface

The Argo CD Command Line Interface (`argocd`) is a utility to control and manage the Argo CD server. Similar to the User Interface, the CLI leverages the API to facilitate the interaction with Argo CD. When certain options are not available from within the User Interface, the default option is to use the CLI as it includes a more in depth set of options and capabilities as compared to the user interface.

Installing the Argo CD CLI can be performed on most major operating systems as there are prebuilt binaries readily available. Other installation options are also available depending on the target operating system and the CLI is also available in a number of formats, including a container image with the CLI included. Consult the Argo CD CLI installation documentation (*https://argo-cd.readthedocs.io/en/stable/cli_installation*) for the list of supported platforms and necessary steps to complete the installation.

Once the CLI has been successfully installed, execute the `argocd` command to see a list of functions that can be managed using the tool. Unfortunately, the majority of the options enabled from within the CLI cannot be used unless a connection to an Argo CD environment is established. Connect the CLI to the `kind` Argo CD instance using the `argocd login` command as shown below.

```
argocd login --insecure --grpc-web argocd.upandrunning.local
```

The –grpc-web parameter enables the use of the gRPC-web protool which enables communication through the ingress controller. Additional configuration steps are needed to enable native gRPC connectivity which will not be covered at this time.

When prompted, enter the Argo CD admin username and password to authenticate the CLI to the `kind` Argo CD instance.

As soon as successful authentication is achieved, a configuration file containing details related to the user, the Kubernetes context and other connectivity data is created in a file located at `$HOMEDIR/.config/argocd/config`. `$HOMEDIR` uses either an environment variable named `ARGOCD_CONFIG_DIR`, `HOME`, or `XDG_CONFIG_HOME` (XDG Base Directory Specification) depending on which value is resolved first.

One of the first steps that is typically taken after logging in via the CLI is to change the default admin password. Changing the admin password is an important step as it increases the security posture of the Argo CD server. Kubernetes Secrets are not encrypted, but base64 encoded which enables entities with access to query Secrets access to the default password.

Change the default admin password by executing the following command:

```
argocd account update-password
```

Enter the current admin password and the value of the desired password to reset the admin password.

Confirm the new password was applied properly by logging out of the current session and logging in once again with the updated password.

```
argocd logout <context>
```

The value of the `<context>` property refers to the argocd context to target. The list of argo cd contexts can be queried by executing the `argocd context` command. After logging out, login again using the `argocd login` command to confirm the password reset was successful.

Changing passwords for Argo CD users is just one action that cannot be accomplished using the User Interface and is one of the benefits provided by the CLI. The use of the CLI will become even more prevalent in upcoming chapters as it provides capabilities for not only administrators and users, but its inclusion and integration into other systems and workflows .

However, what if there was no reason at all to use either the User Interface or the CLI but still get the benefits of being able to manage Argo CD? The final section of this chapter explores two additional methods for interacting with Argo CD.

Additional Methods for Managing Argo CD

The Argo CD User Interface and CLI simplifies how users interact with Argo CD – either through visualization and accessibility features from the perspective of the User Interface or enabling a command line level approach with the CLI. One of the commonalities between these two components is that they both make use of the REST based API that Argo CD exposes. End users can invoke the same API's that Argo CD exposes without being limited based on the features that are included in either the User Interface or CLI.

The first question that may come to mind is "what type of information does Argo CD make available via the API?" One approach could be to use the developer console included by the web browser to inspect the requests that are being invoked from the

Web Console. But, that would be somewhat tedious for being able to determine the exact endpoint and parameters that need to be included.

Fortunately, Argo CD provides a OpenAPI Specification (Sometimes also called Swagger) which describes all of the API's, including the acceptable inputs and provided outputs, that are exposed grealy reducing the burden on the end user.

The OpenAPI specification provided by Argo CD is located at the endpoint `/swagger.json`

Open a web browser and navigate to *https://argocd.upandrunning.local/swagger.json* to view the contents.

Upon loading the OpenAPI specification document, one quickly realizes how verbose a specification can be. With a mature API, such as Argo CD, the document is quite large.

One of the tools provided by the Swagger project is a visualization component for OpenAPI specifications which avoids needing to become familiar with the intricate details of the OpenAPI specification.. This utility is included with Argo CD and can be accessed by navigating to *https://argocd.upandrunning.local/swagger-ui*.

Now that there is an understanding of the API services to query, what are the steps necessary to invoke them? First, a session token must be generated by invoking the `/api/v1/session` endpoint with a valid username and password.

Execute the following command to obtain a session token substituting the username and password in the appropriate fields:

```
curl -k https://argocd.upandrunning.local/api/v1/session -d '{"username":"<USER-
NAME>","password":"<TOKEN>"}'
```

Note: The -k argument disables TLS validation which would have thrown an error similar to what was seen previously when navigating to the Argo CD User Interface from the web browser.

A successful authentication attempt will result in a similar response to the following:

```
{"token":"<TOKEN>"}
```

With a valid session token, the available API endpoints can be invoked.

One of the most important endpoints that is frequently queried, especially during the initial configuration of Argo CD, from any of the invocation methods discussed. The Argo CD settings endpoint is exposed at `/api/v1/settings` which can also be verified within the Swagger UI interface by selecting *SettingsService* and viewing the GET request listed below.

By expanding the responses, it is a wealth of information, much more than is provided by the CLI or the User Interface.

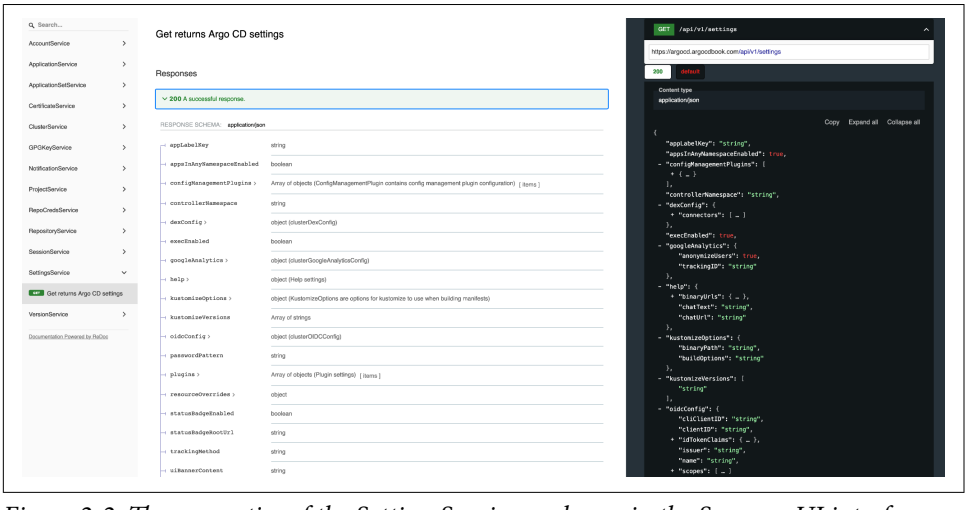


Figure 2-2. The properties of the SettingsService as shown in the Swagger UI interface

To invoke this endpoint, execute the following command substituting the value of the Bearer (Session) token obtained previously:

```
curl -k -H "Authorization: Bearer <TOKEN>" https://argocd.upandrun-
ning.local/api/v1/settings | jq -r
```

A response similar to the following should be displayed:

```
{
"appLabelKey": "argocd.argoproj.io/instance",
"resourceOverrides": {
"apiextensions.k8s.io/CustomResourceDefinition": {
  "ignoreDifferences": "jqPathExpressions: null\njsonPointers:\n- /sta-
tus\n- /spec/preserveUnknownFields\nmanagedFieldsManagers: null\n"
}
},
"googleAnalytics": {
"anonymizeUsers": true
},
"kustomizeOptions": {
"BuildOptions": "",
"BinaryPath": ""
},
"help": {
"chatText": "Chat now!"
},
"passwordPattern": "^.{8,32}$",
"controllerNamespace": "argocd"
}
```

Viewing server settings is just one of the many API endpoints that can be not only queried, but updated, and adds additional weapon to the already robust arenenal of tools that are used to manage Argo CD.

However, what if there was a desire to not leverage any of these tools or any services provided by Argo CD whatsoever, but still retain the benefits and assurances of a well maintained environment?

Recall back in Chapter 1, Argo CD supports a declarative model for managing GitOps and that the Argo CD implements the controller pattern to track the state of resources bassed on defined manifests. While Argo CD responds to changes to Custom Resources, such as Application's and AppProject's, it also tracks additional resources, such as ConfigMaps and Secrets which are used to influence the configuration of the entire platform. So instead of using the User Interface, CLI or invoking the API, the configurations can be applied directly to the Kubernetes cluster.

Each deployable in the Argo CD architecture makes use of configuration properties stored within ConfigMaps and Secrets in the same namespace that Argo CD are deployed within. Some of these resources use a well known and established name, like a ConfigMap with the name `argocd-cm` which contain the primary configuration properties for Argo CD, while others use metadata within each resource to signify their importance and intended capabilities. Indeed the server settings API endpoint that was invoked previously queried the contents of this ConfigMap.

There are a number of Argo CD configurations that can be defined within ConfigMaps and they are detailed in the following table:

Table 2-2. Common Argo CD Configurations

Resource Name(s)	"kind" on page xiii	Description
argocd-cm	ConfigMap	General Argo CD configuration
argocd-cmd-params	ConfigMap	Argo CD environment variable configurations
argocd-rbac-cm	ConfigMap	RBAC configuration
argocd-ssh-known-hosts-cm	ConfigMap	SSH known-host configuration data

Additional resources that influence the configuration of Argo CD are stored within Secrets and do not make use of a standardized naming convention for the resource. Instead, a label with the key `argocd.argoproj.io/secret-type` is placed on the Secret to denote their significance.

For example, a secret with the label `argocd.argoproj.io/secret-type: repository` contains connection details to a remote content source repository. As Argo CD can manage content from multiple remote repositories at a time, by using the label approach, similar content with distinct values for each repository can be applied

within separate Secret resources and then correlated appropriately based on the content.

The following table provides an overview of the the different Secret types and their significance:

Table 2-3. Argo Secret Types

Label	Description
`argocd.argoproj.io/secret-type: cluster`	The definition, configuration and credentials associated with a remote cluster
`argocd.argoproj.io/secret-type: repository`	Consolidated configurations and credentials associated with a remote repository
`argocd.argoproj.io/secret-type: repo-config`	Configurations associated to a remote repository (Not widely used)
`argocd.argoproj.io/secret-type: repo-creds`	Credentials for communicating with a remote repository

As topics are introduced throughout the course of this book, the way in which these resources can be used will come into focus to enable an entirely hands off approach for managing Argo CD server configuration.

This chapter introduced several methods to aid in the management of Argo CD: a visual user interface, an interactive command line utility, a comprehensive API, as well as an entirely declarative model. These options empower a freedom of choice for Argo CD administrators and end users towards using a tool or approach they feel the most comfortable using. Chapter 3 focuses on one if not the most foundational topics in the realm of Argo CD and how it is the centerpoint for facilitating GitOps practices. Managing Applications.

CHAPTER 3
Managing Applications

> **A Note for Early Release Readers**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *jleonard@oreilly.com*.

Argo CD manages the lifecycle of Kubernetes resources using a construct called `Applications`. An Argo CD `Application` is a Custom Resource Definition that contains a logical collection of related Kubernetes resources (i.e. a collection of YAML or JSON files). An Argo CD `Application` is the smallest unit of work in Argo CD and is where Argo CD interfaces with Kubernetes in order to deploy Kubernetes Objects.

In this chapter, we will cover the basics of an Argo CD `Application`, the different components including an overview of the types of sources `Applications` can connect to, and how to use different Kubernetes templating tools that Argo CD natively supports. To wrap up this chapter, we'll cover the lifecycle of an Argo CD `Application`.

Application Overivew

As previously mentioned, an Argo CD `Application` is the atomic working unit in Argo CD. It defines the end state of the desired set of resources within a Kubernetes cluster. Let's take a look at an example of an Argo CD `Application`:

```
apiVersion: argoproj.io/v1alpha1
"kind" on page xiii: Application
metadata:
name: guestbook
namespace: argocd
spec:
project: default
source:
repoURL: https://github.com/argoproj-labs/argocd-example-apps/
targetRevision: main
path: guestbook/
destination:
server: https://kubernetes.default.svc
namespace: example
```

This is a minimal example of what is needed for an Argo CD Application to be functional within a cluster. Things like adding Kustomize post rendering and sync options are also possible using the Application CRD. For the time being, the two main pieces of information that you should focus on are the `.spec.source` and `.spec.destination` sections.

`.spec.source`
: Defines the location containing resources that Argo CD should interface with. This property includes key options, such as `repoURL`, which defines where the Git repository or a Helm Chart Repository that holds the manifests are located. The `targetRevision`, where you can define what branch or tag should be targeted; or in the case of a Helm Chart, the version of the Helm Chart you want deployed. And finally, the `path`; which is where you can find the Kubernetes manifests relative to the `repoURL`.

`.spec.destination`
: This specifies the target Kubernetes cluster to apply the manifests defined under the `.spec.source` section. Here you'll specify the Kubernetes API endpoint in the `server` section (here `https://kubernetes.default.svc` is used as a way to indicate to Argo CD to deploy to the cluster Argo CD is running on), and the `namespace` section indicates which namespace to target on that cluster. Omitting the `namespace` section will cause Argo CD to default to the `default` namespace during deployment.

NOTE: The `namespace` in the `Application` YAML should match the `namespace` of where your Argo CD instance is installed - this is typically the default `argocd` namespace, and we show this in our example. Starting in Argo CD version 2.5, support was added to enable sourcing Applications from namespaces other than where Argo CD is deployed. However, this feature will not be used in the examples found within this publication.

The Argo CD Project Git repository provides a comprehensive view of all of the options available to you.

Other sections of note which we will not cover in detail within this chapter, but in subsequent chapters, are the options for when Applications are synchronized against cluster(s) within the `syncPolicy` property as well as how differences between the expected rendered state of resources and the actual state within a cluster are handled. Understanding and managing resources beyond their initial creation is related to a concept called drift management and is one of the key benefits provided by Argo CD.

Application Sources

Argo CD takes the desired state defined in the `Application` CRD, and attempts to modify the current running state on the Kubernetes cluster based on the defined content. Argo CD was built from the ground up with GitOps in mind, and it therefore supports two sources as the source of truth: Git and Helm.

The `source` field in an Argo CD Application has a 1:1 relationship with the Application specification. In other words, only one source can be configured per Application.

Starting with Argo CD v2.6, you can have a sources field now and specify more than one source. An example of a multi source Application is as follows:

```
spec:
sources:
- repoURL: https://github.com/christianh814/gitops-examples
  path: applicationsets/rollingsync/apps/pricelist-config
  targetRevision: main
- chart: mysql
  repoURL: https://charts.bitnami.com/bitnami
  targetRevision: 9.2.0
  helm:
    releaseName: pricelist-db
    parameters:
      - name: serviceAccount.name
        value: "pricelist-db"
      - name: auth.database
        value: "pricelist"
      - name: auth.username
        value: "pricelist"
```

```
      - name: auth.password
        value: "pricelist"
      - name: secondary.replicaCount
- repoURL: https://github.com/christianh814/gitops-examples
  path: applicationsets/rollingsync/apps/pricelist-frontend
  targetRevision: main
```

NOTE: Using `sources` field will cause Argo CD to ignore the `source` field

Let's review the two source types of an Application.

Git

Using Git as a source is a natural starting point for Argo CD users as it's the focal point of where "GitOps" gets its name. Git is not only the de facto source control management system (SCM) for developers, but is also the place where site reliability engineers (SREs) and platform engineers store their Infrastructure-as-Code (IaC) configurations. Many users making the switch to Argo CD and/or GitOps find that they are storing a lot of things on Git already.

Storing resources in Git, as an Argo CD Application source, can be as simple as having raw YAML stored containing Kubernetes resources within a directory. However, it doesn't have to be raw YAML. The declarations can also be stored and managed via templating tools, such as Kustomize (covered later in this chapter) or Helm (covered next).

Helm

Helm has become the de facto package manager for Kubernetes applications and deployments. At its core, Helm includes a templating engine for use with Kubernetes manifests so that they are reusable, reproducible, and stable. Many organizations have adopted Helm and it's a natural choice for developers and system administrators alike because of its ease of use and flexibility.

Since many organizations have widely adopted Helm, it was a natural fit for Argo CD. Argo CD can use Helm by directly consuming the Helm Chart stored in a standard Helm repository, OCI registry, or embedded within a Git repository.

Destinations

In Argo CD, the "destination" refers to a Kubernetes cluster. This destination cluster can either be the cluster that is running Argo CD or another remote cluster (which can be thought of as "hub and spoke").

The destination cluster is noted under `.spec.destination` in the Argo CD Application manifest. Here is a snippet of the configuration:

```
spec:
destination:
server: https://kubernetes.default.svc
namespace: bgd
```

In this example, the `server` field is set to `https://kubernetes.default.svc` which refers to the cluster that Argo CD is running on. The `namespace` field indicates which namespace to target.

NOTE: The namespace field **DOES NOT** overwrite the .metadata.namespace field if they are declared within your manifests.

Clusters can be added either declaratively or via the `argocd` CLI resulting in a new Secret to be added to the namespace Argo CD is deployed within.

```
$ kubectl get secrets -n argocd -l argocd.argoproj.io/secret-type=cluster
NAME                                TYPE     DATA   AGE
cluster-192.168.1.254-1289728133   Opaque   3      31s
```

More information about adding and managing clusters within Argo Cd will be covered in depth in Chapter 7.

Tools

One of the main tenets of GitOps is that declarations/configurations must exist in an immutate format. In a Kubernetes environment, this means that YAML is stored inside a Git repository. After a while, those who are just starting out in their GitOps journey, ask themselves: "How do I declaratively describe my resources in Git without copying and pasting *the same* YAML all over the place?"

It might seem like you'll have to duplicate a lot of the same YAML after you take things like environments, clusters, regulatory restrictions, and anything else in your organization that might force you to create a lot of YAML with only slight variations

between files into consideration. After a while this simple YAML that should be applicable "anywhere"...all of sudden doesn't fit anywhere. Luckily, there are tools that

can help you mitigate the issue of having to copy and paste the same YAML all over the place while making only small modifications.

Helm

Helm has become the de facto package manager and delivery mechanism for applications and controllers alike in a Kubernetes based environment. If you've ever worked on a Kubernetes cluster previously, you have most likely used Helm at some point to deploy ISVs, stacks, or even deliver your own application by leveraging its automation benefits. Helm provides not only a method of packaging an application and parameterizing YAML manifests, but also a templating engine that can be used to deploy your application to different environments.

Helm consists of different parts, as shown in Figure 3-1. Charts are templatized versions of your application's YAML manifests that are parameterized so that you can inject values into the defined templates. Helm combines the templatized manifests with parameters, called Values, to produce the resources to apply to the Kubernetes cluster. The specific installation of a chart within a cluster is known as a "release". The endstate representation of the produced release manifests is stored as a secret within the installed namespace on the Kubernetes cluster.

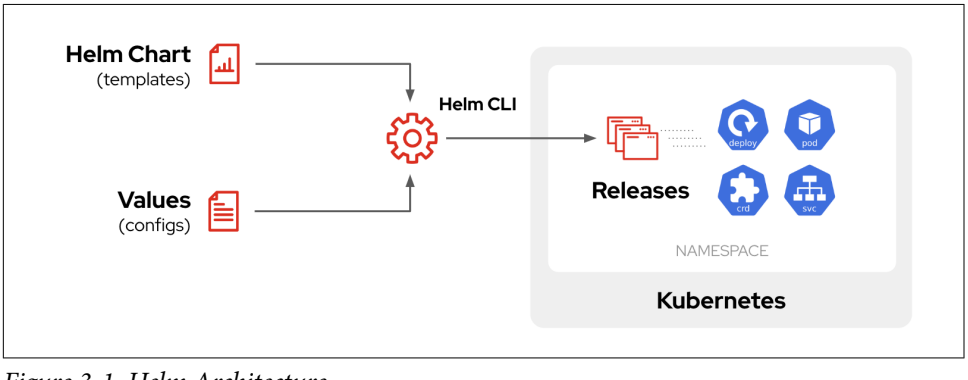


Figure 3-1. Helm Architecture

Secrets are the default backend storage for helm 3. Other backends, such as ConfigMaps or a relational database. Consult the Helm documentation for more information.

Helm has a large ecosystem and many repositories that end users can draw on to deploy pre-built applications.

If your organization uses Helm heavily, you're in luck! Most GitOps tools support deploying Helm charts.

Kustomize

Kustomize is a framework built within the Kubernetes community that lets you patch Kubernetes manifests without needing to modify the original Kubernetes manifests. While patching can be done via JSON patches, the manifest that it modifies needs to be in YAML.

Kustomize is hierarchically organized using a directory structure based on a concept of "bases" and "overlays". While each of these directories have their own purpose within Kustomize, they must contain a kustomization file (`kustomization.yaml`) which defines how to process the contents within the current directory along with importing content from other relative or remote sources. An "overlay" directory will include a reference to one or more bases, but a base directory can be included in one or more "overlays".

The following is a simple example of a `kustomization.yaml`

```
apiVersion: kustomize.config.k8s.io/v1beta1
"kind" on page xiii: Kustomization

namePrefix: kustomize-

resources:
- guestbook-ui-deployment.yaml
- guestbook-ui-svc.yaml
```

When using Kustomize against the prior example, two Kubernetes resources will be produced with their names prefixed with "kustomize-" as defined in the `namePrefix` property.

Kustomize is a powerful tool, and since it is built within the Kubernetes community, support is available within the `kubectl` CLI. Adding the `-k` flag when using `kubectl create` and `kubectl apply` commands will activate Kustomize processing. However, by using the `kubectl kustomize` subcommand instead, the full featureset of the kustomize CLI can be achieved.

Kustomize truly is a powerful tool because it eliminates the duplication of YAML and enables the ability to reuse by providing a method of patching the YAML to fit the need of the deployment. This means that you can store differences (for example between environments) as deltas instead of copying the YAML for each use. The Kustomize structure provides flexibility by creating "overlays" that can leverage other bases and other overlays, almost creating a cascading sequence of files. Those overlays can refer to remote repositories as well. Kustomize can even process Helm charts which can be achieved within Argo CD. We'll cover this in Chapter X.

Beyond Helm and Kustomize

While Helm and Kustomize are the two primary tools that are used in Argo CD to render resources within a Kubernetes cluster (aside from raw YAML), other tools are also supported. Argo CD natively supports the JSON templating language Jsonnet, and will process any file containing the *.jsonnet extension. Non-native tooling can also be included through the use of a Config Management Plugin, eliminating restrictions to customizing how Kubernetes resources are produced. More information on Config Management Plugins and their use can be found in Chapter 11.

Deploying Your First Application

Now that you're familiar with what an Argo CD Application is and its basic functionality, it's time to deploy your first Argo CD Application! Yes, we did walk through deploying an Application back in Chapter 1 when Argo CD was first installed, but by now, you have a better understanding of the purpose of an Application and how they can be used. Throughout the rest of this book, you'll be exploring many ways of deploying an Application, but for this example, we'll be going with the simplest way possible: Deploying from a Helm Chart.

For this example, create the the following Argo CD Application YAML in a file called `quarkus-app.yaml`:

```
apiVersion: argoproj.io/v1alpha1
"kind" on page xiii: Application
metadata:
name: quarkus-app
namespace: argocd
spec:
project: default
destination:
namespace: demo
name: in-cluster
source:
helm:
  parameters:
    - name: build.enabled
      value: "false"
    - name: deploy.route.enabled
      value: "false"
    - name: image.name
      value: quay.io/ablock/gitops-helm-quarkus
chart: quarkus
repoURL: https://redhat-developer.github.io/redhat-helm-charts
targetRevision: 0.0.3
syncPolicy:
automated:
  prune: true
  selfHeal: true
```

```
syncOptions:
- CreateNamespace=true
```

Here, we are going to define the name of the Application to be `quarkus-app` and we will be deploying the Application to the same cluster as Argo CD is running (denoted by `in-cluster` in the `.spec.destination.name` field). We are targeting the `demo` namespace on the destination cluster (i.e. which namespace to deploy the manifests to)

Note: The keyword `in-cluster` is a special keyword that means "target the cluster that the instance of Argo CD is running in"

We are deploying the `quarkus` chart version `0.0.3` from the repo denoted in the `repoURL` field. We are also providing any parameters in the `.spec.source.helm.parameters` field which represent Helm values being set against the chart. Also take note: we are adding the `CreateNamespace=true` option in the `syncOptions` field (in order to make sure the namespace exists before deploying the manifests). This example deployment of a Helm Chart using an Argo CD Application is analogous to the following command:

```
$ helm install quarkus-app --namespace demo --create-namespace --version 0.0.3 \
--set build.enabled=false \
--set deploy.route.enabled=false \
--set image.name=quay.io/ablock/gitops-helm-quarkus \
redhat-helm-charts/quarkus
```

To make use of the sample `helm install` command from above, the Helm Chart repository containing the quarkus chart must be added to the local machine using the `helm repo add https://redhat-developer.github.io/redhat-helm-charts` command. If the chart is installed using the Helm CLI, be sure that it is uninstalled prior to defining the chart using Argo CD. Otherwise errors will be produced.

To create this Argo CD Application within the Kubernetes cluster, you can apply it using the following `kubectl` command:

```
kubectl apply -f quarkus-app.yaml
```

You should see the Application appear in the Argo CD UI, as shown in Figure 3-3.

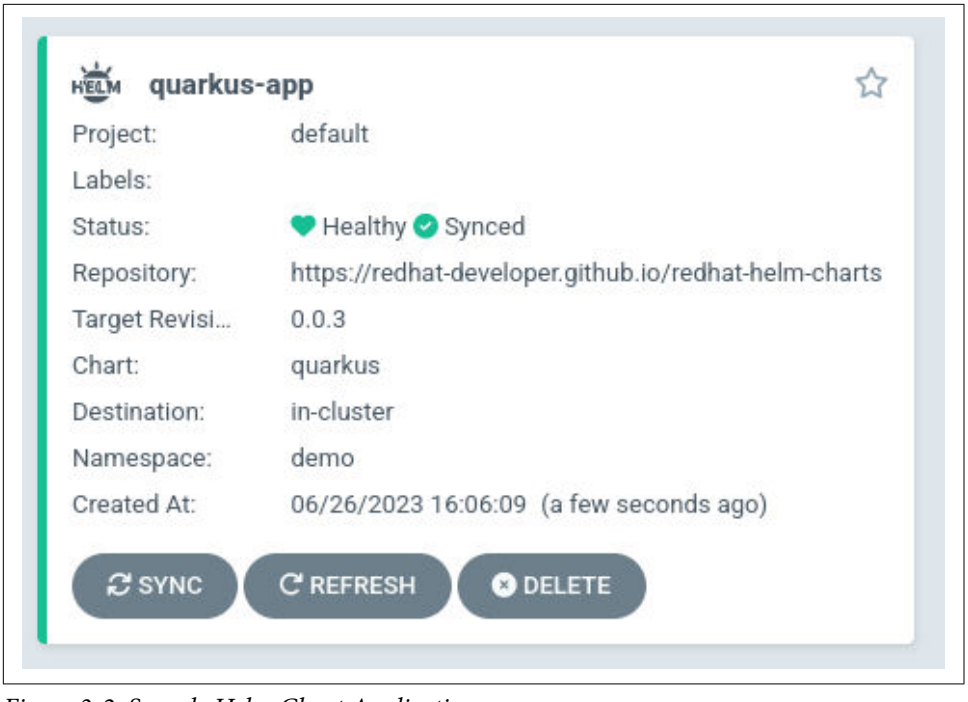


Figure 3-2. Sample Helm Chart Application

NOTE: See Chapter 1 - Installing Argo CD and Chapter 2 - Interacting with Argo CD for more information about connecting to the Argo CD UI

You should also be able to see the manifests deployed on the cluster using the Kubernetes CLI client. For example:

```
$ kubectl get deploy,service,pods -n demo
NAME                           READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/quarkus-app    1/1     1            1           9m24s

NAME                   TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
service/quarkus-app    ClusterIP   10.106.53.207   <none>        8080/TCP   9m24s

NAME                                 READY   STATUS    RESTARTS   AGE
pod/quarkus-app-57cf4d4b5c-q5jb8     1/1     Running   0          9m24s
```

One very important thing to note is the behavior in comparison to using the Helm CLI directly. Running `helm ls –n demo` against the namespace that contains an Argo CD managed Helm chart will not return any results:

```
$ helm ls -n demo
NAME    NAMESPACE       REVISION        UPDATED STATUS  CHART   APP VERSION
```

You may see other Helm releases running, but you won't see anything deployed via Argo CD

Why? Argo CD takes the philosophical approach of only working with "RAW Kubernetes manifests" directly. This means that Argo CD wants to "own" the manifests and not have to rely on trying to interface with another tool. Argo CD achieves this by doing the equivalent of running: `helm template <options> | kubectl apply -f -`

A release is only created whenever the `install` or `upgrade` subcommands of the Helm CLI is used which explains why a release is not present for Helm charts maintained by Argo CD.

Deleting Applications

Regardless of the tool being used to produce Kubernetes resources or the destination where these resources will be created, there may be a need to remove the Application so that the generated resources are no longer managed by Argo CD. Deleting an Application, similar to creating an Application, can be facilitated by using either of the methods that we have touched upon previously. Since we have used `kubectl` commands to create the Helm based Application in the prior section, we can use the same approach to delete the Application.

Execute the following command to delete the Application:

```
kubectl delete application quarkus-app -n argocd
```

Once the Application has been deleted, the tile representing the Application will no longer be present in the Argo CD interface. You should see something like in Figure 3-3.

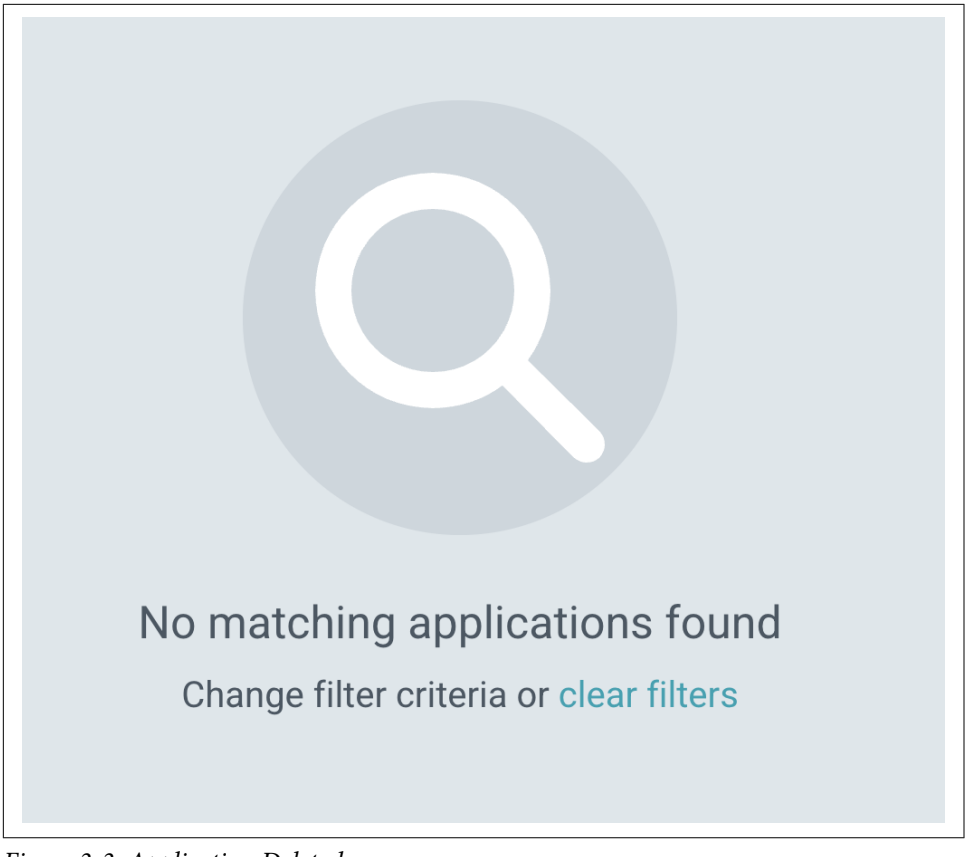


Figure 3-3. Application Deleted

It is important to note, and you may have discovered this already, is that even though the Application was deleted, the resources that were managed by the Application still remain within the `argocd` namespace.

Why is that the case you may wonder?

Argo CD makes the assumption that even though there is no longer a desire to manage these sets of resources, there will still be a need for them to remain within the cluster after the Application is deleted. This is mainly due to the motivation of avoiding data loss of the resources and so that anything dependent on them remains available. This approach is similar to how `PersistentVolumes` are managed within `StatefulSets` upon the removal of the `StatefulSet` itself or one of the replicas.

To remove the resources that are managed by an Application, the `resources-finalizer.argocd.argoproj.io` finalizer can be set on the Application as shown below:

```
apiVersion: argoproj.io/v1alpha1
"kind" on page xiii: Application
metadata:
name: quarkus-app
namespace: argocd
finalizers:
- resources-finalizer.argocd.argoproj.io
```

If an Application has this finalizer present, upon deletion, the Argo CD controller will perform a cascading deletion of all of the resources that it is managing.

Finalizers

Finalizers are a feature of Kubernetes associated with Garbage Collection that controls when a resource is deleted. When a resource is deleted, the `.metadata.deletionTimestamp` field is populated which triggers controllers to clean up any resource that is owned by the resource being deleted. Once the cleanup process completes, the associated controller will remove the finalizer from the resource. Only when all finalizers have been removed will the resource itself be deleted.

When deleting dependent resources, Argo CD makes use of the *foreground* cascade deletion policy which will delete the dependent resources first and then delete the Application afterward. If there is a desire to use the *background* cascade deletion policy which will delete the Application immediately while the controller deletes the associated resources, the `resources-finalizer.argocd.argoproj.io/background` finalizer can be set on the Application as shown below:

```
apiVersion: argoproj.io/v1alpha1
"kind" on page xiii: Application
metadata:
name: quarkus-app
namespace: argocd
finalizers:
- resources-finalizer.argocd.argoproj.io
   - resources-finalizer.argocd.argoproj.io/background
```

Summary

This chapter focused on managing Kubernetes resources through Argo CD Applications, which are Custom Resource Definitions (CRDs) representing a logical collection of related resources. These Applications are the smallest unit of work in Argo CD, defining the desired state of resources within a Kubernetes cluster. This chapter covers the essential components of an Argo CD Application, including the source and destination specifications, and introduces the templating tools supported by Argo CD, such as Helm and Kustomize.

An Argo CD Application specifies the source of resource manifests, typically located in a Git repository or Helm Chart repository. With the introduction of multi-source applications in Argo CD version 2.6, users can now specify multiple sources for a single application. Templating tools like Helm and Kustomize help manage and deploy Kubernetes manifests efficiently, avoiding redundancy and facilitating modifications.

This chapter also provided a guide for deploying applications using Argo CD, focusing on Helm Chart deployments. It explains the synchronization policies and the importance of finalizers in managing application deletions. Finalizers ensure that dependent resources are cleaned up properly, preventing data loss and maintaining resource availability. This chapter emphasizes Argo CD's approach to managing raw Kubernetes manifests, ensuring consistency and control over the deployment process.

CHAPTER 4

Synchronizing Applications

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *jleonard@oreilly.com*.

Argo CD makes it easy to be able to take Kubernetes resources stored within Git or Helm repositories and apply them to a target cluster – a process known as synchronization. Given that this capability is one of the core features of Argo CD, there are a variety of options available for determining when the synchronization process will be triggered and how the Kubernetes resources will be applied. This level of control is important as there may be a need to guard exactly how and when content is applied. In this chapter, we will explore the options available when synchronizing Argo CD Applications, their impact against the lifecycle of the Application itself, the Argo CD Server, and ultimately the target Kubernetes cluster.

Managing how Applications are Synchronized

Given that the synchronization of content from source to target Kubernetes cluster is a fundamental concept in Argo CD, it is important to first understand the defaults that Argo CD applies and the various levels of customizations that are available. If you recall back to Chapter 4 - Managing Applications, synchronization was one of the

topics that were briefly introduced including how the configurations can be defined within the `.spec.syncPolicy` property of an Application.

By default, when Applications are created, none of the rendered resources are applied to the Kubernetes cluster. This may surprise many new Argo CD users given that Argo CD is a tool that manages assets that are destined for Kubernetes. However, there are a number of reasons why this is Argo CD's default behavior:

- As the configurations for an Application are refined, there may be a need or desire to "preview" the changes that would be applied without performing any change
- A desire for control of when and how resource are applied
- Organizational policies prohibit automating changes to infrastructure

The Argo CD user interface and Application resource do provide a glimpse of the resources that would be affected, but any synchronization against the cluster would need to be performed in a manual fashion. Synchronization of manifests can be achieved using the user interface by selecting the "Sync" button on the Application or from the Argo CD CLI using the `argocd app sync` command.

But, since most users would want to take advantage of an automated synchronization of an Application, let's illustrate the ways that this can be achieved:

1. Specifying the sync policy for the Application using the `argocd` CLI.

```
$ argocd app set <APPNAME> --sync-policy automated
```

2. Selecting the "Enable Auto-Sync" button within the Argo CD user interface
3. Defining the configuration explicitly within the Application resource

```
spec:
  syncPolicy:
    automated: {}
```

Regardless of the option chosen, as soon as the source content differs from the live state of the cluster, the Application will be synchronized.

Sync options

Aside from the fundamental determination of whether an Application should be synchronized automatically or manually, Argo CD can be configured to perform a customized operation of how it synchronizes the desired state to the target cluster through the `.spec.syncPolicy.syncOptions` property. These customizations can, for the most part, be configured on the Application resource itself. However, others can be defined as annotations within each individual resource that is associated with an Application. This is especially useful when you want a specific action to occur

against a set of resources, but not all of the manifests associated within an Argo CD Application.

Let's first take a look at the how Sync Options can be used within an Argo CD application

Application Level Options

As mentioned previously, synchronization options are specified under the `.spec.syncPolicy.syncOptions` in the Application Manifest. These options will affect all resources that are associated with f the Argo CD Application.

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
    name: sample-app
    namespace: argocd
spec:
  syncPolicy:
    syncOptions:
      - Validate=true
      - ApplyOutOfSyncOnly=true
      - CreateNamespace=true
      - PrunePropagationPolicy=foreground
      - PruneLast=true
      - Replace=false
      - ServerSideApply=true
      - FailOnSharedResource=true
      - RespectIgnoreDifferences=true
```

Let's dive a little deeper into the `syncOptions` configurations:

`Validate=false`
: By default, Argo CD uses Kubernetes API validation and will fail the sync operation if the manifest is not valid (equivalent to running: `kubectl apply --validate=false`). The default value is: `true`

`ApplyOutOfSyncOnly=true`
: By default, Argo CD applies every object in an Argo CD Application. This could pose a problem if you have thousands and thousands of objects. This option only synchronizes/applies objects that are out of sync.

`CreateNamespace=true`
: This option creates the namespace (under in the `spec.destination.namespace` section of the Argo CD Application), if it does not already exist, before Argo CD attempts to apply the objects in an Application.

`PrunePropagationPolicy=foreground –`

This option shapes how the Application handles pruning/deleting of resources (known as garbage collection). The default is `foreground` and other options available are `background` and `orphan`. To learn more about how Kubernetes handles garbage collection, Read the upstream Kubernetes documentation.

`PruneLast=true`
: This option allows the ability for resource pruning to happen as a final part of a sync operation, after the other resources have been deployed and become healthy, and after all other waves are completed successfully.

`Replace=false`
: Argo CD, by default, does the equivalent of `kubectl apply`. This sometimes poses an issue when the object is too big to fit into `kubectl.kubernetes.io/last-applied-configuration` annotation. Note, this option could be dangerous, and be used with caution.

`ServerSideApply=true`
: This option enables Argo CD to use server side apply when running a sync operation. This is equivalent to running: kubectl `apply --server-side`. Most of the time, since this option is used to apply deltas of changes, the `Validate=false` option is frequently used in conjunction with this option.

`FailOnSharedResource=true`
: With this option, Argo CD will mark the Application as "failed" whenever it finds a resource associated with the Application that has already been applied in the cluster via another Application.

`RespectIgnoreDifferences=true`
: By default, Argo CD uses the `ignoreDifferences` config, found in `.spec.ignoreDifferences`, only for calculating the difference between the live and desired state (but still applies the object as it is defined" in Git). This option also takes it into consideration during the sync operation.

Resource Level Options

Along with the sync options on the Argo CD Application level, users can also apply these configurations/options at the object/individual resource level. Meaning that you don't have to apply any of the sync options against all resources contained within the entire Argo CD Application, but to only specific objects. A subset of the Application sync options are available to individual objects, as well as several other additional options.

These resource level options can be set by annotating the resource you want the option to apply to. You can do this by defining the `metadata.annotations.argocd.argoproj.io/sync-options` annotation on the resource you would

like to apply the option to. For example, to skip Kubernetes validation on a specific object:

```
metadata:
  annotations:
    argocd.argoproj.io/sync-options: Validate=false
```

By implementing this approach, only the object with this annotation will skip Kubernetes validation while the rest of the objects within the Argo CD Application will be validated. The options available via the `argocd.argoproj.io/sync-options` annotation are:

- `Validate`
- `PruneLast`
- `Replace`
- `ServerSideApply`

In addition, the following options are available for individual resources using the `argocd.argoproj.io/sync-options` annotation:

`Prune=false`
: This prevents the annotated object from being pruned.

`SkipDryRunOnMissingResource=true`
: Argo CD, by default, performs a "dry run" of applying the manifests (equivalent to using the `--dry-run` option with `kubectl`); this option skips the dry run step. This is especially useful if you are deploying CRDs or Operators as the associated resource may not be available as a registered resource at the specific validation time. This option is commonly paired with the retry strategy which will perform subsequent attempts to synchronize the Application where a failure no longer occurs as the desired resource has become available.

Users can specify multiple options in the annotation by separating the options with a comma (,) between each of the desired options. For example; to disable validation and use server side apply within a resource, you can set the following in your object:

```
metadata:
  annotations:
    argocd.argoproj.io/sync-options: Validate=false,ServerSideApply=true
```

Using the above configuration, the object with this annotation will disable validation *and* use server side apply.

Sync Order and Hooks

Argo CD has the ability to customize the order in which the manifests are applied. Furthermore, Argo CD incorporates different sync phases so that users can further fine tune how objects are applied to the target cluster.

Hooks

Argo CD has the ability to set up different sync phases by allowing the user to utilize "hooks". These injection points within the Application lifecycle enable additional automation, such as running scripts before, during, and/or after a sync has completed, to supplement applying the standard set of resources. You can also use hooks in the event a sync has failed for whatever reason. While hooks can be implemented as any Kubernetes object, they are usually Pods or Jobs.

There are 4 hooks that can be used in your Argo CD Sync process:

`PreSync`
: This phase occurs prior to the `Sync` phase. This is typically used for actions that need to occur before the Application is synced. A common use case is running a script that performs a schema update against a database.

`Sync`
: This is the "standard" or "default" phase for Argo CD and is executed once the `PreSync` phase has finished. This is typically used to aid the Argo CD Application deployment process in the event more complex activities within the Application needs to occur.

`PostSync`
: This phase occurs after the `Sync` phase has been completed. This can be used to send a notification that the phase has been completed or to trigger a CI progress or continue a CI/CD workflow.

`SyncFail`
: This is a special hook that is run only if a sync operation has failed. This is normally used for alerting or performing cleanup activities.

When setting up an Argo CD Application, the resources that are in your source of truth are applied to the destination cluster during the Sync phase. The other phases are used to perform pre or post tasks before and/or after the objects are applied in the `Sync` phase.

It is important to note that each phase is dependent on the success of the previous phase (with the exception of the `SyncFail` phase). For example: if an error occurs in the `PreSync` phase, the `Sync` phase will not run.

In order to indicate which resource in your Git repository belongs to which phase, you will have to annotate the desired resource with `argocd.argoproj.io/hook` with the value of the phase that it should execute within (the absence of the hook annotation results in the resource being applied during the `Sync` phase). For example, a Job to be executed in the `PostSync` phase, the following annotation is applied:

```
metadata:
  annotations:
    argocd.argoproj.io/hook: PostSync
```

Hook Deletion Policies

Resources that make use of Hooks can be deleted when a sync operation is performed by using the `argocd.argoproj.io/hook-delete-policy` annotation. The following hook deletion policies are available.

`HookSucceeded`
: The hook resource/object is deleted once it has successfully completed.

`HookFailed`
: The hook resource/object is deleted if the hook has failed.

`BeforeHookCreation`
: Any hook resource/object will be deleted before the new one is created.

Here is an example of a `PostSync` hook with a deletion policy of `HookSucceeded`.

```
metadata:
  annotations:
    argocd.argoproj.io/hook: PostSync
    argocd.argoproj.io/hook-delete-policy: HookSucceeded
```

It is important to note that hooks that are named (i.e. ones with `.metadata.name defined`) will be created/run only once. If you want a hook to be re-created or re-ran each time there is a sync operation; either use the `BeforeHookCreation` deletion policy or use `.metadata.generateName` in your resource/object.

As of the time of this writing, certain tools, such as Kustomize, have limited support for the use of the generateName property.

Sync waves

Argo CD applies manifests in a specific order. You can see this order by inspecting the code.In most cases, the default order that Argo CD applies resources should work

in most situations. However, complex deployments may inevitably require changes to this default order. This is where Syncwaves come in.

The concept of Syncwaves is pretty straightforward. The desired resource is annotated with the order in which you wish Ago CD to apply your manifests using `argocd.argoproj.io/sync-wave` key with an integer value denoted as a string.

```
metadata:
  annotations:
    argocd.argoproj.io/sync-wave: "5"
```

By default, every resource gets assigned "wave 0", unless otherwise specified via the annotation. Numbers can be negative as well. So, for example, consider the following:

- Namespace as wave "-1"
- Service Account as wave "0"
- Deployment as wave "1"

The Namespace would be applied first, then the Service Account, and then finally the Deployment.

A good use case for Syncwaves is to applyCustom Resource Definitions first before the corresponding Custom Resource.

Using Syncwaves within Hooks

Syncwaves can also be used within the confines of a Hook. Meaning you can have resources within a `PreSync` Hook Phase be applied in a specific order, within that order, without affecting other Hook Phases. In the following example, the Job will be applied in wave "3" within the `PreSync` Hook Phase.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: create-tables
  annotations:
    argocd.argoproj.io/sync-wave: "3"
    argocd.argoproj.io/hook: PreSync
```

Now, you can also have the following resource in a PostSync hook

```
apiVersion: batch/v1
kind: Job
metadata:
  name: test-deployment
metadata:
  annotations:
    argocd.argoproj.io/sync-wave: "1"
    argocd.argoproj.io/hook: PostSync
```

In these two examples, the `create-tables` will be applied before the `test-deployment` even though `test-deployment` is a lower "wave. This is due to the fact that the `create-tables` resource is in a different Hook Phase. The important thing to note when considering Syncwaves with Hooks is that Syncwaves are scooped within each Hook Phase.

Compare options

There might be cases where you will need to exclude resources from the overall status of your application. For example, if you have a resource created by another controller (this is common when working with Operators. You can read more about Operators here). This can be achieved with the following annotation.

```
metadata:
  annotations:
    argocd.argoproj.io/compare-options: IgnoreExtraneous
```

This only affects the sync status. If the resource's health is degraded, then the Application will also be degraded.

For example, the following Secret instructs the OpenShift OAuth Operator to create another Secret for the OpenShift OAuth Controller to consume. By doing so, Argo CD will mark your Argo CD Application "Out of Sync". To work around this issue, use the aforementioned `argocd.argoproj.io/compare-options: IgnoreExtraneous` annotation.

```
apiVersion: v1
kind: Secret
type: Opaque
metadata:
  name: htpass-secret
  namespace: openshift-config
  annotations:
    argocd.argoproj.io/compare-options: IgnoreExtraneous
data:
  htpasswd: bm90VGhlRHJvaWRzWW91cmVMb29raW5nRm9y
```

This will mark your Application as "Healthy" in Argo CD, but it's important to note that it'll mark the created resource as "out of sync". However, the health is not affected.

You can also ignore certain aspects during diffing and syncing, this will be covered in the following section titled "Managing Resource Differences."

Managing Resource Differences

Argo CD allows you to manage how you handle differences from your source of truth and current state within Kubernetes by the way of ignoring differences. There are several locations where ignoring differences can be configured. This configuration can be applied on a per Argo CD Application basis or for the whole Argo CD system (where all the Applications in an Argo CD installation are affected)

Application Level Diffing

As the name suggests, Application Level Diffing allows you to ignore differences within individual Applications at a specific JSON path, using RFC6902 JSON patches and JQ path expressions. Using the JSON path, you can specify paths referencing properties Argo CD should ignore when it compares the running state with the desired state defined. Here is an example:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: myapp
spec:
  ignoreDifferences:
  - group: apps
    kind: Deployment
    jsonPointers:
    - /spec/replicas
```

The `ignoreDifferences` setting allows you to specify the name of the resource and the namespace as well as the GVK (Group Version Kind). For more complex manifests, you can use the JQ path expression to define specific items to ignore in a more granular fashion. For example:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: myapp
spec:
  ignoreDifferences:
  - group: apps
    kind: Deployment
    jqPathExpressions:
    - .spec.template.spec.initContainers[] | select(.name == "injected-init-
container")
```

You can also ignore fields owned by specific managers by using `managedFieldsManagers` and listing the specific managers to ignore.

An additional item to note.; Most users will use the `RespectIgnoreDifferences` sync option in conjunction with this `ignoreDifferences` setting..

System Level Diffing

Argo CD can also be set up to ignore differences at a system level. This allows administrators to be able to set global ignore settings for the specific Argo CD installation. These configurations can be set up for a specified group and kind by using the `resource.customizations` key of `argocd-cm ConfigMap` using the following format.

```
data:
  resource.customizations.ignoreDifferences.apps_Deployment: |
   jsonPointers:
    - /spec/replicas
```

Take note the `resource.customizations` key also includes the keyword `ignoreDifferences` with the GKV demarcated by an underscore (_) using a flattened approach. For more information about how to formulate these settings please see the official Argo CD documentation site on system level diffing.

Use Case: Database Schema Setup

With an understanding of some of the ways to customize the synchronization and the associating current state for applications, let's see it in action with one of the most common use cases: A Database Schema Setup..

We are going to be deploying an Application that is going to consist of a backend database. The Database will be set up at deploy time, which means that the database schema will need to be loaded as a part of the deployment. Furthermore, the database schema setup needs to run after the database is up and running. For this specific use case we are going to be making use of SyncWaves and Argo CD Application SyncOptions.

Argo CD Application Overview

All the artifacts we will be using are in the aforementioned companion repo that you can find at https://github.com/sabre1041/argocd-up-and-running-book - make sure you've cloned this repository if you have not done so already and ensure that you are in the root directory of this repository.

Inspect the Argo CD Application for this use case which is located in the ch05 directory.

Execute cat `ch05/pricelist-app.yaml` from the root directory of the repository and you will see the following manifest:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: pricelist-app
  namespace: argocd
  finalizers:
    - resources-finalizer.argocd.argoproj.iospec:
  project: default
  source:
    path: ch05/manifests/
    repoURL: https://github.com/sabre1041/argocd-up-and-running-book
    targetRevision: main
  destination:
    namespace: pricelist
    name: in-cluster
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
    syncOptions:
      - CreateNamespace=true
    retry:
      limit: 5
      backoff:
        duration: 5s
        factor: 2
        maxDuration: 3m
```

This manifest should look familiar if you have already completed Chapter 4 - Managing Applications. There are several items of note to point out:

- The `.spec.syncPolicy` has the automated options of `prue: true and selfHeal: true`. This means that Argo CD will synchronize this Application automatically whenever it's out of sync. In addition, it will also delete resources that it is not keeping track of.
- Under `.spec.syncPolicy`, the `CreateNamespace=true` option under `syncOptions` is also defined which specifies that Argo CD will create the destination Namespace if it doesn't already exist.
- Retries under that `.spec.syncPolicy.retry` property have also been defined.

One final item to note is that Argo CD will be deploying manifests under the ch05/manifests/ directory from the repository as denoted in the .spec.source.path section. This last item is what we will cover in the next section.

Manifest Syncwave Overview

If you take a look under the `ch05/manifests/` directory, you will see a `kustomization.yaml` file, which for the purposes of this example, aggregates the manifests that

need to be applied. It's a simple list; basically, it is the resources that we want applied to the cluster.

For more information about Kustomize, please see Chapter 4 - Managing Applications

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
namespace: pricelist
resources:
- pricelist-db-pvc.yaml
- pricelist-db-svc.yaml
- pricelist-db.yaml
- pricelist-deploy.yaml
- pricelist-job.yaml
- pricelist-svc.yaml
```

Normally, Argo CD would apply these manifests in the same order as the output of kustomize build in this directory. However, we've added a syncwave annotation to customize the order Argo CD should apply these manifests.

Prior to any other resource in this Application being applied, we want the database and any backend storage to be up and running first. Therefore, we've annotated the pricelist-db-pvc.yaml (Persistent Volume Claim for the Database) and pricelist-db.yaml (Database Deployment) manifests with the argocd.argoproj.io/sync-wave: "1" annotation to denote that we want these two manifests to be applied first. They both should have the following annotation.

```
metadata:
  annotations:
    argocd.argoproj.io/sync-wave: "1"
```

This will not only make Argo CD apply these manifests first, but the annotation also causes Argo CD to wait until these manifests are in a "Ready" state before attempting to go on the next manifest. Once all the manifests in wave 1 are applied and reporting a Ready state, the next wave is applied.

In our use case, the next wave is the `pricelist-db-svc.yaml` file, which has the `argocd.argoproj.io/sync-wave:` "2" annotation:

```
apiVersion: v1
kind: Service
metadata:
  name: mysql
  annotations:
    argocd.argoproj.io/sync-wave: "2"
```

Since this is the only manifest with that sync wave annotation, this `pricelist-db-svc.yaml` file will be applied after wave 1.

You can inspect the other manifests in the ch05/manifests/ directory to inspect the order that they will be applied in.

- `pricelist-db-pvc.yaml and pricelist-db.yaml` as syncwave 1
- `pricelist-db-svc.yaml` as syncwave 2
- `pricelist-deploy.yaml` as syncwave 3
- `pricelist-svc.yaml` as syncwave 4
- `pricelist-job.yaml` in a `PostSync` hook in syncwave 0

Before moving on, it's important to note that when you inspect the `pricelist-job.yaml` manifest, this Job is responsible for setting up the Database schema. This Job also runs as a PostSync hook, which means that it will be applied after all the manifests in the Sync phase have been applied. Also note that the Job has a syncwave of 0. Although a syncwave of 0 is the default, the annotation was added to illustrate that syncwaves work within phases. Taking a look at the annotations in the `pricelist-job.yaml` manifest:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pricelist-postdeploy
  annotations:
    argocd.argoproj.io/sync-wave: "0"
    argocd.argoproj.io/hook: PostSync
    argocd.argoproj.io/hook-delete-policy: BeforeHookCreation
```

Another important item to note is the use of a hook deletion policy. This annotation ensures that this Job object should be deleted before the hook phase starts in subsequent sync runs if it is present. To learn more about hook deletion policies, please consult the official Argo CD documentation on resource hooks.

Importance of Probes

Argo CD uses several different sources to determine the overall health of the Application being deployed. One of the important metrics used is health status from the Kubernetes API. In order for this capability to be utilized, it's very important to have readiness/liveness probes set up correctly for each object that needs it.

For more information about how Argo CD handles Application health please consult the official documentation on Application health.

In our particular use case, the resources that require probes to be defined in order to achieve the desired goal are the Database Deployment and the web app Deployment. Taking a look at the `pricelist-db.yaml` file, you'll see the following probes.

```
spec:
  template:
    spec:
      containers:
      - image: registry.access.redhat.com/rhscl/mysql-56-rhel7:latest
        name: mysql
        livenessProbe:
          tcpSocket:
            port: 3306
          initialDelaySeconds: 12
          periodSeconds: 10
        readinessProbe:
          tcpSocket:
            port: 3306
          initialDelaySeconds: 12
          periodSeconds: 10
```

In this instance, TCP port 3306 is waiting to become active before considering the database deployment alive and ready to receive requests. For the web app, which is the `pricelist-deploy.yaml` file, you will see the following probes configured.

```
spec:
  template:
    spec:
      containers:
      - image: quay.io/redhatworkshops/pricelist:latest
        readinessProbe:
          httpGet:
            path: /
            port: 8080
          initialDelaySeconds: 5
          periodSeconds: 2
        livenessProbe:
          tcpSocket:
            port: 8080
          initialDelaySeconds: 5
          periodSeconds: 2
```

In the web app Deployment, we are considering the web app alive when TCP port 8080 is active. The app will not be considered ready until an HTTP GET request returns a response code of 200 on port 8080.

In both cases (the database Deployment and web app Deployment), both probes need to be successful before Argo CD considers the Application to be "healthy" and "synced".

For more information on probes and how to set them up, please see the official Kubernetes documentation on probes.

Seeing It In Action

Now that we've reviewed the use case in detail, let's see it in action by using these manifests in our KIND instance. From the root directory of the companion git repository, apply the Application manifest by running the following command:

```
kubectl apply -f ch05/pricelist-app.yaml
```

An Argo CD Application tile should appear in the Argo CD UI as a result. The tile will appear similar to what is depicted in Figure 4-1.

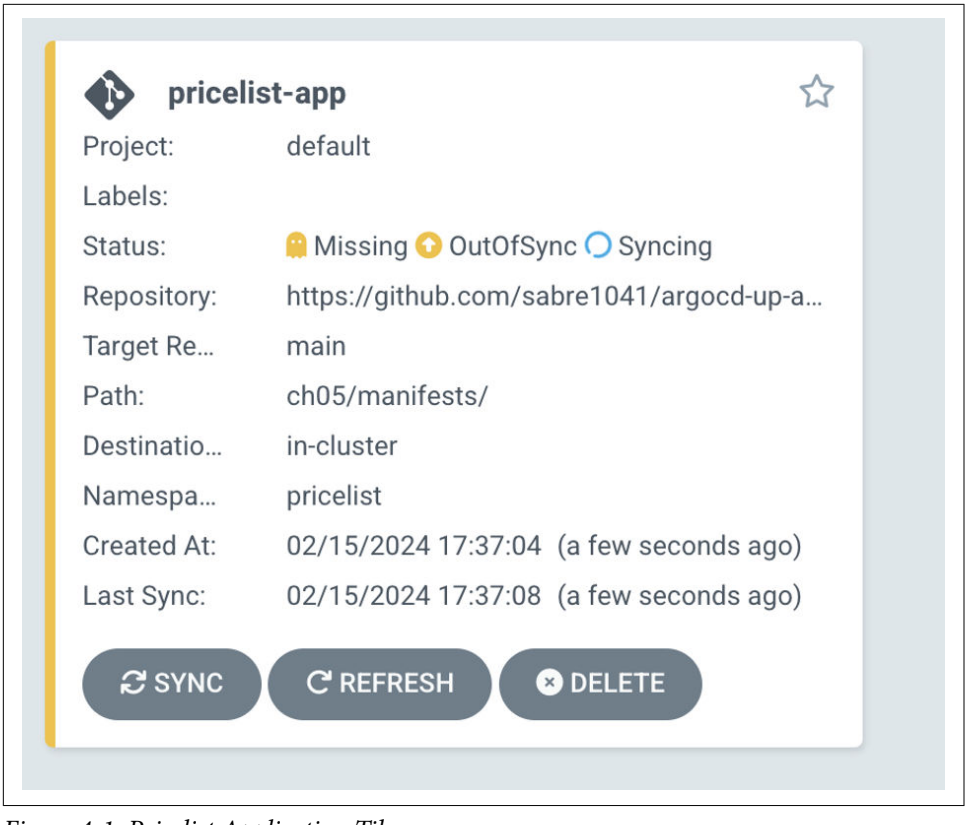


Figure 4-1. Pricelist Application Tile

The first thing Argo CD does is apply the first syncwave, which is our storage and database Deployment. After clicking on the Application tile, you should be able to see these resources enter the syncing phase first while the other resources are in the "missing" state. Take a look at Figure 4-2 for an example on how this is displayed.

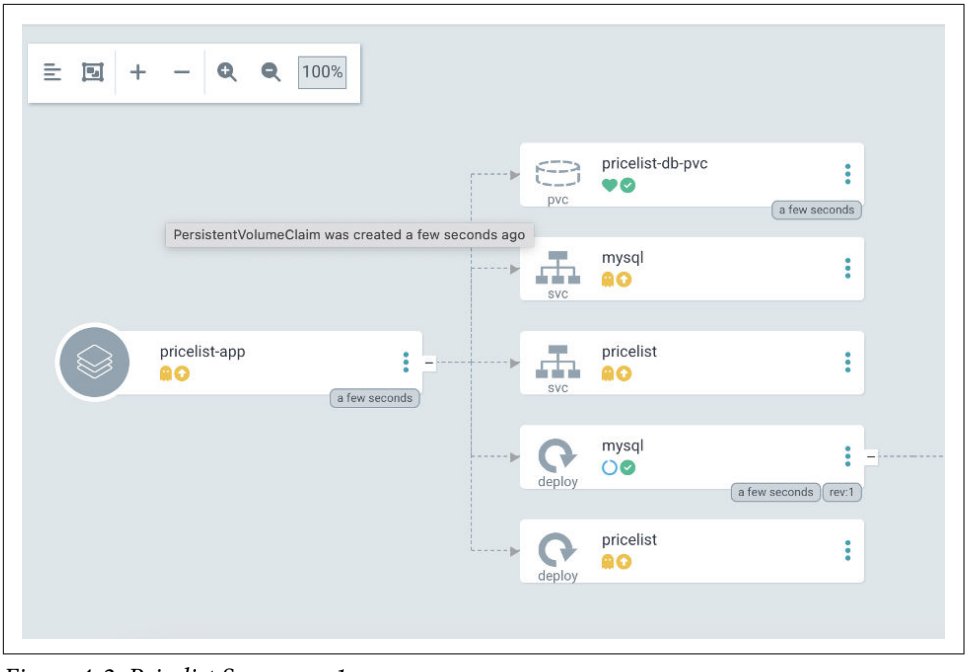


Figure 4-2. Pricelist Syncwave 1

When the storage is provisioned and the MySQL database is deployed, the next object that Argo CD will apply in our use case is the MySQL service. The Application overview will appear similar to Figure 4-3:

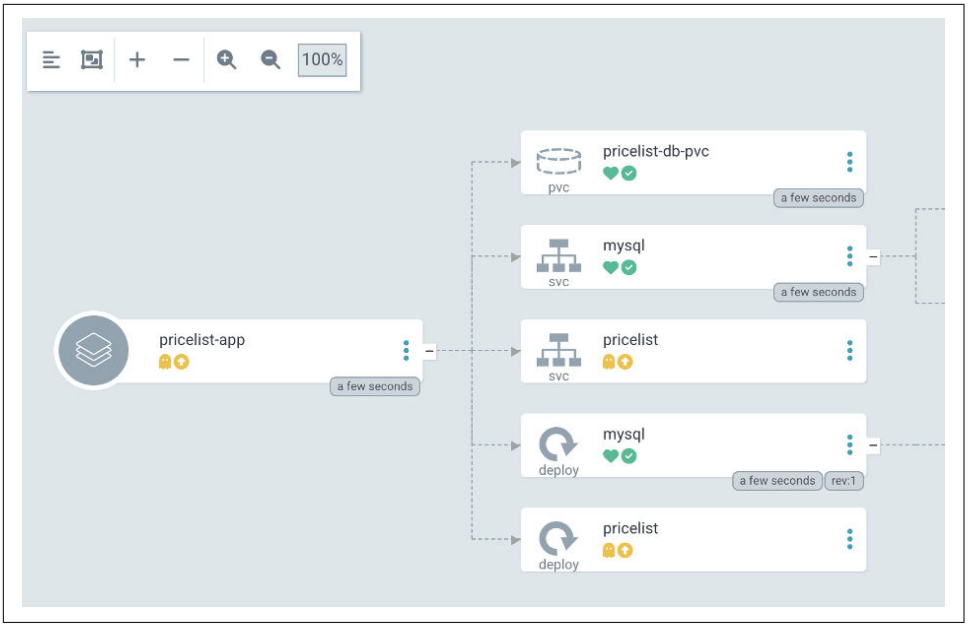


Figure 4-3. Pricelist Syncwave 2

After the service is healthy, Argo CD will apply the web app Deployment as seen in Figure 4-4:

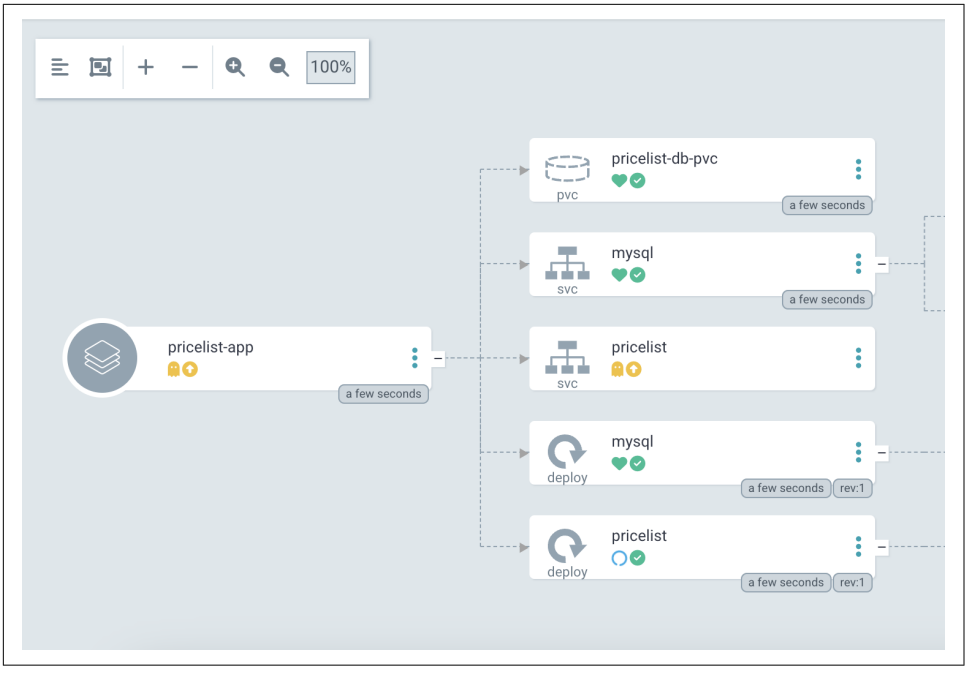


Figure 4-4. Pricelist Syncwave 3

Once the web app is deployed, the Service for the web app is applied as denoted in Figure 4-5:

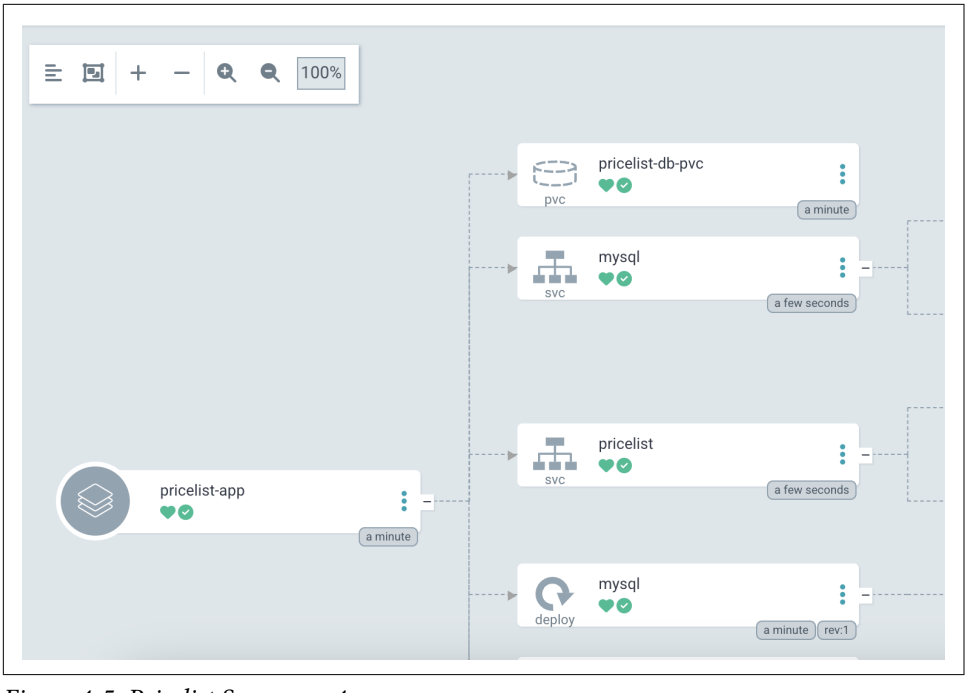


Figure 4-5. Pricelist Syncwave 4

Once the web app Service is deployed and in a healthy state; the Sync phase is considered complete and Argo CD will enter the `PostSync` phase The final step that Argo CD performs is applying the Job that facilitates the Database schema setup. In the Argo CD UI, this is indicated by an anchor (⚓) symbol within the Job, as seen in Figure 4-6:

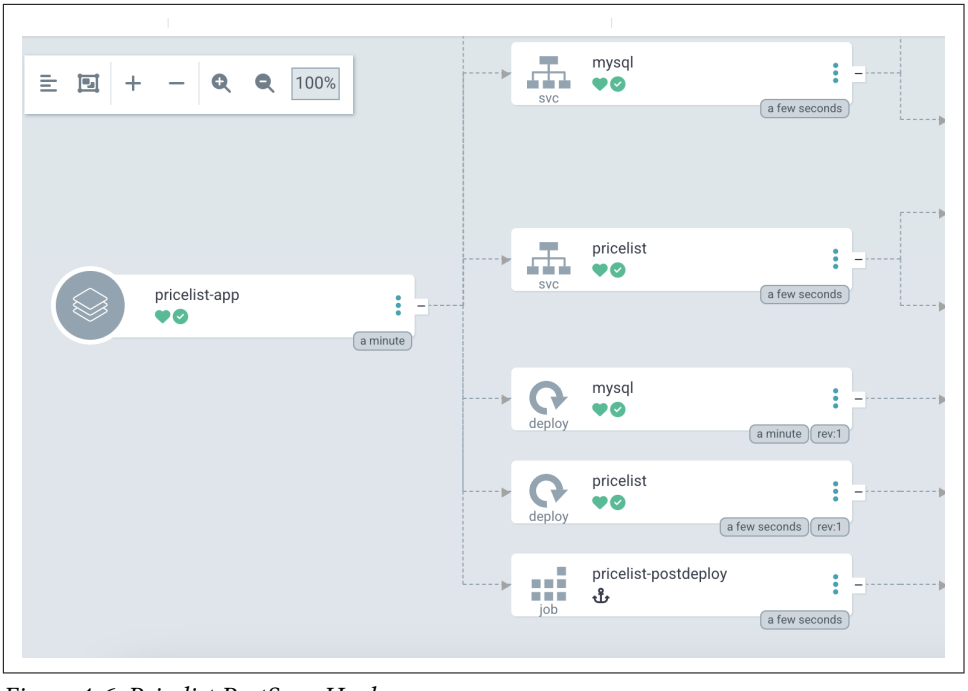


Figure 4-6. Pricelist PostSync Hook

Once the PostSync phase finishes, you should now see the Argo CD Application tile for the Application show "Healthy" and "Synced" status in the Application overview page. See Figure 4-7 for how this appears:

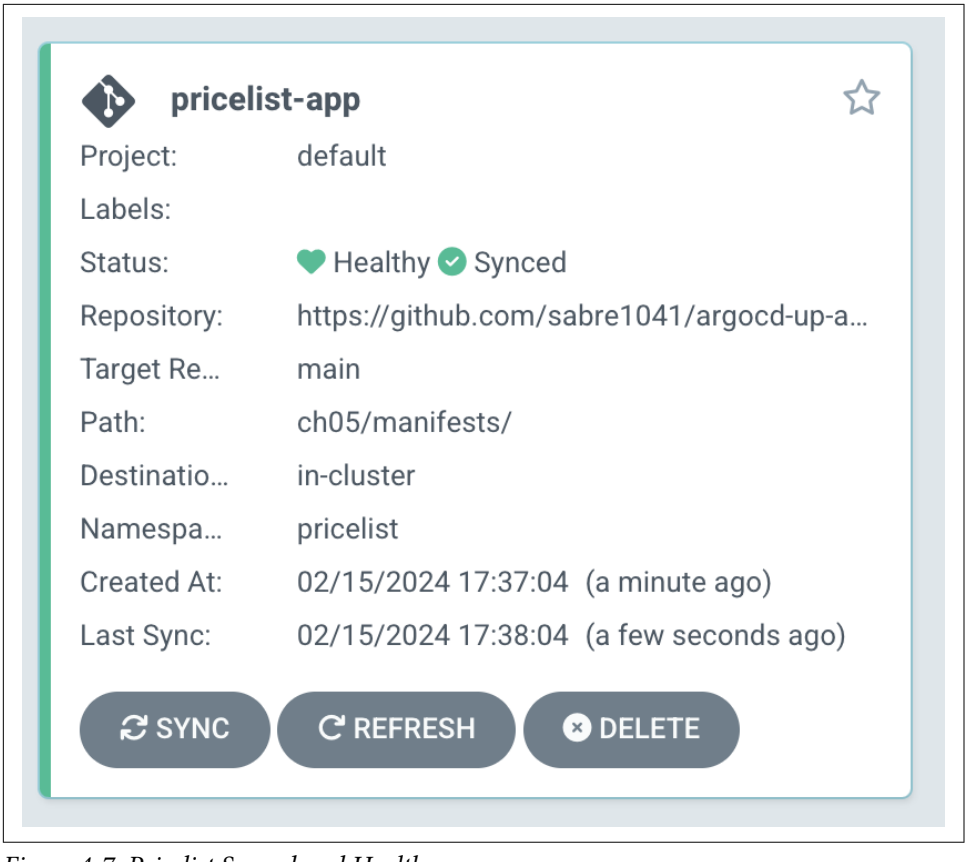


Figure 4-7. Pricelist Synced and Healthy

Summary

In this chapter, we covered how Argo CD synchronizes Applications and how you can customize the method in which Argo CD performs synchronizations on the individual Application level, and the system as a whole. We also reviewed how to further refine your synchronizations by implementing ordering with Syncwaves and Sync Hooks. Finally, we reviewed in detail a use case where Syncwaves and Sync Hooks were used to perform a database schema setup during an Argo CD deployment of an Application. In the next chapter, we will introduce how Argo CD handles Multi-tenancy and how to configure Argo CD Projects to provide those layers of demarcation.

CHAPTER 5
Authentication and Authorization

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *jleonard@oreilly.com*.

Included as part of the standard platform deployment, Argo CD contains a default management user providing unrestricted access to configure the platform using either the user interface or via the API/CLI. By providing this functionality out of the box, it simplifies the getting started experience and enables end users to realize the capabilities provided by Argo CD and the concepts embraced by GitOps methodologies.

As adoption grows beyond a single individual managing and utilizing Argo CD, there becomes a need to support additional users aside from a single elevated management user along with integrating with a centralized user management system, such as LDAP or a compatible OIDC provider. While at the same time, when providing the capability to support additional users, there must also be a way to define and govern the level of access that each entity is entitled to.

In this chapter, we will explore how users are managed in Argo CD including where and how they are defined, the ways that they can perform actions against the tool, as

well as the capabilities to define Role Based Access Control Policies to govern their access.

Managing Users

While Argo CD supports the ability to define and leverage multiple users, upon initial deployment, there is only a single user available for use – "admin". The admin user, as discussed previously, is provided both as a convenience for quickly getting up to speed with the capabilities provided by Argo CD, but also allowing unrestricted access to the entire set of features included by the tool. It can be used as the sole entity when Argo CD is utilized by a single individual, complement the incorporation of additional users once they are introduced, or be disabled entirely. Let's look into this admin user and how it can be leveraged at various phases in Argo CD, at initial deployment time and the use afterward.

The admin user

When Argo CD is first deployed, a secret named `argocd-initial-admin-secret` created within the namespace for which Argo CD has been deployed containing the password for the admin user. Assuming Argo CD has been deployed to the `argocd` namespace, password can be obtained by using the following command:

```
kubectl -n argocd get secret argocd-initial-admin-secret -o jsonpath="{.data.pass-
word}" | base64 -d
```

This method for obtaining the admin password was introduced in earlier chapters as we explored the various ways to interact with Argo CD. Let's now explore how we can manage the admin user in further detail.

Using the `argocd` CLI, login to Argo CD deployed to the `kind` cluster deployed earlier using the previous command to obtain the admin password.

```
argocd login --insecure --grpc-web --username admin --password=$(kubectl -n argocd
get secret argocd-initial-admin-secret -o jsonpath="{.data.password}" | base64 -d)
argocd.upandrunning.local
```

It is important that the kind cluster that is used for this chapter has an ingress controller deployed. Steps to enable the required kind cluster environment can be found in Chapter 3.

Details relating to the user can be found by using the `argocd account get-user-info`. Use this command to obtain information about the admin user:

```
argocd account get-user-info
Logged In: true
```

```
Username: admin
Issuer: argocd
Groups:
```

The prior output confirms that we successfully authenticated and have an active session as the admin user.

Changing the admin account password

As the name of the admin secret suggests, this password for the admin user should only be used for initial access and should be changed to prevent unwanted use given that anyone with the ability to read secrets in the Argo CD namespace can gain access to the password for a privileged user.

The `argocd account update-password` command can be used to change the password for a user. Update the password for the admin user replacing <new_password_value> with the desired password by executing the following command:

```
argocd account update-password --account=admin --current-password=$(kubectl -n
argocd get secret argocd-initial-admin-secret -o jsonpath="{.data.password}" |
base64 -d) --new-password=<new_password_value>
```

By default, the `argocd account update-`password command will update the account of the current user, and in this case, could have been omitted. However, the `--account` flag was included to explicitly select the user for which the password would be updated as well as to demonstrate how to target a different user, if desired.

With the account details updated, let's confirm the updated password works successfully by authenticating to the Argo CD web interface. Launch a browser and enter admin in the username field and the value of the updated password in the password field. If the credentials were accepted, you have successfully updated the admin password.

Now that the password for the admin user has been changed, the secret containing the initial password can be safely removed. Execute the following to delete the initial admin secret.

```
kubectl delete secret argocd-initial-admin-secret -n argocd
```

Local Users

To enable individuals the ability to access Argo CD without needing to use the admin user, Argo CD includes the functionality to manage users that are defined locally within the tool. Local Users serve two primary purposes:

- Provide a facility to generate authentication tokens for use by tools integrating to perform management functions. Examples include CI/CD and configuration management tooling

- Creation of additional users to support small teams or environments where integrating an external user management tool is not needed or desired.

Additional users are defined within the `argocd-cm` ConfigMap using the format `accounts.<username>` as the key along with one of the available capabilities that can be granted to a user.

The following is an example of how a new Local User named alice can be defined within the `argocd-cm` ConfigMap.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: argocd-cm
  namespace: argocd
  labels:
    app.kubernetes.io/name: argocd-cm
    app.kubernetes.io/part-of: argocd
data:
  accounts.alice: apiKey, login
```

Adding the alice user to the `argocd-cm` can also be achieved using kubectl by patching the `argocd-cm` ConfigMap using the following command:

```
kubectl patch -n argocd cm argocd-cm --type='merge' -p='{"data": {"accounts.alice":
"apiKey, login"}}'
```

Once the ConfigMap has been updated with the new user, their details can be displayed by using the argocd account list command:

```
NAME   ENABLED  CAPABILITIES
admin  true     login
alice  true     apiKey, login
```

At the present time, only two capabilities can be associated with a Local User: `login` and `apiKey`.

`login`
: Provides the ability to access the Web User Interface

`apiKey`
: Allows for authentication tokens to be generated to interact with the Argo CD API

For the majority use cases when creating local users, login is the only capability that will be needed as it is typically associated with a human actor. However, for the purposes of this exercise, we provided both available capabilities to the user alice.

Similar to the admin user, the first step that should be taken when creating new Local Users is to reset their password since no password is initially defined and they

would be unable to login. Use the `argocd account update-password` to update the password of the alice user as shown below. Replace `<new_password>` with the desired password that should be associated with the alice user.

```
argocd account update-password -—account=alice --new-password=<new_password>
```

You will then be prompted to enter the password of the current user, and once entered, the password will be changed.

Confirm that the new user alice can authenticate successfully by launching a web browser, navigating to the Argo CD Web Console, and logging in with the username alice and the password previously specified.

If the credentials are accepted, the new user has been created successfully and is ready for use.

Alternatively, instead of using the Argo CD CLI to update the password for a user, passwords can be defined in a declarative fashion by setting the `accounts.<username>.password` and `accounts.<username>.passwordMtime` properties within the `argocd-secret Secret. accounts.<username>.password` is a bcrypt hash containing the password while `accounts.<username>.passwordMtime` contains the date that the password was last modified.

The Argo CD CLI includes the `argocd account bcrypt` helper function for generating a bcrypt hash that can be used to specify the desired password for a user.

Generate a password that will be used for the alice user to authenticate using the following command:

```
argocd account bcrypt --password <new_password>
```

Update the password by patching the the `argocd-secret` with the value generated previously using the following command:

```
kubectl -n argocd patch secret argocd-secret \
  -p '{"stringData": {
    "accounts.alice.password": "<BCRYPT_PASSWORD>",
    "accounts.alice.passwordMtime": "'$(date -u +"%Y-%m-%dT%H:%M:%SZ")'"
  }}'
```

Once again, authenticate as alice using the newly updated password in the Argo CD UI.

Disabling Users

Once users have been created in Argo CD, their access can be disabled. By setting the `accounts.<username>.enabled` property to false in the `argocd-cm` ConfigMap, their access to both the UI and CLI can be disabled.

For example, to disable access for the user alice previously created, set `accounts.alice.enabled` to "`false`" in the `argocd-cm` ConfigMap as shown below.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: argocd-cm
  namespace: argocd
  labels:
    app.kubernetes.io/name: argocd-cm
    app.kubernetes.io/part-of: argocd
data:
  accounts.alice: apiKey, login
  # Disables Alice's Local User Account
  accounts.alice.enabled: "false"
```

The ConfigMap can also be updated directly using kubectl by performing the following command:

```
kubectl patch -n argocd cm argocd-cm --type='merge' -p='{"data":
{"accounts.alice.enabled": "false"}}'
```

Confirm the user alice can no longer access Argo CD by launching a web browser and attempting to authenticate using the alice user. You will be greeted with a message indicating the user account is disabled.

To reinstate the account, either set the accounts.alice.enabled field to true or remove the field entirely. The following is how to reenable the alice user account by removing the property patching the `argocd-cm ConfigMap` using kubectl.

```
kubectl patch -n argocd cm argocd-cm --type=json -p='[{"op": "remove", "path": "/
data/accounts.alice.enabled"}]'
```

Once the property has been removed from the `argocd-cm` ConfigMap, the `alice` user will be able to authenticate once again.

Disabling the admin user. In addition to being able to disable local user accounts, the Argo CD admin account can also be disabled. Once local users have been established and at least one of these accounts has been granted access to perform elevated actions, it is recommended that the admin account be disabled to enhance the overall security posture of Argo CD. Comparable to disabling a local user, the admin user can be disabled by setting the `admin.enabled` field to "`false`" in the `argocd-cm` ConfigMap.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: argocd-cm
  namespace: argocd
  labels:
    app.kubernetes.io/name: argocd-cm
```

```
    app.kubernetes.io/part-of: argocd
data:
  # Disables the admin account
  admin.enabled: "false"
```

This action can also be performed using kubectl by executing the following command:

```
kubectl patch -n argocd cm argocd-cm --type='merge' -p='{"data": {"admin.enabled":
"false"}}
```

Auth Tokens

Aside from being able to define additional users to access Argo CD, Local Users serve another function; the ability to define and generate authentication tokens which can be leveraged by external systems to perform automation actions. Examples of when auth tokens can be used include CI/CD tools to control and monitor the synchronization of Applications as part of a application release pipeline or within an automation tool to perform actions against Argo CD.

You may have noticed that when users are defined, they have the ability to have two associated capabilities. While we previously covered the use case for the `login` capability where a user is granted the ability to access the Argo CD web console. The `apiKey` capability allows for the generation of an auth token that is associated with their account.

Let's walk through how an auth token can be generated and used.

First, create a new Local User called "automation" which will be used the demonstrate how Auth Tokens can be created, managed and utilized, by patching the `argocd-cm` ConfigMap with the following command:

```
kubectl patch -n argocd cm argocd-cm --type='merge' -p='{"data": {"accounts.automa-
tion": "apiKey"}}'
```

Auth Tokens can be generated using the `argocd account generate-token` command. Individual user accounts for which the token will be generated can be targeted using the `--account` flag. Otherwise, a token will be generated for the currently logged in user.

To generate an Auth Token for the newly created automation user, execute the following command:

```
argocd account generate-token --account automation
```

An Auth Token consisting of a JSON Web Token (JWT) will be displayed as the output of the command. This token can be used to interact with the Argo CD CLI or API. Within the CLI, the `--auth-token` parameter can be used when invoking any command. So, to confirm that user backing the token is being honored by the

CLI, execute the following command which will display information about the user invoking the command:

```
argocd account get-user-info --auth-token=<token>
```

Replace the value of `<token>` with the token value generated previously. A response similar to the following should be displayed:

```
Logged In: true
Username: automation
Issuer: argocd
Groups:
```

Alternatively, the `ARGOCD_AUTH_TOKEN` environment variable allows for the auth token to be defined once and avoids needing to provide it as a parameter for each invocation of the CLI.

Specifying an auth token either through the flag or environment variable has a higher precedence than any other previously authenticated user.

Auth tokens, by default have no expiration which could be seen as a security risk. To increase the security posture surrounding auth tokens, it is recommended that they expire after a certain amount of time. The `--expires-in` flag can be used to specify a duration for which the token is valid (such as 1hr, 90d, etc).

Generate a time bound auth token of 90 days using the `--expires-in` parameter using the following command:

```
argocd account generate-token --account automation --expires-in 90d
```

Tokens associated with a user can be displayed using the `argocd account get` command. Display details about the automation user including the two tokens previously generated using the following command:

```
argocd account get --account automation
Name:               automation
Enabled:            true
Capabilities:       apiKey
Tokens:
ID                                    ISSUED AT                  EXPIRING AT
89ec94b0-aff8-47c6-b59a-229c4b564688  2024-01-20T14:15:16-06:00
2024-04-19T15:15:16-05:00
70cf36ea-b365-4f04-9fc0-56229cd41620  2024-01-20T12:39:16-06:00  never
```

Notice how the first token has an infinite lifespan whereas the second token will expire at a time relative to the time it was generated.

Tokens can be explicitly revoked whenever there is a desire to do so. Examples of when one might want to revoke an Auth Token is when it has been accidentally exposed or is being used by a member of the team who no longer requires

access. Tokens are tracked within the `argocd-secret` Secret in a key called `accounts.<account>.tokens` and while this property can be modified manually, it is much more straightforward to use the Argo CD CLI

Delete one of the tokens previously generated by using `argocd account delete-token` command and specifying the name of the account the token is associated with and the ID of the token. The ID of all auth tokens is shown when invoking the `argocd account get` command and is a uuid that is generated at token creation time. To use a more friendly name, use the `--id` flag of the `argocd account generate-token` command.

Revoke an auth token by executing the following command:

```
argocd account delete-token --account <account> <ID>
```

SSO

Local users are a great way to onboard a small team into Argo CD or leverage Auth Token to perform automation actions. As Argo CD adoption grows, especially within a large organization, the management of users within Argo CD through the use of the Local Users feature can become untenable. Fortunately, Argo CD has the capability to integrate with external user management tools to offload the capability to an external system.

Two forms of SSO are available:

- Dex OIDC Provider:
- Direct OIDC Integration

Either option uses OpenID Connect (OIDC) authentication protocol to facilitate how users authentication and their details are consumed by Argo CD.

Dex

Dex is an identity service which is bundled with Argo CD and runs as a separate pod that acts as a bridge between one or more identity providers through the use of connectors. These connectors provide advanced and provider specific functionality that maps user details into a format that Argo CD can understand in a standardized manner. Supported connectors include Git based services, like GitHub and GitLab, enterprise integrations from Google and Microsoft, and LDAP for integrating more traditional user management platforms. Multiple connectors can be specified to account for one or more identity services that contain users who would like to access and leverage Argo CD.

Direct OIDC

If the desired user management tool exposes an OIDC interface (for example Microsoft, Google, Keycloak), Argo CD can delegate the entire authentication process to the provider. By using this method, many of the same configurations that have been used previously when interacting with the OIDC provider by other tools can be reused for Argo CD enabling a more native and consistent method for accessing identity details from the provider.

SSO in Action

With an understanding of the options available when leveraging the SSO capability within Argo CD, let's look at the steps involved when implementing SSO in our `kind` cluster. While there are a variety of options available for integrating users that are managed externally, wwe will utilize Keycloak, an open source identity and management tool. Users externally, we will describe how to integrate users stored in Keycloak. Keycloak exposes a native OIDC compatible interface, which makes it ideal to demonstrate an SSO integration with both Dex and the native OIDC options.

The first step is to deploy Keycloak to your `kind` cluster. Tooling is available within the project Git repository to facilitate the deployment and configuration of Keycloak.

Ensure that you have the project codebase available and navigate to the `ch06` directory.

To simplify the deployment and configuration of Keycloak, the Keycloak Operator will be used. Execute the following script to deploy the operator to a new namespace called `keycloak`.

```
helm upgrade -i -n keycloak --create-namespace keycloak-operator charts/keycloak-
operator
```

Confirm the Keycloak Operator is running by listing the pods within the `keycloak` namespace

```
kubectl -n keycloak get pods
```

With the operator running, let's work on deploying Keycloak itself. While the operator provides the capabilities to manage the majority of concerns related to the deployment and configuration of Keycloak, it does not have the functionality to generate a SSL certificate to secure ingress communication.

Use the openssl tool to generate a self signed certificate for Keycloak and place the generated certificates within the files folder of the keycloak Helm chart. These files will be leveraged afterward when the chart is installed. Execute the following command to generate the certificate.

```
openssl req -subj "/CN=keycloak.upandrunning.local/O=O'Reilly Media/C=US" -newkey
rsa:2048 -nodes -keyout charts/keycloak/files/key.pem -x509 -days 365 -out charts/
keycloak/files/certificate.pem
```

As you might have noticed, the generated certificate uses the hostname `keycloak.upandrunning.local`. Using a similar process that was utilized in Chapter 3, add the following value to the `/etc/hosts` file so that requests are made against the KIND environment.

```
127.0.0.1 keycloak.upandrunning.local
```

Now, install the Helm chart to configure the supporting components including a PostgreSQL database backend, TLS certificates previously generated and the `Keycloak` Custom Resource. The Keycloak Operator in turn deploy and configure keycloak.

```
helm upgrade -i -n keycloak keycloak charts/keycloak
```

In a few moments, an Argo CD pod will be created. This can be seen by querying the pods in the keycloak namespace.

```
kubectl get pods -n keycloak
```

Once the pod is up, navigate to the Keycloak interface at https://keycloak.upandrunning.local. Accept the self signed certificate and you will be presented with a dashboard of options to choose from. Select **Administration Console**.

The password for the default administrator account is automatically created in a secret called `keycloak-initial-admin` and stored within the `keycloak` namespace by the Keycloak Operator.

Extract the value by executing the following command:

```
kubectl get secret -n keycloak keycloak-initial-admin -o jsonpath='{ .data.pass-
word }' | base64 -d
```

Use the retrieved password and login with the username admin. Once authenticated, you will be presented with the Keycloak dashboard within the argocd *realm*. A realm in keycloak is where you define and manage resources, including users, clients, and other entities.

master is the name of the default realm in Keycloak and to emphasize a separation of duties, another realm called argocd will be used to define the integration with Argo CD. The Helm chart that we installed previously created a new realm called *argocd* and populated the instance with a baseline set of resources for us to start. Let's explore the *argocd* realm to see what was created for us.

First, ensure that you are using the argocd realm. The active realm the UI is displaying is located on the upper left hand portion of the page. A dropdown of available

realms is also available if there is more than the default *master* realm. If the dropdown does not display argocd currently, go ahead and click the dropdown and select *argocd* so that you are focusing on the appropriate realm.

Two users were also created: john, who represents an Argo CD administrator and mary, a senior software developer. They can be seen by selecting the *Users* button on the left hand navigation pane.

Two Keycloak groups have also been defined. ArgoCDAdmins, which represent Argo CD administrators and Developers which represent members of the software development team. John is a member of the admins group and Mary is a member of the developers group. Group definition and the membership can be seen by selecting the Groups button on the left hand navigation pane.

Now, let's complete the necessary configuration to enable Argo CD to integrate with Keycloak. Create a new Keycloak client by selecting the **Clients** button on the left hand navigation pane and then selecting the **Create Client** button at the top.

Enter **argocd** as the Client ID and **Argo CD** as the Client Name and then click **Next**.

Enable Client Authentication by switching the toggle to the enabled position and leaving the remaining values in their default position. Then, click **Next**.

Set the **Root URL** and **Web Origins** to the URL of the Argo CD instance: *https://argocd.upandrunning.local*

Argo CD exposes callback URL's for requests to invoke once the authentication process is successful for each of the SSO types at the context paths */api/dex/callback* for Dex and */auth/callback* for direct OIDC. As a result, enter the following values in the **Valid Redirect URI's** field. Hit the *Add Valid Redirect URI's* link to add the second value.

- https://argocd.upandrunning.local/auth/callback
- https://argocd.upandrunning.local/api/dex/callback

We can then set the default page within the console that a user is directed to upon a successful authentication. Set the **Home URL** to **/Applications** so that they will be sent to the page displaying all of the applications they are allowed to view.

Finally, **enter https://argocd.upandrunning.local** into the textbox next to **Valid post logout redirect URIs**

Click **Save** to create the Keycloak client.

Since the Client Authentication option was selected, the OIDC Confidential Access Type was enabled. As a result, a set of credentials were generated so that Argo CD can use them to facilitate user authentication via a browser. Obtain the client secret by selecting the **Credentials** tab for the argocd client and select the copy button

to capture the value to the clipboard. Feel free to select the eyeball icon which will display the value.

To enable the groups that a user is a member of to be included as part of the JWT, create a new Client Scope by selecting Client Scope on the left hand navigation and then select **Create Client Scope**.

Enter *groups* as the name of the client scope and then click **Save**.

Click on the **mappers** tab to enable the groups claim to be added to the token. Select **Configure a new mapper** and then select **Group Membership**.

Enter *groups* for the **Name** and **Token Claim Name**. Deselect **Full Group Path** and leave the remaining options enabled. Click **Save** to apply the configuration.

Finally, add the new Client Scope to the argocd client by once again selecting Clients on the left hand menu and then argocd.

On the argocd client configuration page, select the **Client Scopes** tab and then select the **Add client scope** button.

Select the checkbox next to groups, select the **Add** button and then from the options provided, select **Default** so that the groups claim will always be included in the token without needing to be explicitly requested.

At this point, Keycloak has been configured to support the integration with Argo CD. Before we can focus on the Argo CD configuration itself, there needs to be an adjustment made within our `kind` cluster. Recall that we updated the `/etc/hosts` file on our machine with the URL's for both Argo CD and Keycloak so that they would resolve and route appropriately to our `kind` cluster.

Since Argo CD will need to access Keycloak to complete the authentication process, it too will need some assistance resolving the Keycloak server. `kind` makes use of CoreDNS for intra-cluster DNS resolution. We can perform a similar pattern where requests made against the Keycloak endpoint are rewritten to an internal Kubernetes service that was configured as part of the Keycloak Helm chart that was deployed earlier.

Edit the CoreDNS configuration file stored in coredns ConfigMap within the kube-system namespace.

```
kubectl edit cm coredns -n kube-system
```

Add the following bolded content to the configuration file which will add in the rewrite rule:

```
apiVersion: v1
kind: ConfigMap
data:
  Corefile: |
```

```
    .:53 {
        rewrite name keycloak.upandrunning.local. keycloak-internal.key-
cloak.svc.cluster.local.
        errors
        health {
           lameduck 5s
        }
…
```

Delete the CoreDNS pods so that the changes are picked up.

```
kubectl delete pod -n kube-system -l=k8s-app=kube-dns
```

Verify that applications running within the kind cluster can resolve Keycloak now that the rewrite rule was configured in CoreDNS.

```
kubectl exec -n ingress-nginx svc/ingress-nginx-controller -- curl -skLI https://
keycloak.upandrunning.local | head -1
```

If the response returned `HTTP/2 200`, DNS resolution is working correctly.

Regardless of the type of SSO backend Argo CD communicates with or the type of SSO integration that is selected, one property must be set within the `argocd-cm` ConfigMap: the URL of the Argo CD server as defined by the `url` key. Execute the follow command to patch the argocd-cm ConfigMap:

```
kubectl patch -n argocd cm argocd-cm -p '{"data":{"url": "https://argocd.upandrun-
ning.local"}}'
```

Now let's shift our attention to the necessary configurations within Argo CD.

The Client ID and Secret need to be included within the SSO configuration so that Argo CD can authenticate with Keycloak. Since the Client Secret is a sensitive asset, instead of explicitly specifying the value explicitly, it can be stored in a Secret and then referenced from the configuration file. Secrets can be referenced from two locations:

- The global `argocd-secret` Secret
- A separate secret within the namespace Argo CD is deployed within

To avoid mixing default Argo CD and user provided content, create a separate secret called `keycloak-secret` within the argo namespace and specify the Client ID and Client Secret from the argocd Client previously defined within using the following command:

```
kubectl create secret generic -n argocd keycloak-secret --from-literal=clientSe-
cret=<keycloak_argocd_clientSecret>
```

In order for Argo CD to make use of the secret for use, it must include the label `app.kubernetes.io/part-of: argocd`. Execute the following command to add the label to the `keycloak-secret` Secret.

```
kubectl label secret -n argocd keycloak-secret app.kubernetes.io/part-of=argocd
```

Sensitive data stored within secrets can then be referenced within Argo CD Configuration files. Values beginning with a $ look for keys within a secret matching the value. If the value takes the form `$<secret>:a.key.in.k8s.secret`, Argo CD will look for the value within the Secret `<secret>` and the key which follows the colon (:).

For example, if the following was declared within a ConfigMap:

```
myProperty: $foo:bar
```

The referenced sensitive value would be sourced from a Secret called `foo` and the key `bar`.

Alternatively, sensitive values can also be stored within the global `argocd-secret` Secret instead of a dedicated secret. The only difference when referencing the value within a configuration is that the name of the Secret that the content would be placed within and the colon (:) separator is omitted. So, when replicating the example above, the following would reference the `bar` key within the `argocd-secret` global Secret.

```
myProperty: $bar
```

With an understanding of how sensitive resources can be accessed, in the case of the Client Secret that was previously stored in the `keycloak-secret` Secret, the value can be referenced within Argo CD configurations using the form `$keycloak-secret:clientSecret`.

Now that we have the insights and the necessary supporting components to enable SSO in Argo CD complete, let's walk through how to configure Argo CD to leverage Keycloak using both Dex and Direct OIDC integrations.

Either option is enabled by updating the content of the `argocd-cm` ConfigMap. It is important to note that Dex and Direct OIDC integration can not be enabled at the same time.

SSO Using Dex. SSO for Argo CD using Dex can be enabled by specifying the `dex.config` property of the `argocd-cm` ConfigMap. This property is an inline representation of the standard Dex Configuration file that would be used in standalone deployments of Dex. Argo CD manages most of the boilerplate content and the end user is responsible for defining the connectors (strategy to authenticate against another identity provider) that will be leveraged. Since Dex does not contain a connector specifically engineered for Keycloak, we will leverage the generic OIDC connector.

Aside from the Client ID and Client Secret, the only other property that we will need to provide within Dex is the location of the OIDC issuer (The based URL for OIDC resources). This address can be accessed from the Realm Settings of the *argocd* realm in the Keycloak user interface.

Locate the **OpenID Endpoint Configuration** link under the *endpoints* section on the Realm Settings page. Clicking on this link brings up the OIDC discovery document which contains all of the OIDC metadata required to understand how to interact with this endpoint. Since the issuer URL is just the base URL, we can omit *.well-known/openid-configuration* leaving us with an issuer URL of *https://keycloak.upandrunning.local/realms/argocd*

Update the `argocd-cm` ConfigMap with the following content:

```
dex.config: |
  connectors:
    - type: oidc
      id: keycloak
      name: Keycloak Dex
      config:
        issuer: https://keycloak.upandrunning.local/realms/argocd
        clientID: argocd
        clientSecret: $keycloak-secret:clientSecret
        insecureSkipVerify: true
        insecureEnableGroups: true
```

The ConfigMap can be modified interactively by executing the following command:

```
kubectl edit cm -n argocd argocd-cm
```

Several items of note from the configurations from the `dex.config` property above:

- The `clientSecret` property is making use of the Keycloak Client Secret that was configured in the `keycloak-secret` Secret
- Since Keycloak uses a self-signed certificate to enable TLS communication, the `insecureSkipVerify` property ignores verification errors
- The `insecureEnableGroups` property allows Dex to process groups defined within Keycloak from the `groups` claim.

Once the configuration has been applied, launch the Argo CD user interface. If you were previously authenticated and still have an active session, go ahead and log out.

On the login page itself, notice how there is a new button titled “Log In Via Keycloak” in addition to the username and password option that was used previously. Click on the “Log In Via Keycloak” button and you will be transferred to the Keycloak instance in order to authenticate.

If the "Log In Via Keycloak" button does not appear (if it's not working), you may need to forcibly trigger a reload of the configuration by deleting all of the pods in the argo namespace using the command kubectl delete pods -n argocd --all

Recall two users were defined in Keycloak. Go ahead and authenticate as the Argo CD Administrator John using the username **john@upandrunning.local** and password **arogocdAdmin123**. Upon a successful authentication, you will be transferred back to the Argo CD instance, and as defined within Keycloak, the Applications page.

Select the **User Info** link on the left hand navigation pane to view details related to the current user. Notice how the username matches the user we authenticated as and the issuer matches the value we obtained from Keycloak and configured within the `dex.config` property. Most importantly, the list of groups that John is a member of is also displayed confirming that Dex was able to retrieve the values from the groups claim.

Now that we have validated SSO user authentication using Dex, let's see how we can enable Argo CD SSO integration to Keycloak using the direct OIDC approach.

SSO Using Direct OIDC. Configuring Argo CD to communicate directly with the OIDC provider offers greater simplicity as well as eliminates a component (Dex) from being deployed and managed. The process for enabling direct OIDC integration mirrors the steps as described in the previous section.

First, remove the `dex.config` property as both Dex and Direct OIDC integration cannot be enabled concurrently. Direct OIDC integration is defined within the oidc.config property within the `argocd-cm` ConfigMap. Specify the following contents to enable Direct OIDC integration with the Keycloak instance.

```
  oidc.config: |
    name: Keycloak
    issuer: https://keycloak.upandrunning.local/realms/argocd
    clientID: argocd
    clientSecret: $keycloak-secret:clientSecret
    logoutURL: "https://keycloak.upandrunning.local/realms/argocd/proto-
col/openid-connect/logout?client_id=argocd&id_token_hint={{token}}&post_logout_redi-
rect_uri={{logoutRedirectURL}}"
```

As you can see, the contents are almost identical. The final step is to configure Argo CD to ignore verification errors to the OIDC endpoint. Instead of this property being defined within the OIDC config, it is instead a top level property within the `argocd-cm` ConfigMap. Add the following to the `argocd-cm` ConfigMap to disable OIDC SSL verification:

```
oidc.tls.insecure.skip.verify: "true"
```

This property can also be set by executing the following command:

```
kubectl patch -n argocd cm argocd-cm --type='merge' -p='{"data": {"oidc.tls.inse-
cure.skip.verify": "true"}}'
```

With the configurations for direct OIDC integration in place, navigate to the Argo CD web console at https://argocd.upandrunning.local. You should be greeted once again with the option to authenticate using a local account or using Keycloak SSO. Login as the Argo CD Administrator John using the username **john@upandrunning.local** and password **arogocdAdmin123**. Select the **User Info** link on the left hand navigation pane and confirm all of the properties align to the expected values as well as those that were present previously when Dex was enabled as the provider.

Indeed, from an end user point of view, there is no difference when authenticating against Dex or Direct OIDC integration for Argo CD SSO. By offloading user management to an external, purpose built utility, Argo CD administrators and users can benefit from a simplified experience while reducing the management overhead within Argo CD itself.

SSO Using the Argo CD CLI. In addition to being able to access the Argo CD UI with an SSO user, the same user can also leverage the Argo CD CLI to be able to take advantage of the capabilities provided by the tool. To authenticate as an SSO user from the Argo CD CLI, the `--sso` flag can be specified which will trigger the authentication process with the configured SSO solution.

To enable SSO users to authenticate with the Argo CD CLI, several additional configurations must be implemented within the SSO solution. In our environment, this involves modifications within Keycloak.

Navigate once again to the Keycloak Administration Console at *https://keycloak.upandrunning.local/admin* and authenticate as the admin user.

Two modifications need to be made within the *argocd* Keycloak client.

- An additional callback URL
- Disable Client Authentication

When the CLI initiates the SSO authentication process, it starts a small web server on port 8085. The primary function of this component is to receive the callback after a user authenticates successfully.

Within the Keycloak administration console, navigate to the argocd realm and select Clients on the left hand navigation, and select the argocd Client. Locate the Valid redirect URIs option and click **Add valid redirect URI's** to make available an additional textbox entry. Enter http://localhost:8085/auth/callback into the textbox to allow Keycloak to trust the CLI endpoint and click **Save**.

Since the CLI operates in a similar fashion to a client side web application, it is unable to manage the client credential associated with the Keycloak client. As a result, the access type for the client must be changed from confidential to public which removes the requirement to provide a client secret. Change the access type within the argocd Keycloak client by locating the Capability Config section and deselecting **Client authentication**. Click **Save** to apply the change.

One final modification needs to be made, and this change is specific to the `kind` cluster we are operating within. The CLI sends a set of HTTP headers as it authenticates. However, the content being transmitted is larger than the defaults that are configured within the NGINX ingress controller. Fortunately, this issue can be mitigated by setting the `proxy-buffer-size` parameter within the NGINX configuration which is stored within a ConfigMap in the ingress-nginx namespace.

Update the nginx configuration by setting the `proxy-buffer-size` value to `100k` using the following command:

```
kubectl patch -n ingress-nginx cm ingress-nginx-controller--type='merge'
-p='{"data": {"proxy-buffer-size": "100k"}}'
```

With the required changes applied, login to the Argo CD CLI using the SSO user John (john@upandrunning.local) with the the following command:

```
argocd login --sso --insecure --grpc-web argocd.upandrunning.local
```

Once authenticated, the same user details that are found within the *User Info* page of the Argo CD UI can be seen within the CLI by executing the following command:

```
argocd account get-user-info
Logged In: true
Username: john@upandrunning.local
Issuer: https://keycloak.upandrunning.local/realms/argocd
Groups: ArgoCDAdmins
```

Role Based Access Control

Once a user has authenticated successfully to Argo CD -- whether it be via the CLI or the UI, they are not granted unrestricted access to resources by default and must be granted permissions to perform certain actions.These controls are managed by Argo CD's included Role Based Access Control (RBAC) capability which governs the actions that entities can perform against Argo CD resources. In the prior section, we not only established John, the acting Argo CD administrator, within Keycloak, our user management system, but provided him the ability to log into Argo CD. However, even though he represents an Argo CD administrator, without explicit permissions being granted to his user account, his ability to perform certain actions is restricted.

See this in practice for yourself. Using the Argo CD CLI which has established an authenticated session for John, attempt to list all of the registered certificates and known hosts by executing the following command:

```
argocd cert list
```

Instead of returning the desired result, you will be presented with an error message similar to the following:

```
FATA[0015] rpc error: code = PermissionDenied desc = permission denied: certifi-
cates, get, , sub: 5bcb4862-3cd6-4af2-b784-1039d02bdcb8, iat: 2024-02-05T07:37:18Z
```

A similar message is displayed within the Argo CD UI when performing the same operation and can be seen by clicking on the Settings on the left hand navigation pane and selecting **Repository certificates and known hosts**

Since John is acting as an Argo CD administrator, he should be given the ability to manage all aspects of the Argo CD server. Let's work towards providing him the necessary access that he needs by first reviewing the architecture of the Argo CD RBAC system.

Argo CD RBAC Basics

Argo CD makes use of the Casbin authentication system to define and enforce RBAC rules. Only two roles are included by default:

`role:admin`
: Unrestricted access to all resources

`role:readonly`
: View, but not modify all resources

These roles and the rules behind them take the form of comma separated values and provide a way to define both policies which can then be applied to users and groups.

At a high level, there are two definition structures to define RBAC within Argo CD:

- All resources exception Application related permissions:

```
p, <role/user/group>, <resource>, <action>, <object>, <effect>
```

- Applications, applicationsets, logs, and exec (which belong to an AppProject)

```
p, <role/user/group>, <resource>, <action>, <appproject>/<object>, <effect>
```

Resources represents the following:

```
clusters, projects, applications, applicationsets, repositories, certificates,
accounts, gpgkeys, logs, exec, extensions
```

While an action includes:

```
get, create, update, delete, sync, override, action/<group/kind/action-name>
```

Once a policy is created, it can then be assigned to a user, group, or even another role using the following form:

```
g, <user/group/role>, <role>
```

Since the goal is to provide not only John, but all users that are members of the ArgoCDAdmins group, the ability to manage Argo CD fully, there is no need to create a new policy. Instead, the existing `role:admin` role can be applied to the group.. To do so, the following role mapping policy can be specified:

```
g, ArgoCDAdmins, role:admin
```

RBAC definitions and configurations are specified within a ConfigMap with the name `argocd-rbac-cm` within the namespace Argo CD is deployed within. Policy definitions are contained, by default, within the `policy.csv` key.

While we could modify the argocd-rbac-cm ConfigMap manually or perform an inline patch of the resource, it is easier to manage policy definitions in a separate CSV file.

Create a CSV file called policy.csv which includes the following content:

```
g, ArgoCDAdmins, role:admin
```

Since there is no policy.csv content defined initially within the argocd-rbac-cm ConfigMap, there are no concerns as they relate to overwriting any content that may have been defined.

Execute the following command which will generate a ConfigMap resource containing the policy.csv file and merge it with the existing ConfigMap within the cluster.

```
kubectl create configmap -n argocd argocd-rbac-cm --from-file=policy.csv=policy.csv
--dry-run=client -o yaml | kubectl patch configmap -n argocd argocd-rbac-cm --type
merge --patch-file /dev/stdin
```

If you inspect the contents of the `argocd-rbac-cm` ConfigMap, you will see the `policy.csv` within the ConfigMap matches the content of our local `policy.csv` file.

Now that members of the ArgoCDAdmins group have been granted the role:admin role, confirm that John now has the ability to access and manage all Argo CD resources by once again attempt to list all of the repository certificates and known hosts that have been defined:

```
argocd cert list
```

This time, the full result list should be returned confirming the policy was configured and applied appropriately.

Custom Role Creation

Argo CD includes two roles, `role:admin` and `role:readonly`, that can be designated to users and groups as necessary. However, as more users and groups adopt Argo CD, there becomes a need for a separate role to be created which encompasses the specific permissions that are desired. In the prior section, we covered the basic structure of a role and how it can be applied. In this section, we will define a new role that is targeted at developers and their use case for deploying applications into Kubernetes using Argo CD.

If you recall the setup of Keycloak, two users and groups were defined. We covered John in detail who represents an Argo CD administrator. Mary, the other user defined, is a software developer and is looking to leverage Keycloak, but as a developer, needs to be able to modify certain resources (so the `role:readonly` role does not apply), but does not need full access to Argo CD (disqualifying the `role:admin` role).

Developers require access to perform the following actions:

1. Deploy and manage Applications
2. View a list of clusters for which they could deploy their applications
3. View and access repositories containing their source code
4. View and access certificates and known hosts associated with repositories

Based on the parameters that should be associated with this role, the following policy can be constructed:

```
# Define Policies for a new role called role:developers
p, role:developers, applications, *, */*, allow
p, role:developers, applicationsets, *, */*, allow
p, role:developers, clusters, get, *, allow
p, role:developers, repositories, get, *, allow
p, role:developers, certificates, get, *, allow
# Apply the role:developers role to Developers group
g, Developers, role:developers
```

Breaking down the policies, we first allow developers unrestricted access to Application and ApplicationSets within all projects. Recall, that Application related permissions have a slightly different scheme which includes the name of the Project and the resources within them. To fulfill our requirements, the pattern `*/*` is used which allows for access to all Projects and their resources.

The other two policy permissions enable access to view all cluster and repository definitions. Finally, the role is assigned to the Developers group that is defined within Keycloak.

This policy definition could be appended to the previous policy.csv file which was used in the prior section to grant administrator access to the ArgoCDAdmins group. However, Argo CD does include the functionality to separate policy definitions to allow for policy definitions to be composed (a common use case when using the Kustomize templating tool).

Separate policy files must make use of the format `policy.<any_string>.csv`. With this in mind, create a new file called policy.developers.csv with the policy content provided previously.

With the new policy file created, we could patch the contents to the `argocd-cm` ConfigMap using a similar approach as the `policy.csv`. However, creating policies can be a complex process and introducing a syntactical error is a common occurrence (such as a missing comma). Applying a misconfigured policy could potentially risk the stability of the Argo CD server.

To mitigate these concerns, options are available to perform validation prior to the resource being included within the `argocd-cm` ConfigMap by using the `argocd admin settings rbac validate` command and specifying the desired policy file to validate using the `--policy-file` parameter.

Execute the following command to validate the `policy.developers.csv` policy file.

```
argocd admin settings rbac validate --policy-file=policy.developers.csv
```

If the contents of the policy file does not contain any errors, the message `Policy is valid`. will be displayed. Otherwise, an error will be thrown.

First, before applying the policy, authenticate to the Argo CD UI as Mary, our resident software developer using the username **mary@upandrunning.local** and password **argocdDeveloper123**.

Once authenticated, navigate to the list of repository certificates by selecting **Settings** on the left hand navigation pane and then selecting **Repository certificates and known hosts**.

As expected, a permission error should be displayed.

Apply the `policy.developers.csv` policy by patching the `argocd-cm` ConfigMap using the following command:

```
kubectl create configmap -n argocd argocd-rbac-cm --from-file=policy.develop-
ers.csv=policy.developers.csv --dry-run=client -o yaml | kubectl patch configmap -n
argocd argocd-rbac-cm --type merge --patch-file /dev/stdin
```

Attempt to once again view the Repository certificates and known hosts page within the Argo CD Settings and since the `role:developers` role has been associated with the Developers group for which Mary is a member of, she is now able to view all of the defined certificates and known hosts.

Feel free to validate the remainder of the policies associated with `the role:developers` role including creating, synchronizing and finally deleting an Application.

RBAC Defaults

The RBAC capability provides several different methods for customizing the level of access that users and groups have against Argo CD resources. These assets build upon the default role and their associated policies employed by Argo CD as specified by the `policy.default` property within the `argocd-cm-rbac` ConfigMap. When Argo CD is installed, this property is empty – meaning that no level of access will be granted against any resource. While errors may not be returned when querying resources, no values will be returned.

To enable a specific role to be used when authenticating against Argo CD, the following command can be used to set the `policy.default` property:

```
kubectl patch -n argocd cm argocd-cm-rbac --type='merge' -p='{"data": {"pol-
icy.default": "role:<name_of_role>"}}'
```

Anonymous Access

Argo CD, by default, requires that a user authenticate before being able to access the UI or make queries using the CLI. However, there are capabilities available to enable anonymous access to any entity to access Argo CD resources without needing to authenticate.

Anonymous access can be enabled by setting the `users.anonymous.enabled` property within the `argocd-cm` ConfigMap with a value of `true`. Once enabled, users are granted the level of access granted as specified by the value in the `policy.default` property.

One of the biggest differentiators as it relates to GitOps tools for which Argo CD possesses is the included user interface and the associated integrations – whether it be the command line interface or API. Understanding how these assets can be accessed using Argo CD's included Local Users facility or integrating an external user management system through the SSO functionality enables productivity from day one.

In addition, by using the RBAC capabilities provided by Argo CD, policies can be constructed into roles and applied to users and groups to govern the level of access that these entities have when interacting with the platform.

Conclusion

This chapter provided an overview of how users and groups can be defined and managed along with how Role Based Access Control policies can be defined and configured in Argo CD, resulting in a more secure and productive platform for all.

CHAPTER 6
Cluster Management

> **A Note for Early Release Readers**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 7th chapter of the final book
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *jleonard@oreilly.com*.

Argo CD can deploy applications to the Kubernetes cluster that Argo CD is installed to without further configuration from administrators. This, out of the box, default setting makes it easy for administrators to get up and running and reap the benefits of Argo CD immediately. Whether just starting off in your GitOps journey, or if you are a seasoned DevOps practitioner; this default setting helps administrators to implement their solutions.

The simplicity of the Argo CD deployment can accelerate adoption beyond just a single team, to the point where management of additional clusters is needed and desired. Although you can deploy Argo CD instances to these additional clusters, Argo CD has the ability to add, manage, and deploy resources to additional clusters using a "hub and spoke" design. This hub and spoke design is colloquially known as "Argo CD Control Plane" in larger installations.

In this chapter, we will explore how clusters are managed in Argo CD including how and where they are defined in the control plane, the ways in which they can

be managed, and how we can set up different Role Based Access Control (RBAC) policies to control their access in a multi-tenant situation.

Cluster Architecture

The cluster architecture of Argo CD is fairly straightforward; pon initial deployment, Argo CD has access to the local Kubernetes cluster (i.e. the Kubernetes cluster Argo CD was installed within). This access, as discussed previously, is enabled by default and can be referenced in an Argo CD Application deployment as `https://kubernetes.default.svc` (if using the server key in the configuration file) or `in-cluster` (if using the name key in the configuration file). The creators of Argo CD realized that administrators would like to manage more than just the local Kubernetes clusters, but also deploy to and manage other clusters; concurrently - most administrators would like a single pane of glass view of all their clusters.

Let's take a look at how clusters are defined and managed in Argo CD.

Local vs Remote Clusters

When it comes to clusters, Argo CD doesn't treat the local `in-cluster` any differently than remote clusters. To Argo CD, it sees the `in-cluster` as just another deployment target defined in the Argo CD Application manifest. As we went over in Chapter 4, this is denoted under `.spec.destination` in the Argo CD Application manifest. The following is a snippet of how the target server is defined with an Argo CD Application:

```
spec:
  destination:
    server: https://kubernetes.default.svc
    ## Can also use the following instead of “server”
    # name: in-cluster
    namespace: bgd
```

Remote clusters are referenced the same way. Again, Argo CD treats every cluster the same way - so deploying to a remote cluster is accomplished by merely changing the destination configuration to the desired target cluster. For example:

```
spec:
  destination:
    server: https://cluster1.mydomain.tld:8443
    ## Can also use the following instead of “server”
    ## the below name comes from the cluster secret
    # name: cluster1
    namespace: bgd
```

The namespace will only be set for namespace-scoped resources that have not set a value for the `.metadata.namespace` field.

How are clusters defined? How does Argo CD know what certificate to use to connect to that endpoint? Or, which endpoint to communicate with when specifying `name` instead of `server` within the destination of an Application? What if you want to use a specific ServiceAccount when connecting to the remote cluster? In the next section, we will go deeper into how clusters are defined and how you can further refine how Argo CD connects to these clusters.

Hub And Spoke Design

Before we get into how clusters are defined, it's important to understand that when Argo CD manages clusters, it does so in a hub and spoke design. See Figure 6-1 for a high level view into what this architecture entails.

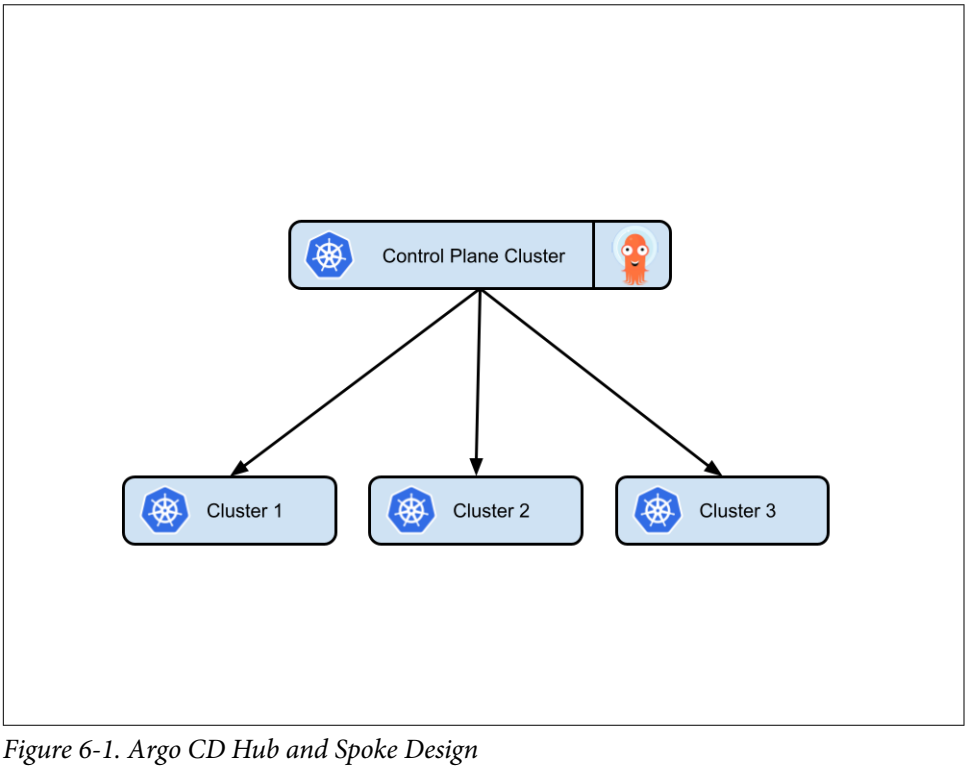


Figure 6-1. Argo CD Hub and Spoke Design

Argo CD "reaches out" in order to perform actions on the target cluster. This is often referred to as the "push model". This means that configurations are obtained and cached on the Control Plane Cluster (where Argo CD is running), and they are "pushed" to the desired destination cluster. It's important to keep this in mind when architecting your installation as considerations, such as firewall rules and accessing the Kubernetes API endpoint need to be taken into account.

How Clusters are Defined

Now that we've established an understanding in how Argo CD sees clusters (whether it is the local cluster or a remote cluster) as just Kubernetes API endpoints (or "destinations"); where does Argo CD retrieve the needed information for this API endpoint? Since the Kubernetes API endpoint has already been established as a means to a connection, Argo CD now needs the credentials for that Kubernetes API endpoint.

Cluster credentials are stored in a Kubernetes Secret in the same namespace as Argo CD is installed within (in our case, this is the `argocd` namespace). To that end, you can surmise that Argo CD clusters are defined via a Kubernetes Secret. The Secret has the following fields:

`name`
: The name given for the cluster. This value is what is referenced when using the `name` property within the `destination` section of the Argo CD Application manifest.

`server`
: The Kubernetes cluster's api server url. This value is what is referenced when using the `server` property in the `destination` section of the Argo CD Application manifest.

`namespaces`
: This is an optional field. One can put a comma-separated list of namespaces which are accessible in the cluster. Note that cluster level resources would be ignored if the `namespace` field is not empty.

`clusterResources`
: This is an optional field. A boolean string ("`true`" or "`false`") determines whether Argo CD can deploy cluster-level resources on this cluster. This setting is used only if the list of managed namespaces in the `namespace` field is not empty.

`project`
: This is another optional string to designate this as a cluster that is only available to the specified Argo CD Project name. Projects will be covered in depth in a following chapter.

`config`
: Written in JSON, this is a representation of the connection configuration.

The config field is where you specify how Argo CD connects to the endpoint of the remote clustert. This is also where additional properties including the cluster CA certificate and the ServiceAccount token are defined.

Here is an example of a minimal configuration of the secret representing a cluster:

```
apiVersion: v1
kind: Secret
metadata:
  name: prod-cluster
  namespace: argocd
  labels:
    argocd.argoproj.io/secret-type: cluster
type: Opaque
stringData:
  name: prod-cluster
  server: https://prod.k8s.example.com:6443
  config: |
    {
      "bearerToken": "<ServiceAccount token should NOT be encoded>",
      "tlsClientConfig": {
        "insecure": false,
        "caData": "<base64 encoded certificate>"
      }
    }
```

For a more in-depth explanation about all available options please consult the official documentation.

It's worth noting that the `bearerToken` section in the `config` field should *not* be base64 encoded and is represented in plain text, while the `caData` section in the `config` field *should* be encoded. Also, the label defines that the content contained in this secret contains cluster related properties.

Customizing the Local Cluster Settings

As mentioned earlier, the local cluster (typically referred to as in-cluster) is the cluster that Argo CD is installed on. There is no need to define this cluster. However, there are use cases where you might need to further refine the settings. By default, this cluster has no Secret associated with it. You can confirm this assessment with the following command:

```
$ kubectl get secrets -n argocd -l argocd.argoproj.io/secret-type=cluster
No resources found in argocd namespace.
```

This is because Argo CD has "sane" defaults for easy deployment. The assumption that Argo CD makes is that it uses the default Kubernetes Service address for the API endpoint, the default Kubernetes CA certificate for that endpoint, and the token for the `argocd-application-controller` ServiceAccount. So, if there is a desire to make updates to the `in-cluster` configuration, how could that be accomplished? Fortunately, the solution is simple.

Let's take the use case where only users who have access to the `sysadmin` Argo CD Project should be able to deploy to the `in-cluster` cluster. To facilitate this requirement, a new Secret that defines the in-cluster configuration needs to be created and within that Secret, the `project` field must grant the sysadmin Argo CD project access. First, create the Kubernetes Secret with the name in-cluster along with the secret type label indicating that the configuration contains an Argo CD cluster definition. Take note that the values specified are the default values with the addition of the `project` field.

The following example is a cluster secret in a file called `in-cluster.yaml`.

```
apiVersion: v1
kind: Secret
metadata:
  name: in-cluster
  namespace: argocd
  labels:
    argocd.argoproj.io/secret-type: cluster
type: Opaque
stringData:
  name: in-cluster
  server: https://kubernetes.default.svc
  project: sysadmin # what we're adding
  config: |
    {
      "tlsClientConfig": {
        "insecure": false
      }
    }
```

Once the file has been created, you can apply it to your cluster by running the following:

```
$ kubectl apply -f in-cluster-secret.yaml
```

You can also update cluster settings in the Argo CD UI under the "Settings" section.

Not only are you able to now see the `in-cluster` configuration listed as a secret, but it also has been scoped to only be available to users who have access to the `sysadmin` Argo CD Project.

You will need to also set the appropriate RBAC in order to scope the in-cluster to only be available to the supplied project. See Chapter 8 for more information about Argo CD RBAC and Projects.

```
$ kubectl get secrets -n argocd -l argocd.argoproj.io/secret-type=cluster
NAME         TYPE     DATA   AGE
in-cluster   Opaque   4      74s
```

You can also see the configuration using the `argocd` CLI.

```
$ argocd cluster get in-cluster -o json | jq -r .project
sysadmin
```

We've added a Project in this configuration for demonstration purposes. We will go over Projects in depth in Chapter 8.

Adding Remote Clusters

There are two methods to add remote clusters to Argo CD: using the `argocd` CLI and declaratively within a Kubernetes Secret. We'll explore each of these methods, but first, let's go over the basics of creating a cluster.

Creating A Cluster

In order to demonstrate how to add a remote cluster, we have to create a cluster using KIND. In order for both `argocd` CLI and Kubernetes Secret to work, we must expose the Kubernetes API endpoint. For this to function properly, the environment variable of the IP address of the host that KIND is running on must be set.

Set an environment variable called `REMOTE_CLUSTER_IP` with the IP address of the host KIND is running on.

```
$ export REMOTE_CLUSTER_IP=192.168.4.134
```

The IP address in your environment will differ.

Given that we are going to be creating a new KIND cluster, we should manage the kubeconfig file separately, export the KUBECONFIG environment variable to reference a file located at `~/remote-cluster.config` which will be populated when the cluster is created.

```
$ export KUBECONFIG=~/remote-cluster.config
```

Next, create the KIND cluster using the name `remote` with the IP address exported previously:

```
$ kind  create cluster --name remote --config=<<EOF
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
networking:
  apiServerAddress: "${REMOTE_CLUSTER_IP}"
EOF
```

You should *take caution* when exposing your Kuberentes API endpoint on a public network.

At this point, two KIND clusters should be running: The one we've been working with has Argo CD installed, and a new one called `remote` that was created now.

```
$ kind get clusters
kind
remote
```

To return to being able to work with the Argo CD cluster, unset the KUBECONFIG environment variable:

```
$ unset KUBECONFIG
```

At this point, we are ready to add the `remote` cluster to our Argo CD instance.

Adding a Cluster with the CLI

As mentioned earlier,the `argocd` CLI utility can be used to interact with the Argo CD instance when accessing the Kubernetes API via `kubectl` when the API is not accessible or is not allowed. To that end, we can use this Argo CD CLI tool to add a cluster using the `kubeconfig` file that was just created. Before the cluster can be added, ensure that you are logged into your Argo CD instance:

```
$ argocd login --insecure --grpc-web --username admin --password=$(kubectl -n
argocd get secret argocd-initial-admin-secret -o jsonpath="{.data.password}" |
base64 -d) argocd.upandrunning.local
```

Once authenticated, the list of currently registered clusters can be listed.

```
$ argocd cluster list
SERVER                          NAME        VERSION  STATUS      MESSAGE  PROJECT
https://kubernetes.default.svc  in-cluster  1.29     Successful
```

You'll notice that the in-cluster is already listed without having to add it. Now, let's add the KIND `remote` cluster we just created with the `argocd cluster add` subcommand, while providing the location of the Kubeconfig path.

```
$ argocd cluster add kind-remote --yes --kubeconfig ~/remote-cluster.config --name
remote
INFO[0000] ServiceAccount "argocd-manager" created in namespace "kube-system"
INFO[0000] ClusterRole "argocd-manager-role" created
INFO[0000] ClusterRoleBinding "argocd-manager-role-binding" created
INFO[0005] Created bearer token secret for ServiceAccount "argocd-manager"
Cluster 'https://192.168.1.254:38187' added
```

A few things to note about the options from this command:

`kind-remote`
: is the name of the Kubernetes context inside the `kubeconfig`. To find the name of the context, we ran kubectl `config get-contexts --kubeconfig ~/remote-cluster.config`

`--yes`
: confirms adding the cluster (without prompting)

`--name`
: sets the name of the cluster in Argo CD

Argo CD uses the Kubeconfig file to connect to the remote cluster and creates a ServiceAccount called `argocd-manager` with a corresponding RBAC in the `kube-system` namespace. This `argocd-manager` ServiceAccount is used by Argo CD to manage the remote cluster. See LINK_HERE for more information about the RBAC that is configured when adding new clusters.

Once the cluster has been added, it will be visible when executing the `argocd cluster list` command once again.

The state will be in an `Unknown` state until something is deployed to the cluster.

```
$ argocd cluster list
SERVER                          NAME        VERSION  STATUS      MES-
SAGE                                                 PROJECT
https://192.168.1.254:38187     remote              Unknown     Cluster has no
applications and is not being monitored.
https://kubernetes.default.svc  in-cluster  1.29     Successful
```

The remote cluster is now ready to be deployed to! You can reference this cluster by the name, `remote`, or by the server address, https://192.168.1.254:38187 as indicated in the output from the prior command in the Argo CD Application manifest. For example:

```
spec:
  destination:
    ## “name” can be used instead of “server”
    # name: remote
    server: https://kubernetes.default.svc/
    namespace: demo
```

Deleting a cluster with the CLI is fairly straightforward. either the name of the cluster or the server address should be specified.

You should remove this cluster if you want to try out the declarative approach in the next section. If you're not planning on trying it out declaratively, don't delete the cluster. We'll be using this cluster later in this chapter.

Remove the cluster using the `argocd cluster rm` command:

```
$ argocd cluster rm --yes remote
Cluster 'remote' removed
```

Confirm the remote cluster is no longer displayed in the list of registered clusters:

```
$ argocd cluster list
SERVER                          NAME        VERSION  STATUS      MESSAGE  PROJECT
https://kubernetes.default.svc  in-cluster  1.29     Successful
```

Adding a Cluster Declaratively

The Argo CD CLI utility is a great way to work with Argo CD as it lowers the barriers of entry. One of the big advantages is that the Argo CD CLI falls under the governance of the Argo CD RBAC. So, administrators can freely give CLI access to the platform without having to give them access to the Kubernetes API (via CLI or other methods).

Still, Administrators following the GitOps Principles would like a more declarative way to define and manage clusters. To support this approach Administrators can opt to (as discussed earlier in this chapter) define clusters via a Kubernetes Secret.

Storing Kubernetes Secrets in plaintext on source code is NOT recommended, and it is a security risk! It is recommended that an appropriate secrets management solution should be utilized. Integrations with various secrets management solutions can be facilitated with operators, like the External Secret Operator or solution specific tools.

Before creating the Secret representing an Argo CD cluster, make sure you are using the correct Kubernetes context (the instance that Argo CD is running within).

```
$ kubectl config get-contexts
CURRENT   NAME          CLUSTER       AUTHINFO      NAMESPACE
*         kind-kind     kind-kind     kind-kind
```

The `kubectl` CLI will be used to create the Secret representing the remote cluster. The secret needs to be in the format that was described earlier in this chapter and the necessary information will be extracted from the `kubeconfig` file using the `kubectl` config command. Before we do that, we need to create a ServiceAccount for Argo CD to use in the `remote` cluster. In addition, RBAC related resources need to be created and associated with the newly created Service Account in the remote cluster. These steps parallel the process that is facilitated by the Argo CD CLI which we will emulate.

If you are following along and you did the example using the Argo CD CLI, you don't need to create the ServiceAccount or the ClusterRoleBinding. You can skip to the creation of the token.

First, create a ServiceAccount called `argocd-manager` in the `kube-system` on the `remote` cluster:

```
$ kubectl create --kubeconfig ~/remote-cluster.config sa -n kube-system argocd-
manager
```

Next, create a ClusterRoleBinding for that `argocd-manager` ServiceAccount, assigning it the built-in `cluster-admin` role.

```
$ kubectl create --kubeconfig ~/remote-cluster.config clusterrolebind-
ing argocd-manager-role-binding --clusterrole=cluster-admin --serviceaccount=kube-
system:argocd-manager
```

Now, generate a token that is associated with the argocd-manager ServiceAccount for Argo CD to use. The token is displayed after executing the command. As a result, we will store it in a variable called `TOKEN` for later use. Note, that a long duration is being specified for the purposes of this example. We recommend assigning a duration that is compliant with your organization's policy and that you rotate these credentials often.

```
$ TOKEN=$(kubectl create token --kubeconfig ~/remote-cluster.config argocd-manager
-n kube-system --duration 999999h)
```

The Kubernetes API server may specify a maximum lifespan of a token that may limit the token validity.

Verify that the TOKEN variable is set:

```
$ echo $TOKEN
```

Using this information, and information that will be extracted from the `kubectl config` command, create the Secret for Argo CD to use.

For more information about the options available when using the `kubectl config` command, consult the Kubernetes Documentation.

```
$ cat <<EOF | kubectl apply -n argocd -f -
apiVersion: v1
kind: Secret
metadata:
  name: remote
  labels:
    argocd.argoproj.io/secret-type: cluster
type: Opaque
stringData:
  name: remote
  server: $(kubectl config view --kubeconfig ~/remote-cluster.config -o json-
path='{.clusters[?(@.name == "kind-remote")].cluster.server}')
  config: |
    {
      "bearerToken": "${TOKEN}",
      "tlsClientConfig": {
        "insecure": false,
        "caData": "$(kubectl config view --raw --kubeconfig ~/remote-cluster.config
-o jsonpath='{.clusters[?(@.name == "kind-remote")].cluster.certificate-authority-
data}')"
      }
    }
EOF
```

This result from the prior command is a Secret in the `argocd` namespace:

```
$ kubectl get secret remote -n argocd --show-labels
NAME     TYPE     DATA   AGE   LABELS
remote   Opaque   3      2m    argocd.argoproj.io/secret-type=cluster
```

The newly added cluster can now be seen using the Argo CD CLI tool:

```
$ argocd cluster list
SERVER                          NAME        VERSION  STATUS      MES-
SAGE                                                 PROJECT
https://192.168.1.254:38187     remote              Unknown     Cluster has no
applications and is not being monitored.
https://kubernetes.default.svc  in-cluster  1.29     Successful
```

Updating clusters managed by Argo CD can be done via the CLI (by using `argocd cluster set`) or by updating the corresponding secret (by using `kubectl patch` or `kubectl edit`). Both methods produce the same result and are useful when there is a need to update cluster configurations, such as ServiceAccount tokens or CA certificates.

Deploying Applications to Multiple Clusters

As we're going through these steps, you can get the sense that Argo CD has the capability to not only manage multiple clusters, but also the ability to deploy resources to multiple clusters as well. However, you may have noticed when going through the

Argo CD Application specification page on the official documentation, that only a single cluster can be defined with an Argo CD Application manifest. In a way, you can think of Argo CD Applications as having a 1:1 relationship with the cluster that application is being deployed to. Effectively, an Argo CD Application can be seen as an instance of your running application.

So, how can we effectively deploy our applications to multiple cluster destinations? Fortunately, several patterns are available to achieve this goal.

App of Apps Pattern

The “app of apps" pattern first appeared as a method of bootstrapping Argo CD instances and can also be used as a method of recovery from a catastrophic failure or major outage. This method is also flexible where organizations have a desire for creating a logical deployment across many clusters. Another advantage is that you can use other Argo CD features, like Sync Waves and Phases you can orchestrate (order) Argo CD Application deployments.

As the name suggests, the app-of-apps pattern is an Argo CD Application that just contains other Argo CD Applications. Since Argo CD Applications are just Kubernetes resources, the Argo CD Application paradigm can be used with other Argo CD Applications. Take a look at Figure 6-2 to see how this approach is depicted in the Argo CD UI:

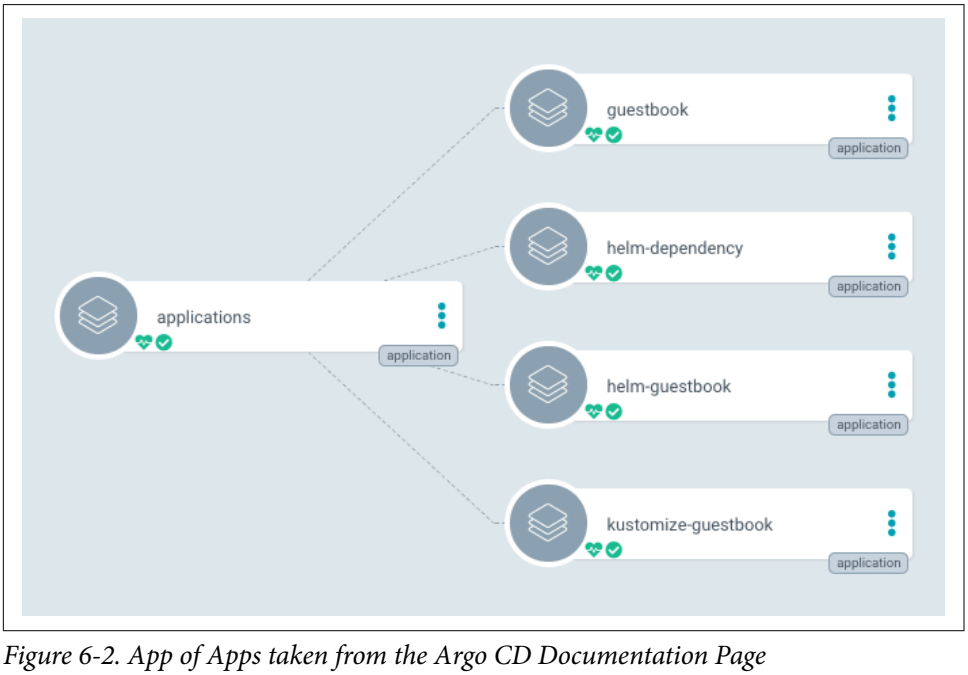


Figure 6-2. App of Apps taken from the Argo CD Documentation Page

The following example can be found in this book's accompanying repository. First, apply the "parent" Argo CD Application

```
$ kubectl apply -n argocd -f ch07/pricelist-app-of-apps.yaml
```

This Argo CD Application manifest included multiple Application manifest which created several Argo CD Applications.

```
$ kubectl get applications -n argocd
NAME                SYNC STATUS   HEALTH STATUS
pricelist-app       Synced        Healthy
pricelist-config    Synced        Healthy
pricelist-database  Synced        Healthy
pricelist-frontend  Synced        Healthy
```

Each of these Argo CD Applications represents the same application with the difference being the target different destination cluster.

Using Helm

Several challenges are introduced when starting to consider Argo CD applications to multiple destination clusters. First, you may be thinking: "that's a lot of YAML to write just for one small delta (changing the destination cluster)". While on the other side: i "I have to change a lot for my application to run successfully on each cluster". As a result, many Argo CD administrators have started utilizing Helm to parameterize the deployment of Argo CD Applications.

Let's take a quick look at the example from the official Argo CD documentation page for using Helm.

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: guestbook
  namespace: argocd
  finalizers:
  - resources-finalizer.argocd.argoproj.io
spec:
  destination:
    namespace: argocd
    server: {{ .Values.spec.destination.server }}
  project: default
  source:
    path: guestbook
    repoURL: https://github.com/argoproj/argocd-example-apps
    targetRevision: HEAD
```

As depicted in the manifest above, certain properties from the Argo CD Application can be parameterized and can be injected using a Helm values file. An example can be found below:

```
spec:
  destination:
    server: https://kubernetes.default.svc
```

While this is still a valid (and fully supported) way of deploying your Argo CD Applications, this pattern of using Helm was first implemented in a time before Argo CD Applications could natively be templated. It is recommended that those who can, migrate to ApplicationSets.

ApplicationSets

An Argo CD ApplicationSet is a Kubernetes CRD that can be seen as a templating engine for Argo CD Applications. This templating engine is fed parameters, known as "Generators" which produces N number of Argo CD Applications based on those provided configurations (which can also include business logic depending on the Generator selected). The original author of the Argo CD ApplicationSet controller described ApplicationSet's as a "factory that produces Argo CD Applications".

The aforementioned "Generators" are a method for producing the necessary information for an Argo CD Application. These Generators range from simple key/value pairs to structures based on your git repository organization layout. Here is a list of Generators at the time of this writing.

List generator
: Generates Argo CD Applications based on a fixed list of any chosen key/value element pairs.

Cluster generator
: Generates Argo CD Applications based on the list of clusters that are defined within (and managed by) Argo CD (which includes automatically responding to cluster addition/removal events from Argo CD).

Git generator
: Argo CD Applications are created based on files within a Git repository, or based on the directory structure of a Git repository.

Matrix generator
: Enables the creation of Argo CD Applications by combine the generated parameters of two separate generators.

Merge generator
: Merge the generated parameters of two or more generators. Additional generators can override the values of the base generator.

SCM Provider generator
: Uses the API of an SCM provider (eg GitHub) to automatically discover repositories within an organization.

Pull Request generator
: Uses the API of an SCM provider (eg GitHub) to automatically discover open pull requests within a repository.

Cluster Decision Resource generator
: Interfaces with Kubernetes custom resources that use custom resource-specific logic to decide which set of Argo CD clusters to deploy to.

Plugin generator
: Provides the great level of customization as the end user customizes the content produced in response to RPC HTTP requests.

For most organizations, starting off with the List generator, Cluster generator, or one of the Git generators (there are two sub generators for the Git generator) is the easiest way to get started with Argo CD ApplciationSets. Let's take another example from the accompanying repository, where an Application is deployed using different settings to separate clusters based on the content originating in different repositories.

<Argo CD ApplicationSet YAML>

```
$ kubectl apply -n argocd -f ch07/appset-bgd.yaml
```

With this one manifest, you can see that the ApplicationSett generated Argo CD Applications based on the parameters of the List generator:

```
$ kubectl get applicationsets -n argocd
NAME   AGE
bgd    29s
$ kubectl  get applications -n argocd
NAME         SYNC STATUS   HEALTH STATUS
bgd-blue     Synced        Healthy
bgd-green    Synced        Healthy
```

One thing to note about Argo CD ApplicationSets is that functionality, such as Sync Waves and Phases between Applications, are not fully supported. If there is a need to leverage such functionality, it is recommended that the standard App-of-Apps pattern be used for the time being. That being said, there is an *alpha* feature (i.e. not ready for production) called ProgressiveSyncs that you can read about in the official documentation site.

Summary

In this chapter, you learned how clusters are defined in Argo CD and how Argo CD is architected in a hub-and-spoke design. You also explored how to add, delete, and manage the lifecycle of the managed cluster. Finally, several patterns for deploying Argo CD Applications to different clusters were introduced. In the next chapter, you will learn how to handle multi-tenant based deployments of Argo CD including

considerations that should be taken under consideration when architecting for multi-tenancy along with several patterns and examples.

CHAPTER 7
Multi-Tenancy

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 8th chapter of the final book

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *jleonard@oreilly.com*.

One of the things that make Argo CD so popular is the user experience that it provides. From the UI to the rich RBAC system; Argo CD continues to not only be the GitOps tool of choice, but also a great choice in providing a rich user experience as well. With that in mind, Argo CD also extends multi-tenancy beyond just basic RBAC. It also has the ability to granularly set access controls based on the actor performing the action (user, group, or automated service account), which resource is being accessed, and what action is being performed.

In Chapter 6 - Authentication and Authorization, you learned about RBAC and its various uses. In this chapter, we are going to extend that knowledge by introducing the Argo CD AppProject concept and how to manage RBAC configurations on a per-Project basis. We are going to start off by demonstrating different Argo CD Deployment models. Then, we are going to explore, in detail, what Argo CD AppProject's are and how to effectively use them in a multi-tenant system. Finally, we will explore how to perform resource management using Projects

Argo CD Installation Modes

There are two primary ways to install Argo CD and each includes a set of capabilities for achieving Multi-Tenancy. As one might expect, there are advantages and disadvantages depending upon the chosen deployment mode. Additional considerations as it relates to multi-tenancy need to be taken into account depending on how your organization is laid out, how your release process is handled, and/or if you have to meet certain criterias for regulation purposes.

Cluster scoped

The most common and default model for deploying Argo CD is the Cluster Scoped method. l.

This is also the deployment method that we have been using thus far during our exploration of Argo CD.

This method is used, specifically, for installations that require Argo CD to act in a multi-tenant mode. This provides all the tooling and features needed (like RBAC, AppProjects (more on that later), and Roles/Groups) for Argo CD Administrators to create a GitOps platform that can support many applications, users, teams, and groups within their organization. From the point of view of Argo CD, it now becomes the interface on how to interact with all managed Kubernetes clusters.

The biggest challenge of a cluster scoped deployment of Argo CD is that, by default, the service accounts associated with Argo CD, effectively has `cluster-admin` privileges on all managed clusters. This was a design decision to enable Argo CD to fully *manage the* cluster. The permissions, however, can be scoped down using standard Kubernetes RBAC by adjusting the ClusterRole/ClusterRoleBinding for the Argo CD service account.

Namespace scoped

The alternate method for deploying Argo CD as is relates to multi-tenancy is the Namespace scoped method. This installation method requires only privileges against a single namespace allowing cluster administrators the ability to install different instances of Argo CD on the same cluster and then delegate the control over to individual teams. Since these installations do not have privileges outside of their own namespace, it is an attractive solution for security conscious Argo CD administrators to achieve multi-tenancy and an increased security posture.

There are a few drawbacks to this type of installation. First, instead of being able to use the `in-cluster` cluster (the default in a cluster scoped deployment of Argo CD), additional steps must be taken to configure the local cluster for use by Argo CD including setting up the associated Service Accounts and RBAC policies.. Another drawback is that the installation doesn't install the Argo CD CRDs as it is assuming that users of a Namespaced deployment of Argo CD will not have the associated privilges as this task requires elevated permissions. Argo CD administrators will need to work with the Kubernetes cluster administrations to make sure that the associated Argo CD CRD's are installed on the cluster. It's also worth noting that there will be overhead in managing multiple Argo CD instances.

Given the number of steps involved for deploying Argo CD using the Namespace method, this book will instead continue to focus on the Cluster scoped method, and this chapter will show you how to utilized the included tools and capabilities needed to set up a multi-tenant system using this installation method.

Projects

Argo CD has a concept of a Project (which is controlled via the `AppProject` CRD). An Argo CD Project provides a grouping of Applications and it is a point of RBAC/demarcation for Argo CD. This logical grouping of Argo CD components is paramount for Argo CD Administrators that are setting up their installation to support multi-tenancy.

With an Argo CD Project, administrators can:

- Restrict the sources of content that can be used (git, helm, etc)
- Restrict where Argo CD Applications can be deployed to (clusters and namespaces)
- Restrict which Kubernetes objects CAN be deployed (Deployments, Services, CRDs, NetworkPolicy, etc)
- Restrict who has access to which resources based on Group/User membership.

Argo CD, by default, includes a Project called `default`. This Project allows the deployment of any resource to any cluster by anyone. While you can't delete the `default` Project, you can lock it down to the point where no one can use it. When Argo CD is initially installed, it has the following permissions for the default project:

```
spec:
  sourceRepos:
  - '*'
  destinations:
  - namespace: '*'
    server: '*'
```

```
  clusterResourceWhitelist:
  - group: '*'
    kind: '*'
```

It's important to note that an Argo CD Application can only ever belong to one Project. When the `AppProject` isn't specified, the `default` Project is used.

Resource Management

Resource management is at the heart of an Argo CD Project and it is what allows Argo CD Administrators to set up the platform to support multi-tenancy. It follows the "allow/don't allow" model where the first matching rule takes precedence. The following are some examples of how this works.

Let's review how you manage Git repositories within an Argo CD Project under the `.spec.sourceRepos` of an `AppProject` manifest.

```
spec:
  sourceRepos:
    - '!ssh://git@github.com:argoproj/test'
    - '!https://gitlab.com/group/**'
    - '*'
```

Note the use of the **!** symbol to indicate an explicit "deny" against the associated repositories. In the example above, Users would not be allowed to deploy from the "test" repository in the "argoproj" GitHub organization; nor would users be allowed to deploy anything from GitLab. However any other repository would be allowed.

Similarly, you can accomplish the same goal for managing the clusters and namespaces that can be deployed to under the `.spec.destinations` property.

```
spec:
  destinations:
  - namespace: '!kube-system'
    server: '*'
  - namespace: '*'
    server: '!https://team1-*'
  - namespace: '*'
    server: '*'
```

Again, note the use of the **!** symbol to indicate an explicit "deny" against those destinations. In this case, users will be able to deploy to any namespace, except the namespace `kube-system` or any cluster with the URL that matches `team1-*`. Any other namespace/server combination would be allowed.

You can also limit what Kubernetes objects may or may not be created. This is for both namespaced and cluster scoped objects. For example, to allow all namespaced-scoped resources to be created, except for `ResourceQuota`, `LimitRange`, and `Network`

`Policy`; you can set the associated policy in the `.spec.namespaceResourceBlacklist` property. For example:

```
spec:
  namespaceResourceBlacklist:
  - group: ''
    kind: ResourceQuota
  - group: ''
    kind: LimitRange
  - group: ''
    kind: NetworkPolicy
```

Conversely, you can deny all namespaced-scoped resources from being created, except for those specified within the `.spec.namespaceResourceWhitelist` proprety. This has the same format as `namespaceResourceBlacklist` shown previously.t

Cluster scoped resources can be constrained in a similar fashion using the `.spec.clusterResourceWhitelist` and `.spec.clusterResourceBlacklist` properties in a similar fashion as their namespace scoped counterpart.t. For example, the following example can be used to deny all cluster-scoped resources from being created, except for a Namespace.

```
spec:
  clusterResourceWhitelist:
  - group: ''
    kind: Namespace
```

In Chapter 6, you learned the basics of RBAC and how it can be configured at the Argo CD platform level. You can also configure RBAC at the Argo CD Project level as well. For example, the following illustrates how to set a policy that only enables those with the `role:developer` permission the ability to `view` and `sync` on the `pricelist` Argo CD Project.

```
spec:
  roles:
  - description: Developers get view and sync
    name: developer
    policies:
    - p, role:developer, applications, get, pricelist/*, allow
    - p, role:developer, applications, sync, pricelist/*, allow
    - p, role:developer, projects, get, pricelist, allow
```

As we've reviewed here, you can see how granular you can get with resource management with Argo CD. You can even apply these policies to specific users and/or groups. In the following section, we will go over a use case to see how AppProjects can be used in your environment.

Use Case: Developer Portal

When working through *Chapter 6 - Authentication and Authorization*, you may have gotten the impression that you can get really granular with RBAC permissions. And you'll be right! That same level of granularity can be achieved at the Project level. This enables Argo CD administrators to grant permission ranging from "read only" to delegating complete control to specific Argo CD Applications.

The most common pattern Argo CD Administrators seem to start with when implementing RBAC at a project level is to grant groups/end-users the ability to see Applications within a Project, perform syncs on demand, but not modify or delete anything. This provides a sort of a "Developer Portal" where end users can see and perform triage and also perform on-demand syncs when needed.

In order to complete this section, you must have set up SSO as described in *Chapter 6 - Authentication and Authorization* as the users and groups will be reused.

Create Project

We will first create the project using the Argo CD CLI which will allow us to deploy an Application that is Project scoped. First, make sure that you are logged in as the "admin" user which provides the necessary permissions to create a Project:

```
$ argocd account get-user-info  -o json | jq .username
"admin"
```

Retrieve the list of currently defined Projects

```
$ argocd proj list -o name
default
```

Only a single Project, `default`, will be returned since this is our first opportunity to manage Projects. Create a new Project called `golist` using the following command:

```
$ argocd proj create golist --src '*' --dest '*,*' --allow-cluster-resource '*/*'
```

This new project should now be present when listing Projects:

```
$ argocd proj list -o name
default
golist
```

With the `golist` Project created, it can be associated with newly created Applications We will configure more granular RBAC for this Project in a later section.

Deploy Applications

Now that the Project `golist` has been created, review the Application manifests in the repository accompanying this book under the `ch08/argocd/applications/` directory.

```
spec:
  # ...omitted for brevity
  project: golist
```

Note that each Application will be deployed into the `golist` Project as denoted under the `.spec.project` section of each manifest. Create each Application using either `kubectl` or the `argocd` CLI (the following example shows makes use of the `argocd` cli)

```
$ argocd app create --file ch08/argocd/applications/golist-db.yaml
$ argocd app create --file ch08/argocd/applications/golist-api.yaml
$ argocd app create --file ch08/argocd/applications/golist-frontend.yaml
```

List the Applications confirming that they were added to the project?

```
$ argocd app list -o name | grep golist
argocd/golist-api
argocd/golist-db
argocd/golist-frontend
```

At this point, the workloads managed by the Applications should be running (note that the database may take some time to become Ready. During this time, you may notice other `Pods` in a `CrashLoopBackOff` state. This is expected and should correct itself after some time)

```
$ kubectl get pods -n golist
NAME                              READY   STATUS    RESTARTS        AGE
golist-api-764879758b-bs57q       1/1     Running   5 (9m59s ago)   11m
golist-db-mariadb-0               1/1     Running   0               10m
golist-frontend-7647cb44d4-g7kvx  1/1     Running   0               10m
```

Now that the Application has been deployed to the Project, the next step is to configure RBAC policies within the Project to grant access to a particular SSO group.

Configure Project

In the previous section, we created the Project imperatively using the `argocd` CLI. While a completely valid way of configuring the Project, the most effective way is to do it declaratively. Take a look in the `ch08/argocd/projects/` directory and you will see a `golist.yaml` Project file. Reviewing the file reveals the following contents.

```
spec:
  # ...omitted for brevity
```

```
roles:
- description: Developers get view and sync
  name: golist-developer
  policies:
  - p, proj:golist:golist-developer, applications, get, golist/*, allow
  - p, proj:golist:golist-developer, applications, sync, golist/*, allow
  groups:
  - Developers
```

In the `polices` section, notice that "get" and "sync" are allowed for all Applications in the Project. All other actions are disallowed since there is an implicit "deny" associated with the RBAC model of Argo CD. Under the groups section contains a list of SSO groups for which the policies will be applied against. This group name originates from the OIDC configuration that was completed in *Chapter 6 - Authentication and Authorization*.

For more information on RBAC and its use, please refer to Chapter 6 - Authentication and Authorization

Apply this Argo CD Project manifest in order to set these configurations:

```
$ argocd proj create --upsert --file ch08/argocd/projects/golist.yaml
```

Test Setup

With the configuration of the `golist` Project complete including the deployment of Applications and policies to grant permissions for a specific group, let's confirm the expected results.

Login to your Argo CD instance as **mary@upandrunning.local** (since this user is part of the **Developer** group), and you should see the aforementioned Applications in the Argo CD overview page as depicted in Figure 7-1

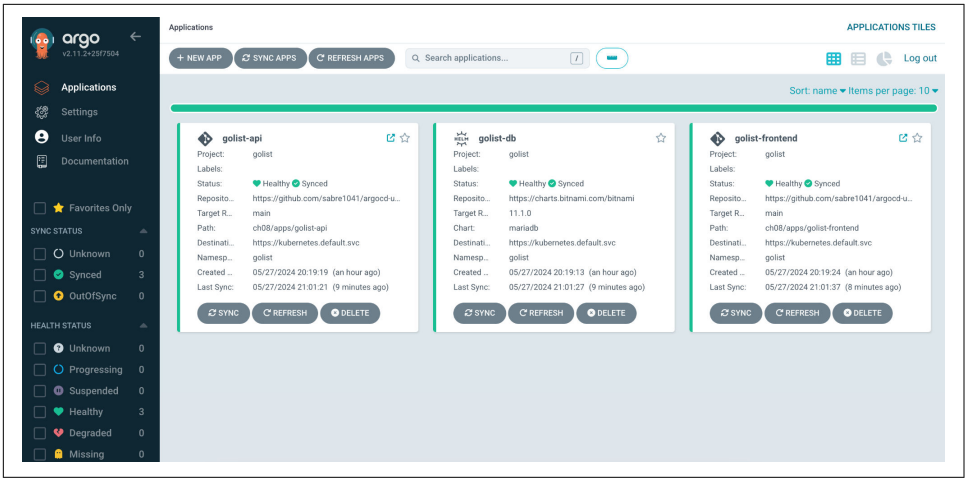


Figure 7-1. Application Overview

On the overview page, click on SYNC APPS, select all Applications, and click on SYNC. You should see all Applications sync, with the status of "Complete" which will appear similar to Figure 7-2

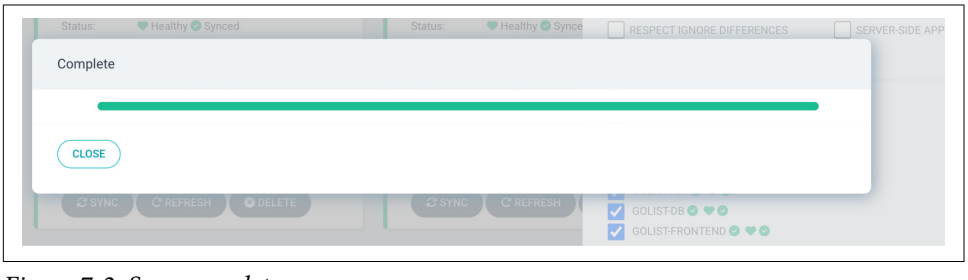


Figure 7-2. Sync complete

Now, click on the **golist-db** "card", click on DELETE and in the popup prompt, type in "golist-db" (leaving the rest of the default values) and click OK. An error similar to Figure 7-3 will be displayed.

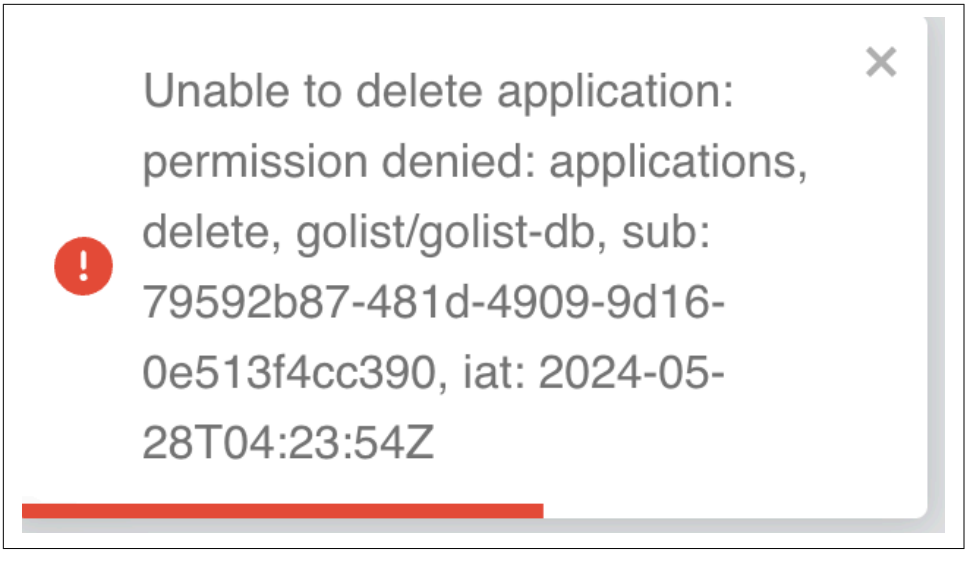


Figure 7-3. Error when deleting

The ability to delete this Application is disallowed since the configuration of the **golist** Project doesn't allow users in the **Developers** group to delete Applications.

Summary

In this chapter, you learned about the two models for how Argo CD implements Multi-tenancy and got familiar with Argo CD AppProject's. You investigated how to manage resources using Projects and how to have fine grained permissions for not only deploying resources, but also the actions that users can perform once they are deployed. Finally, you put this knowledge to use by creating an Argo CD Project, configuring RBAC policies, and verifying that certain actions could only be performed by members of the specified group.

In the next chapter, we will deepen our understanding of how Argo CD managesSecurity. In particular, we will explore the different methods that can be used to harden the security level of Argo CD and how to communicate securely with target systems. In addition, we will also discuss how sensitive content that is used at various points within the Argo CD lifecycle can be handled to avoid being discovered by others.

CHAPTER 8
Security

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 9th chapter of the final book

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *jleonard@oreilly.com*.

One of the top technology concerns, whether from the perspective of an individual developer or enterprise organization is security. Ensuring that systems are protected in a manner that reduces compromise while communicating using secure mechanisms are just some of the steps that can be taken to increase the overall level of security in an environment. Argo CD includes a number of native capabilities that support operating secure operations and enforces certain requirements for use when operating and interacting with the platform.

In this chapter, we will explore the different methods that can be implemented to harden the security level of Argo CD and how to communicate securely with target systems. In addition, we will also discuss how sensitive content that is used at various points within the Argo CD lifecycle can be handled to avoid being discovered by individuals and systems that should not be granted access.

Securing Argo CD

One of the key areas where security hardening can be employed in Argo CD is within the Argo CD Server component as it represents the location where the REST API and user interface is exposed to end users. Considerations should be made whenever there are any externally facing resources as there is an increased potential where attackers could gain unauthorized access to Argo CD.

The admin user enables users the ability to get quickly up to speed with Argo CD. However, this user also presents a potential risk for this misuse by an attacker. First, whenever Argo CD is deployed, a secret called `argocd-initial-admin-secret` is created within the namespace where Argo CD is deployed. within the namespace.

One of the first steps that an Argo CD administrator should take is to change the default password for the admin user. Otherwise, anyone with access to read Secrets within the Argo CD namespace can readily decode the password and gain elevated access to Argo CD. Fortunately, "Chapter 6 - Authentication and Authorization" covered the steps in detail for changing the admin password as well as deleting the Secret containing the initial password as the contents are no longer valid. Of course, if the admin user is no longer being used or needed, the account can be disabled entirely to completely eliminate the potential risk. Steps to accomplish this task are also described within Chapter 6.

Aside from managing the security aspects of the Argo CD server from a user access level, considerations can also be implemented at a transport level; specifically on how end users communicate with either the API or user interface. When Argo CD was deployed, the `--insecure` extra argument was added within the Helm values file. By specifying this parameter, the Argo CD server starts without TLS enabled, enabling the communication with the server to occur without any form of encryption.

Encrypting network traffic using TLS certificates is almost a must these days as it guarantees that the communication with Argo CD can not be easily observed as it is being transferred. However, enabling TLS certificates typically requires additional steps that need to be implemented by the end users, such as the creation of certificates and managing their lifecycle. We saw some of these steps firsthand when configuring Keycloak as an OIDC server in Chapter 6.

Fortunately, Argo CD simplifies the process for enabling TLS by automatically generating a set of certificates at server startup eliminating the need to communicate insecurely whenever the `--insecure` option is not enabled. Let's update the configuration of Argo CD by removing the use of the --insecure extra argument.

While we could update the `argo-cd-argocd-server` Deployment manually, let's use Helm to deploy a new Release of the Argo CD chart and remove the `--insecure` property from the Helm values file.

The updated values file can be found in the `ch09/helm/values` directory of the repository accompanying this Book. Execute the following command to enable TLS within the Argo CD server.

```
helm upgrade -i argo-cd argo/argo-cd --namespace argocd --create-namespace -f
ch09/helm/values/values-argocd-secure.yaml
```

With the new release rolled out, launch a web browser and navigate to the Argo CD user interface at *https://argocd.upandrunning.local*.

What you quickly observed by attempting to navigate to the Argo CD user interface is that something is not configured correctly. Your browser most likely reported an error with "Too many redirects" being the cause. So, what could be the issue?

By default, when the Argo CD Helm chart configures the Ingress resource, it performs what is known as edge termination, resulting in TLS traffic being terminated at the NGINX ingress controller. Traffic is then sent to Argo CD unencrypted. When removing the --insecure argument from the Argo CD server, we effectively closed the method of communication that the NGINX was expecting to be able to communicate. Argo CD responds to the request redirecting to the secure channel, but the subsequent request from NGINX still attempts to connect insecurely. This cyclical loop continues until the maximum allowed setting in the browser, resulting in the error that was displayed.

There are several ways that this issue can be solved, including establishing a new TLS connection from NGINX to communicate with Argo CD. However, the simplest method is to offload the management of certificates entirely and passthrough the connection to the Argo CD backend without any form of TLS termination within the NGINX controller. To accomplish this task, two changes need to be made:

1. Set the `nginx.ingress.kubernetes.io/ssl-passthrough` annotation on the Ingress resource.
2. Enable SSL Passthrough support within the NGINX ingress controller by specifying the `--enable-ssl-passthrough` CLI argument at startup as this feature is disabled by default

CLI arguments for the NGINX controller can be defined within the `controller.extraArgs` Helm value and by specifying `controller.extraArgs.enable-ssl-passthrough=true`, SSL passthrough support will be enabled.

Enable SSL passthrough support within the NGINX Ingress Controller by updating the Helm chart using the `values-ingress-nginx-ssl-passthrough.yaml` values file in the `ch09/helm/values` directory by executing the following command:

```
helm upgrade -n ingress-nginx ingress-nginx ingress-nginx/ingress-nginx -f ch09/
helm/valus/values-ingress-nginx-ssl-passthrough.yaml
```

Finally, specify both the `nginx.ingress.kubernetes.io/ssl-passthrough: "true"` and `nginx.ingress.kubernetes.io/force-ssl-redirect: "true"` annotation on the Ingress resource of Argo CD within the Helm values file to not only enable SSL passthrough support on requests made against this Ingress resource, but to automatically redirect insecure connections (HTTP) to their secure counterparts (HTTPS).

Upgrade the Argo CD Helm chart using the `values-argocd-secure.yaml` values file within the `ch09/helm/values` directory be specifying the following command:

```
helm upgrade -i argo-cd argo/argo-cd --namespace argocd --create-namespace -f
ch09/helm/values/values-argocd-secure.yaml
```

With SSL passthrough support enabled on both the NGINX ingress controller as well as within the Ingress resource for the Argo CD server, once again navigate to *http://argocd.upandrunning.local* in a web browser. Accept the self signed certificate warning that is presented within the browser from the automatically generated Argo CD certificate to confirm the Argo CD server is once again accessible, now with end to end TLS support.

Configuring TLS Certificates

The automatic generation of TLS certificates by Argo CD enables the ability to securely communicate without any additional effort by the Argo CD administrator. However, complications are introduced when relying on this feature as any external system that communicates with the Argo CD server will struggle to fully trust the certificate as it is always generated when the instance starts up. Instead of relying on the automatic certificate generation feature within Argo CD, it is recommended that static certificates be provided by the Argo CD administrator so that a secure and reliable communication can be achieved when communicating with Argo CD components.

While TLS certificates can be configured within each Argo CD component (including dex and the repo server) to avoid the automatic TLS certification generation feature, since end users will directly communicate with the Argo CD server, we will limit our discussion to only this component.

Generating Argo CD TLS Certificates

TLS certificates can be created for the purpose of securely communicating with the Argo CD server. The process for generating certificates was covered briefly in "Chapter 6 - Authentication and Authorization" when Keycloak was deployed to support SSO based authentication. The key difference in this case is that two sets of certificates, a root certificate and another for the Argo CD server, will be generated to enable the creation of a certificate chain. By creating the Argo CD server TLS certificate on top of a root certificate, only the root certificate will be needed to trust

an array of certificates that could be created in the future to serve other purposes or components.

Generate the root certificate by executing the following command:

```
openssl req -nodes -x509 -sha256 -newkey rsa:4096 \
  -keyout root.key \
  -out root.crt \
  -days 365 \
  -subj "/O=O'Reilly Media/CN=Argo CD: Up and Running Root CA" \
  -extensions v3_ca \
  -config <( \
  echo '[req]'; \
  echo 'distinguished_name=req'; \
  echo 'extensions=v3_ca'; \
  echo 'req_extensions=v3_ca'; \
  echo '[v3_ca]'; \
  echo 'keyUsage=critical,keyCertSign,digitalSignature,keyEncipherment'; \
  echo 'basicConstraints=CA:TRUE')
```

Next, generate the TLS certificate for Argo CD based on the root certificate stored in the `root.crt` and `root.key` files:

```
  openssl req -nodes -x509 -sha256 -newkey rsa:4096 \
  -keyout argocd.key \
  -out argocd.crt \
  -days 365 \
  -subj "/O=O'Reilly Media/CN=argocd.upandrunning.local" \
  -extensions v3_ca \
  -CA root.crt \
  -CAkey root.key \
  -config <( \
  echo '[req]'; \
  echo 'distinguished_name=req'; \
  echo 'extensions=v3_ca'; \
  echo 'req_extensions=v3_ca'; \
  echo '[v3_ca]'; \
  echo 'keyUsage=critical,digitalSignature,keyEncipherment'; \
  echo 'subjectAltName=DNS:argocd.upandrunning.local'; \
  echo 'extendedKeyUsage=serverAuth'; \
  echo 'basicConstraints=CA:FALSE')
```

TLS certificates for the Argo CD server are defined in a secret called `argocd-server-tls` within the namespace containing Argo CD. Since a root certificate was also generated in addition to the certificate for the Argo Server, combine the two certificates into a single file called `argocd-fullchain.crt` containing the entire certificate chain.

```
cat argocd.crt root.crt > argocd-fullchain.crt
```

Now create the `argocd-server-tls` secret:

```
kubectl create -n argocd secret tls argocd-server-tls \
  --cert=argocd-fullchain.crt \
  --key=argocd.key
```

The Argo CD server automatically detects the creation of the `argocd-server-tls` secret and will load the newly provided certificate. Navigate to the Argo CD user interface and confirm the newly generated certificate chain is being used. You will once again be greeted with a warning related to trusting the provided certificate. By inspecting the certificate, you can confirm that it matches the instance created previously:

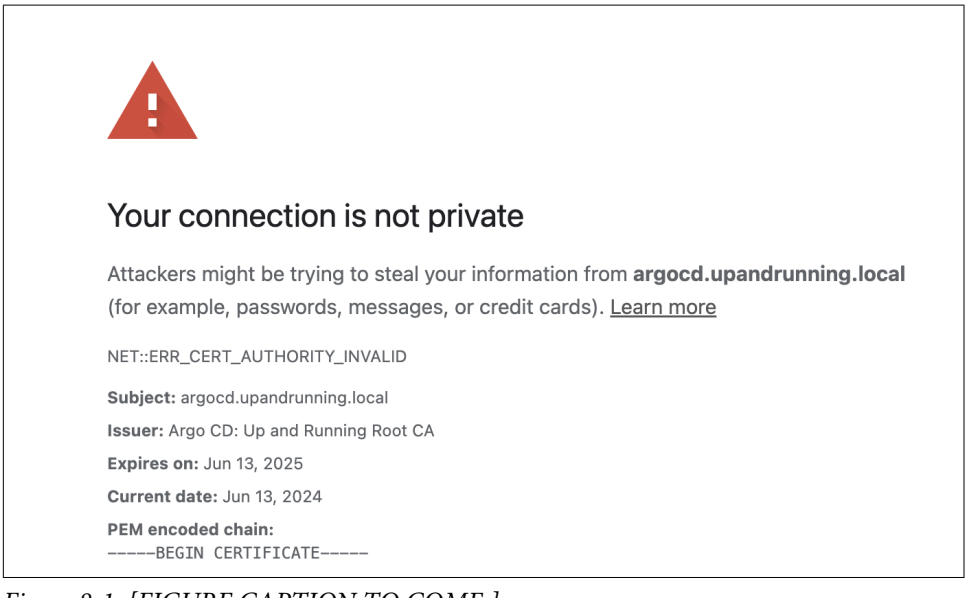


Figure 8-1. [FIGURE CAPTION TO COME.]

Accept the self signed certificate to proceed to the Argo CD user interface.

The root certificate can be configured at an Operating System level to avoid the warnings related to untrusted connection. Since the configurations are Operating System dependent, the steps will not be covered in detail.

The Argo CD server, including how external resources communicate with the REST API and user interface, is just one of the areas for which TLS certificates can be configured. In the following sections, we will explore some of other ways the TLS certificates play a role within Argo CD.

Repository Access

Argo CD, as a tool that implements GitOps practices, interacts with a variety of externally facing resources to source content that can be applied to one or more Kubernetes clusters. These interactions can be configured to communicate in a secure fashion, such as requiring the use of TLS certificates.

Thus far, we have sourced all of the exercise content from the Git repository that corresponds to this publication. This repository is hosted in publicly hosted Git service, and while this service greatly simplifies how anyone can easily access the content, it does limit the type of configurations that can be applied to demonstrate the capabilities of Argo CD. In order to avoid these limitations, we will deploy a Git server of our own to demonstrate some of the ways that Argo CD can be configured to securely communicate with Git repositories.

Given the popularity of Git, there are a multitude of options available when looking to operate a self hosted Git server ranging from an instance that exposes just the git protocol to fully functional collaboration suites. Gitea is an Open Source git platform that offers a good middle ground as it includes a number of useful features, such as a source code and project management capabilities, but also being lightweight compared to other options in the market.

Much like how Argo CD and the rest of the supplemental tools that have been deployed previously throughout this book, Gitea will be installed using a Helm chart. To simplify the interaction with the Gitea instance, it will be initialized with a set of content that we will use throughout this chapter and contained within a wrapper chart located in the `ch09/helm/charts/gitea` directory.

Before the wrapper chart can be used, first, add the upstream Gitea Helm repository:

```
helm repo add gitea-charts https://dl.gitea.com/charts/
helm repo update
```

A custom Helm values file is located in the `ch09/helm/values` directory of the repository accompanying this book. Take a moment and inspect the `values-gitea.yaml` file within this directory containing the Helm Values. Notice within the `ingress` property, details related to `tls` configuration are provided including the name of a Secret containing TLS certificates. Unlike how Argo CD was configured, TLS termination will not occur at the Gitea instance and instead take place within the NGINX Ingress Controller. By including the reference to the Secret containing TLS certificates, these assets will automatically be picked by and configured by the NGINX Ingress Controller.

The creation of the TLS Secret is an "out of band" action and occurs before the installation of the Helm chart. Let's now create a TLS certificate using the same

TLS root certificate that was used for Argo CD. Execute the following command to generate a new Certificate pair for Gitea:

```
openssl req -nodes -x509 -sha256 -newkey rsa:4096 \
-keyout git.key \
-out git.crt \
-days 365 \
-subj "/O=O'Reilly Media/CN=git.upandrunning.local" \
-extensions v3_ca \
-CA root.crt \
-CAkey root.key \
-config <( \
echo '[req]'; \
echo 'distinguished_name=req'; \
echo 'extensions=v3_ca'; \
echo 'req_extensions=v3_ca'; \
echo '[v3_ca]'; \
echo 'keyUsage=critical,digitalSignature,keyEncipherment'; \
echo 'subjectAltName=DNS:git.upandrunning.local'; \
echo 'extendedKeyUsage=serverAuth'; \
echo 'basicConstraints=CA:FALSE')
```

Next, create a new Namespace called `gitea` that will be used to create the Secret containing the TLS certificates and the Gitea instance.

```
kubectl create namespace gitea
```

Now, add the previously generated TLS certificate to the Namespace within a Secret called `git-server-certificate`. Similar to the Argo CD Server, the Gitea and root certificate must be combined into a single file so that it can be added to the secret.

```
cat git.crt root.crt > git-fullchain.crt
```

Create the Secret containing the combined certificate and private key

```
kubectl create secret tls -n gitea git-server-certificate --cert=git-
fullchain.crt --key=git.key
```

Finally, deploy the Gitea instance by installing the wrapper Helm chart with the corresponding Values file. Prepare the wrapper chart by updating the dependencies to pull down the upstream Gitea chart and then install the wrapper chart.

```
helm dependency update ch09/helm/charts/gitea
helm upgrade -i --create-namespace -n gitea gitea --create-namespace ch09/helm/
charts/gitea -f ch09/helm/values/values-gitea.yaml
```

Once the chart has been deployed successfully, launch a web browser and navigate to *https://git.upandrunning.local*. Accept the use of the self signed certificate which will then direct you to the Gitea home page which is depicted below

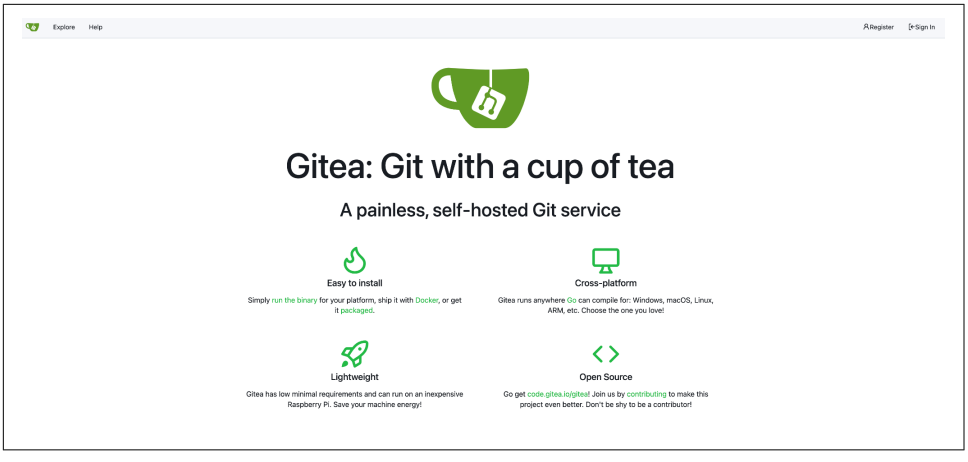


Figure 8-2. [FIGURE CAPTION TO COME.]

On the top right corner of the page, click the Sign In link and use the following credentials:

Username: **gitea_admin**

Password: **Argocdupandrunning1234@**

Once logged in, you will be redirected to the Gitea landing page.

Let's take a moment and review the content that has been automatically populated within the Gitea instance. An organization called *upandrunning* was created and contains a set of Git repositories that will be used throughout this chapter as different concepts are introduced.

On the right side of the page within the *Repositories* box, locate and select the **upandrunning/ch09-tls** repository. The repository includes a directory called *manifests* which contains the Kubernetes resources that will be synchronized by Argo CD.

To make use of this repository as a source of content in Argo CD, an Application called *ch09-tls* is found within the `ch09/argocd` directory in the accompanying book repository in a file called `ch09-tls-application.yaml`.

Apply the manifest to the Kubernetes cluster by executing the following command:

```
kubectl apply -f ch09/argocd/ch09-tls-application.yaml
```

Check the status of the Application using the `argocd` CLI:

```
argocd app get ch09-tls
```

Upon inspecting the output, you will notice that the sync was not successful and the cause (which is also displayed) is noted below:

```
Failed to load target state: failed to generate manifest for source 1 of 1:
rpc error: code = Unknown desc = Get "https://git.upandrunning.local/upandrun-
ning/ch09-tls.git/info/refs?service=git-upload-pack": tls: failed to verify cer-
tificate: x509: certificate signed by unknown authority
```

Similar to the message that was presented when Gitea instance was accessed for the first time in a Web Browser, trust could not be established between the Argo CD repository pod and Gitea. Since a custom Certificate Authority (Root Certificate) was created for these exercises, Argo CD is unaware of the authenticity, and will, by default, deny all communication.

Fortunately, Argo CD provides several options for managing trust when communicating with remote repositories.

Configuring TLS Repository Certificates

TLS Certificates can be configured within Argo CD to allow for the secure communication with remote repositories. These configurations can be applied using either the user interface, CLI or using native Kubernetes resources. Let's use the Argo CD user interface to add the certificate associated with Gitea so that Argo CD will be able to interact with the remote repository in a secure fashion.

Launch the Argo CD user interface at *https://argocd.upandrunning.local*. Click on **Settings** and then select **Repository certificates and known hosts**. Click on **Add TLS Certificate** to launch the dialog for adding the Gitea certificate.

In the *Repository Server Name*, enter **git.upandrunning.local**. Copy the contents of the combined *git-fullchain.crt* file that was created in the prior section when deploying the Gitea instance in the *TLS Certificate (PEM Format)* text area. Click the **Create** button and the newly added certificates will be displayed in the list of known and trusted TLS certificates.

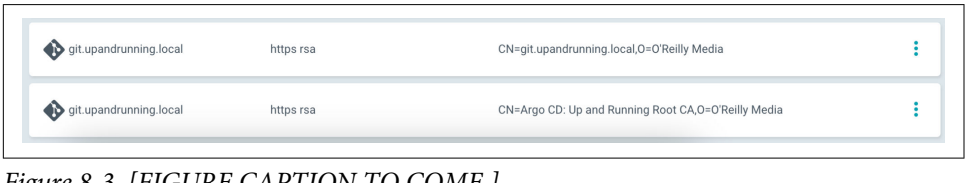


Figure 8-3. [FIGURE CAPTION TO COME.]

Now that the TLS certificates associated with Gitea have been configured in Argo CD, display the configured Applications by clicking on the **Applications** button and then select on the *ch09-tls* Application. Check the status of the Application to determine if the resources stored in the Git repository were applied to the Kubernetes cluster now that Argo CD has been configured to trust the Gitea instance. If the Application is still in an errored state, click the **Refresh** button to manually trigger Argo CD which will allow the Application to attain a Healthy and Synchronized state.

TLS certificates associated with repositories can also be managed using the Argo CD CLI using the `argocd cert` subcommand. List the configured repository using the `argocd cert list` command.

```
argocd cert list
```

What you may have noticed in both the results from the preceding command as well as the page in the Argo CD user interface contains more than the list of TLS repository certificates. Also present are the list of known SSH hosts which will be covered in a later section.

TLS repository certificates can be removed using the `argocd cert rm` command. To remove the previously added certificates associated with the Gitea instance, execute the following command:

```
argocd cert rm git.upandrunning.local
```

To add the Gitea certificate back to Argo CD, use the `argocd cert add-tls` command with the hostname to associate with the certificate and the location of the certificate using the `--from flag` on the local machine as shown below:

```
argocd cert add-tls git.upandrunning.local --from git-fullchain.crt
```

Of course, since Argo CD defines its configurations in a fully declarative fashion, TLS repository configurations can be managed directly within the `argocd-tls-certs-cm` ConfigMap, the same resource that both the CLI or user interface interface with.

The ConfigMap is structured in a straightforward manner, where the key represents the hostname associated with the certificate and the value being the certificate itself and is shown below:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: argocd-tls-certs-cm
  namespace: argocd
data:
  <hostname>: |
    <certificates>
```

Protected Repositories

Thus far, all interactions with remote repositories (whether they be from a Git or Helm source) have been with resources that are readily available and accessible and do not enforce any form of access restrictions. Since the content that is managed by Argo CD can contain either sensitive information or relate to the configuration of the Kubernetes clusters or applications, it is important that appropriate controls are applied to restrict access to only the individuals and systems that require it.

Argo CD includes support for communicating with remote repositories using either HTTPS or SSH based credentials. Both of these credential types and their associated configuration will be described in detail against resources stored within the Gitea instance previously deployed.

HTTPS Credentials

The deployment of the Gitea instance automatically created a set of repositories that require credentials be provided to access the content. They are denoted within the Gitea user interface with the word "Private" next to the repository Several options are available when authenticating with Gitea using an HTTPS based credential and include a user name and password combination or an access token.

The `ch09-credentials-https` repository within the Gitea instance and in the *upandrunning* organization will be used to integrate Argo CD using HTTPS based credentials.

First, let's explore how Argo CD reacts when it attempts to fetch resources that it does not have access to. Apply the `ch09-credentials-https` Application located within the `ch09/argocd/ch09-credentials-https-application.yaml` file.

```
kubectl apply -f ch09/argocd/ch09-credentials-https-application.yaml
```

Check the status of the *ch09-credentials-https* Application using the Argo CD CLI

```
argocd app get ch09-credentials-https
```

As expected, the Application is failing since the content cannot be accessed as authentication is required.

```
Failed to load target state: failed to generate manifest for source 1 of 1: rpc
error: code = Unknown desc = authentication required
```

Similar to TLS certificates, repository credentials can be managed either using the Argo CD user interface or CLI and the configurations that are made using either of these tools are realized as a Kubernetes secret.

First, use the Argo CD user interface to define the credentials to access the *ch09-credentials-https* repository by navigating to the **Settings** page and selecting **Repositories** in a web browser. Click the **Connect Repo** button to begin the process for defining repository configuration.

Enter the following into the dialog:

Connection method: **https**

Project: **default**

Repository URL: **https://git.upandrunning.local/upandrunning/ch09-credentials-https.git**

Username: **gitea_admin**

Password: **Argocdupandrunning1234@**

Additional options are available for configuring TLS client certificates to enable mutual authentication as well as ignoring TLS verification when connecting to remote repositories.

Since mutual authentication was not configured and the Argo CD server has been configured to trust the certificates exposed by the Gitea instance, those options will not be used.

Click the **Connect** button to create the repository configuration.

Confirm the connection status has a green checkmark indicating that verification of the connectivity between Argo CD and the remote repository was successful.

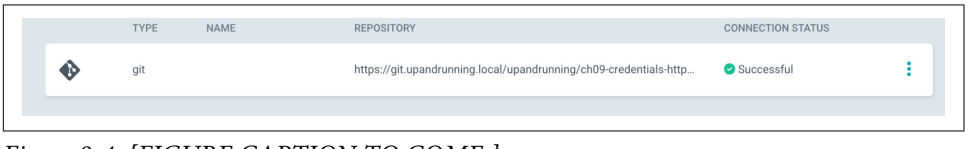


Figure 8-4. [FIGURE CAPTION TO COME.]

With the repository configured and confirmed, navigate to the **Applications** page and select the *ch09-credentials-https* Application and click the **Refresh** button which will make use of the repository configuration created previously to enable the successful synchronization of the Application.

Configuring repository credentials can also be accomplished using the Argo CD CLI with the `argocd repo` subcommand. Using a similar flow that was accomplished in the previous section when managing TLS certificates, first list the defined repository configurations using `argocd repo list`:

```
argocd repo list

TYPE  NAME
REPO                                                                    INSE-
CURE  OCI    LFS    CREDS  STATUS      MESSAGE  PROJECT
git         https://git.upandrunning.local/upandrunning/ch09-credentials-
https.git  false     false  false  true   Successful           default
```

Remove the previously configured repository using the `argocd repo rm` command and include the name associated with the repository (https://git.upandrunning.local/upandrunning/ch09-credentials-https.git as in the example above)

```
argocd repo rm https://git.upandrunning.local/upandrunning/ch09-credentials-
https.git
```

Add the repository configuration back to Argo CD using the `argocd repo add` command while specifying the git repository url, username and password as shown below.

```
argocd repo add https://git.upandrunning.local/upandrunning/ch09-credentials-
https.git  --username=gitea_admin --password=Argocdupandrunning1234@
```

One of the benefits of the CLI over the user interface when adding repository configurations is that connectivity against the remote repository is validated in real time before it is added and an appropriate error is presented. The user interface will add the repository regardless of whether the connection to the remote repository was successful.

When a new repository configuration is added, a Secret is created within the namespace Argo CD is deployed within containing the provided properties. In "Chapter 7 - Cluster Management:, you saw how Argo CD clusters are also defined as Kubernetes Secrets and use the `argocd.argoproj.io/secret-type=cluster` label to denote that the contents contain properties defining an Argo CD cluster

An Argo CD repository configuration is defined in a similar fashion, but utilizes the value of the `argocd.argoproj.io/secret-type` label as "`repository`". The following is how the *ch09-credentials-https* repository configuration would be represented as a Secret:

```
apiVersion: v1
kind: Secret
metadata:
  annotations:
    managed-by: argocd.argoproj.io
  labels:
    argocd.argoproj.io/secret-type: repository
  name: ch09-credentials-https
  namespace: argocd
stringData:
  password: Argocdupandrunning1234@
  type: git
  url: https://git.upandrunning.local/upandrunning/ch09-credentials-https.git
  username: gitea_admin
type: Opaque
```

Aside from a username and password, both Argo CD and Gitea support the use of tokens as a form of authentication. A token can be thought of as a password that typically has a separate life cycle than a standard user account password. Most git based solutions include support for some form of token based authentication. Tokens also have the benefit of being scoped to specific resources or functions, such as access to only certain repositories or the ability to perform certain functions within those repositories (read vs write).

Credentials associated with repositories can be updated using either the Argo CD user interface or using the `argo repo add` command. To update an existing repository configuration, include the `--upsert` flag when invoking the CLI to apply the desired changes.

SSH Based Authentication

The other primary option for authenticating against remote repositories is to use SSH keys. SSH based authentication involves a cryptographic keypair, a public and a private key. The public key is broadly shared and used to determine whether trust should be established while the private key is proof of the users' identity. Let's illustrate how Argo CD can authenticate with the remote Gitea instance to retrieve manifests using SSH based credentials.

The first step is to generate an SSH keypair using the `ssh-keygen` command. Create a new keypair in the current directory using the following command:

```
ssh-keygen -t ed25519 -f argocd_ssh -C "argocd@upandrunning.local" -q -N ""
```

A private key was generated in a file called `argocd_ssh` while the associated public key was generated in a file called `argocd_ssh.pub`. It is important to note that SSH keys with passphrases are not currently supported in Argo CD.

Next, add the public key to Gitea so that it will be able to trust the Argo CD instance when it attempts to communicate using the private key. Gitea supports associating SSH keys with either a user or with individual repositories. To limit the level of access that Argo CD has against the Gitea instance, the previously generated SSH key will be associated with only a single repository within Gitea using a facility called *Deploy Keys*.

Navigate to the Gitea instance (*https://git.upandrunning.local*) and locate the **upandrunning/ch09-credentials-ssh** link within the box denoted by *Repositories* on the right hand side of the page. Click on **Settings** and then **Deploy Keys**. Select the **Add Deploy Key** to define the key that should be trusted for the repository.

Enter **argocd** in the *Title* textbox and paste the contents of `argocd_ssh.pub` file from the generated SSH keypair. Click the **Add Deploy Key** to add the public SSH key to the repository.

Now that Gitea has been configured, the next step is to configure Argo CD. Navigate to the Argo CD instance (https://argocd.upandrunning.local) and once again revisit the Repository configuration page by clicking on the **Settings** button and then selecting **Repositories**.

Adding an SSH repository follows a very similar process as was described previously using TLS certificates (https). Click the **Connect Repo** button. Enter the following into the fields in the dialog:

Connection method: **ssh**

Project: **default**

Repository URL: **git@gitea-ssh.gitea:upandrunning/ch09-credentials-ssh.git**

In the *SSH private key data* field, enter the contents of the SSH private key stored in the *argocd_ssh* file.

Click the **Connect** button to verify the connection.

Argo CD is taking a slightly different path when communicating with the Gitea instance over SSH. Traffic is leveraging the internal Kubernetes Service network as the NGINX ingress controller is only exposing HTTP/S based traffic (80/443). While this configuration does limit direct connectivity over SSH, alternate methods, like `kubectl port-forward` can be used to connect to Gitea via SSH if needed.

Unfortunately, adding the repository will result in a failed connection state. Even though the SSH key that is being used to communicate with the Gitea has been configured at a repository level, an additional step needs to take place for Argo CD to trust connecting to the Gitea instance via SSH.

The SSH protocol includes a series of verification steps to enforce that connections to remote sources are trusted prior to allowing the connection being established. This process of requiring trust is similar to how TLS based connections require that certificates are trusted and verified. SSH clients maintain a list of the public keys that they trust and reference these entries at connection initiation.

To enable Argo CD to connect to Gitea, the public key exposed by the Gitea instance needs to be added to the list of known SSH hosts that Argo CD maintains within the *argocd-ssh-known-hosts-cm* ConfigMap. These entries can be managed on the **Repository certificates and known hosts** page within the *Settings* section of the Argo CD user interface or with the `argocd cert add-ssh` CLI subcommand.

`ssh-keyscan` is one of the tools that can be used to obtain the public key from remote servers. Since SSH access is not exposed outside of the Kubernetes clusters, `kubectl exec` will be used to execute the `ssh-keyscan` command to communicate with Gitea. The output of the command will be redirected to the `argocd cert add-ssh` command which will add the public key to the list of known hosts in Argo CD.

Execute the following command to obtain and add the public key to Argo CD:

```
kubectl -n argocd exec -c repo-server $(kubectl get pods -l=app.kuber-
netes.io/component=repo-server -n argocd -o jsonpath='{ .items[*].meta-
data.name }') -- ssh-keyscan gitea-ssh.gitea | argocd cert add-ssh --batch
```

Confirm the public key was added to Argo CD by navigating to the **Repository certificates and known hosts** page within the *Settings* section of the Argo CD user interface.

Figure 8-5. [FIGURE CAPTION TO COME.]

With the Gitea instance added to the list of known SSH hosts, return to the **Repositories** page within the *Settings* section and confirm the *ch09-credentials-ssh* repository is displaying a successful status. If the status remains in a *Failed* state, disconnect the repository by selecting the kabob menu icon and clicking *Disconnect*. The repository can then be added once again using the values as described earlier in this chapter. Once again confirm that the repository is reporting a successful connection to Gitea.

With the connection to the repository established, create an Application that synchronizes the contents into the Kubernetes cluster. Execute the following command from the `ch09` directory of the accompanying project repository:

```
kubectl apply -f ch09/argocd/ch09-credentials-ssh-application.yaml
```

Confirm that the *ch09-credentials-ssh* Application was not only added successfully, but was synchronized successfully verifying the integration between Argo CD and Gitea using SSH based communication.

Enabling reuse through Credential Templates

One item that might have come to mind when working through this chapter and each of the steps necessary to configure the connectivity to repositories from Argo CD is the long term management and scalability considerations. While ultimately, only two repositories were configured, time and effort was dedicated to support the set up, configuration and verification. Replicating for each repository at a large organization scale, and it becomes a nightmare to consider.

Fortunately, Argo CD includes a capability called Credential templates which allows for a single repository configuration to be defined that can then be reused across multiple repositories. Credential templates make use of URL prefix matching when selecting potential repositories for which the configuration should be applied to.

For example, instead of defining a configuration for each individual repository, a single Credential template that utilized the URL prefix https://git.upandrunning.local/upandrunning, it would match all of the repositories that we have used

thus far as they are all within the same Gitea organization. However, if a repository configuration is defined at an individual repository level, it will take precedence over a Credential template.

To set up a Credential template from the Argo CD user interface, configure the https or ssh repository configuration as described throughout this chapter, but instead of selecting "*Connect*", select "*Save as Credential Template*".

From an Argo CD CLI perspective, the `argocd repocreds` subcommand enables the management of Credential templates. The content that is ultimately persisted as a Secret specifies the label `argocd.argoproj.io/secret-type=repoc-reds`. Which differentiates itself from a standard repository configuration.

While the use of Credential templates will not be covered in depth, feel free to experiment by removing the existing repository configurations and define a single repository configuration that would match all of the private repositories in the *upandrunning* Gitea organization.

Enforcing Signature Verification

Argo CD plays a key role in the overall delivery of software. By managing how and when applications are deployed, it is important to ensure that nothing has unwillingly compromised the integrity of the system. Recent attacks on the software supply chain have caused both organizations and government entities to take a closer look at how they deliver software. One method for ensuring that no malicious activities have occurred during the normal course of how software is built and delivered is to apply cryptographic signatures at various steps throughout this process. By enabling the use of signatures, not only is there a mechanism to understand the origin of the content, but there is an assurance that no unwanted or expected actions occurred after the signature was applied.

Support for signature verification is available in Argo CD and, once enabled, the synchronization of resources can be achieved when the referenced Git repository has a revision that has a GnuPG (GNU Privacy Guard) signature present and the keys used to sign the content has been trusted by Argo CD.

The enforcement that content be signed is applied at a Project level and when configured, applies to every Application associated with the project.

Signature verification is enabled by performing the following steps

1. Importing the public key that was used to sign the content
2. Configure a project and associate one or more of the public keys that Argo CD trusts

Since signature verification applies against commits in a Git repository, at the time of this publication, signature verification is not supported for Helm repositories.

Enable Signature Verification

In order to begin enforcing signatures, a GnuPG formatted public key must be configured in Argo CD. An existing public key may be used or a new keypair can be generated. If your machine does not have the GPG Command Line Tools installed, follow the steps on the GnuPG website to download, install, and confi the tools on your local machine.

https://www.gnupg.org/download/

Once the tools have been installed, generate a keypair:

```
gpg --full-generate-key
```

When prompted, generate an RSA formatted key with a keysize of your choosing. Selecting the default size that is suggested is acceptable. When specifying your personal information, be sure to use an email address that you will remember as it is needed later on when referencing the generated key.

Once a keypair has been generated, obtain the ID of the key.

```
KEY_ID=$(gpg --list-secret-keys --keyid-format=long | grep sec | cut -f2 -d '/'
| awk '{ print $1}')
```

Export the public key in armored format so that it can be added to Argo CD. Be sure to replace the email that was used when generating the key into the command below:

```
gpg --output public.pgp --armor --export <email>
```

GPG keys can be managed either within the *GnuPG public keys* page within the Argo CD user interface or the CLI using the `argocd gpg` subcommand;

Add the exported public key using the `argocd gpg add` command:

```
argocd gpg add public.pgp
```

Confirm the key was added successfully by viewing the list of keys in the Argo CD user interface or by using the `argocd gpg list` command of the CLI.

GPG public keys are stored in the `argocd-gpg-keys-cm` ConfigMap which enables the management of this content in a declarative fashion.

Since signature verification is enforced at a Project level and to avoid affecting any of the existing Applications that have been created previously, create a new Argo CD Project called `ch09-gpg` by applying the AppProject manifest stored in the `ch09-gpg-appproject.yaml` file from within the `ch09/argocd` directory of the accompanying repository.

```
kubectl apply -f ch09/argocd/ch09-gpg-appproject.yaml
```

With the new Argo CD Project created, enable signature verification by adding the ID of the GPG key that was created previously. Navigate to the **Projects** page from within the **Settings** page of the Argo CD user interface.

Select the `ch09-gpg` Project and locate the *GPG Signature Keys* section. Click **Edit** and then **Add Key**. Select the ID of the GPG key from the dropdown and then click **Save**.

Signature Verification in Action

At this point, signature verification has been enabled against the `ch09-gpg` project.

To illustrate just how Argo CD performs and enforces signature verification, create an Application that references content in the *ch09-gpg-signatures* repository in the Gitea instance where commits have not been signed.

```
kubectl apply -f ch09/argocd/ch09-gpg-signatures-application.yaml
```

Inspecting the status of the Application reveals a *ComparisionError* with a message similar to the following:

```
Target revision a9e4a971219b690e2d591605417f8cacba6ab0cf in Git is not signed,
but a signature is required
```

Argo CD has blocked the Application from sync'ing because the commit associated with the revision was not signed with the configured GPG key.

To resolve the error and enable the Application to synchronize successfully, a signed commit must be made against the repository. Since your machine has already been configured with a set of GPG keys and the public key has been installed in Argo CD as the method for signature verification, let's clone the repository locally, enable your git client to use the newly created GPG key, and add a signed commit that can be pushed to the remote repository for Argo CD to use.

First, clone the contents of the *ch09-gpg-signatures* repository to your local machine and change into the repository directory..

```
git -c http.sslVerify=false clone https://git.upandrunning.local/upandrun-
ning/ch09-gpg-signatures.git
cd ch09-gpg-signatures
```

Enter the username and password for Gitea if prompted

The `http.sslVerify=false` config option was specified to ignore TLS certificate errors when communicating with the self signed certificate exposed by the Gitea instance and will be used in each interaction with the Git server.

Next, associate the GPG key with the git client so that it can be used to sign commits by specifying the ID of the key that was stored previously as the *KEY_ID* environment variable.

```
git config --global user.signingkey $KEY_ID
```

Now, update the content of the README.md file in the ch09-gpg-signatures repository so that a signed commit can be made.

```
echo “Now with signed commits!” >> README.md
```

Create a signed commit by specifying the -S flag to enable GPG signing

```
git commit -S -am “Updated README”
```

Confirm the commit was signed by running the following command:

```
git log --show-signature
```

A commit log message with a signature applied will appear similar to the following:

```
gpg: Signature made Sun Jul  7 03:52:14 2024 UTC
gpg:                using RSA key 5CG73B102FD36W88C6F522A1B27298BS6A0E355B
gpg: Good signature from "John Doe <jdoe@upandrunning.local>" [ultimate]
```

With a signed commit being present, the content can be pushed to the remote Gitea instance

```
git -c http.sslVerify=false push origin main
```

Return to the Argo CD user interface and the *ch09-gpg-signatures* Application and click **Sync** to synchronize the Application with the content in the git repository.

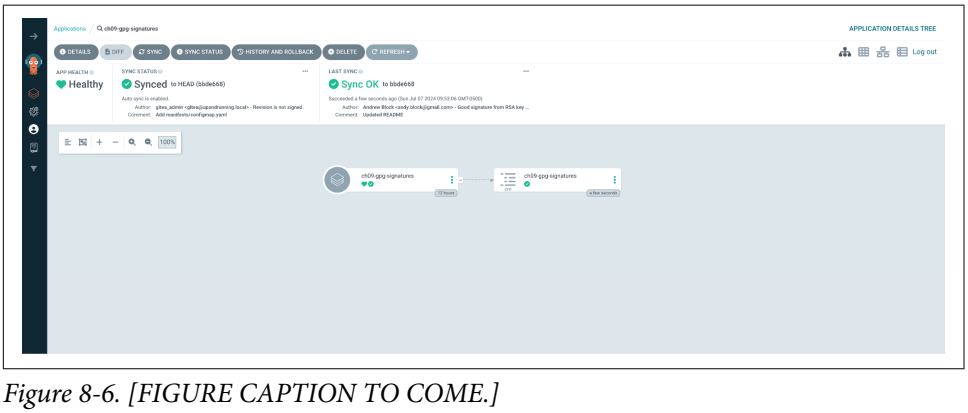


Figure 8-6. [FIGURE CAPTION TO COME.]

Since the *HEAD* revision is signed with the public key that is configured for the *Project* the *Application* is associated with in Argo CD, the synchronization was successful and the associated manifests were added to the Kubernetes cluster.

Signature verification of git commits is just another way that security can be applied within Argo CD. However, if there was a desire to disable the capability entirely, the `ARGOCD_GPG_ENABLED` environment variable can be added to the *argocd-server*, *argocd-repo-server* and *argocd-application-controller* Deployments

Summary

Security will continue to remain an important area of consideration for both Kubernetes developers and administrators. Users interacting with Argo CD can feel confident knowing that the platform includes several features specifically designed to enforce common security practices.

In this chapter, you first learned how to serve custom TLS certificates, enabling end to end encryption between the caller and Argo CD. Then, you deployed an instance of Gitea to act as a git repository, thus allowing for more specialized configurations which are common in many enterprise organizations, to be explored.

Once the Gitea instance was established, you extended your understanding of the benefits of operating securely with TLS certificates and configured trust within Argo CD so that resources stored in Gitea could be accessed securely.

We then transitioned to accessing content stored in protected git repositories and the various methods that Argo CD supports specifying credentials, including HTTPS with usernames/passwords and tokens along with SSH keys.

Finally, the integrity of content from git repositories was hardened by enforcing that commits were signed using a GnuPG key, ensuring that no malicious actions occurred from the time the commit took place to when Argo CD accesses the content.

It is also important to note that while this chapter did cover quite a number of capabilities related to security, it is not a comprehensive list of features that Argo CD supports in this realm. However, the topics covered are some of the most common that apply to Argo CD administrators and users.

CHAPTER 9
Applications at Scale

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 10th chapter of the final book

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *jleonard@oreilly.com*.

In Chapter 4, you were introduced to the `Application` Custom Resource Definition (CRD) object which facilitates the logical grouping of your Kubernetes manifests. This `Application` object serves as the atomic unit of work in Argo CD, allowing you to manage all Kubernetes objects within as a single entity, mirroring how Argo CD handles them. Essentially, Argo CD leverages the `Application` CRD to manage your application deployments in Kubernetes effectively.

Argo CD Applications operate autonomously, meaning that one Application does not have awareness of the status or health of another. This autonomy can pose challenges, especially in organizations employing a microservices architecture where each component resides in its own `Application` CR. For instance, certain Applications may need to be deployed sequentially—such as a database before a backend service or a service mesh before the main application. As infrastructure scales, managing these dependencies and Argo CD Applications becomes increasingly complex.

In this chapter, we will explore various deployment patterns available in Argo CD. These include approaches like the App-of-Apps with Syncwaves and ApplicationSets

with Progressive Syncs. These patterns will assist in managing dependencies between Argo CD Applications and facilitate the deployment and management of these Applications at scale.

Argo CD Application Drawbacks

In Chapter 4, we delved into the concept of an Argo CD Application, exploring its role as the fundamental unit of work within Argo CD. This chapter provided a comprehensive understanding of its structure, functionality, and significance within the broader Argo CD ecosystem. We examined how an Argo CD Application serves as a critical component in managing and automating the deployment of Kubernetes resources. Moving forward to Chapter 4, we engaged in practical exercises that offered hands-on experience with an Argo CD Application. Through these exercises, we learned how to manage a collection of related Kubernetes resources as a cohesive, deployable unit.

In Chapter 5, we explored the customization of the Argo CD Sync operation to accommodate the varying complexities of deployments. While not explicitly stated, the chapter implicitly conveyed the default behavior of an Argo CD Application, which applies Kubernetes manifests as-is. Although this approach is effective in many scenarios, it can pose challenges when the deployment sequence of workloads is critical. To address this issue, the chapter introduced the concept of SyncWaves. SyncWaves provide a mechanism to orchestrate the deployment of Kubernetes resources in a predetermined sequence, thereby ensuring the correct order of operations. This feature is instrumental in mitigating potential issues arising from unordered deployments, thus enhancing the reliability and predictability of the deployment process. For example, if you want to create a `Namespace` before a `Pod`, set the value of the `argocd.argoproj.io/sync-wave` annotation appropriately:

```
apiVersion: v1
kind: Namespace
metadata:
  name: web
  annotations:
    argocd.argoproj.io/sync-wave: "1"
---
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: nginx
  annotations:
    argocd.argoproj.io/sync-wave: "2"
  name: nginx
  namespace: web
spec:
  containers:
```

```
    - image: nginx
      name: nginx
      resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Always
```

While SyncWaves are an effective method for ordering the manifests within a single Argo CD Application, they are limited to the resources contained within that specific application. They do not apply to the Argo CD Application itself or facilitate ordering between multiple Argo CD Applications. Additionally, Argo CD Applications are designed to be autonomous, meaning there is no inherent mechanism for establishing dependencies or relationships between individual applications.

This poses a challenge when managing multiple Argo CD Applications. For instance, consider a scenario where you have an application named "database" that must be deployed and confirmed as healthy before deploying another application named "api." Unfortunately, there is no native feature within the Argo CD Application specification to enforce such dependencies. However, it is still possible to establish dependencies between Argo CD Applications using various tools, methods, and deployment strategies available within the Argo CD ecosystem.

The methods we will cover include:

- Eventual Consistency
- App-of-Apps With Syncwaves
- ApplicationSets Progressive Sync

Before delving into these methods, there are several important considerations to keep in mind, which we will discuss first. These will include things like resource health, Argo CD Application health checks, and Argo CD Application specific health.

Consideration And Best Practices

There are some important considerations to that must be taken into account when implementing any of the approaches with respect to scaling and orchestrating Argo CD Applications. With that in mind, we will review these considerations before delving into any implementation details. One reason being that you'll probably need to implement these regardless which method you go with. The second reason being these are generally best practices when working with Kubernetes and Argo CD.

So before diving into how to handle Argo CD Application at scale, we'll go over some prerequisites and best practices. These include Readiness/Liveness probes, Argo CD Application Health Checks, and Resource Health Checks.

Set up Probes

It's generally good practice to configure readiness and liveness probes within Kubernetes manifests that support them. For those who aren't familiar with the concept, liveness probes assess whther a resource (like a container in your Deployment) is up and running (aka "alive"); readiness probes check to see if your resources are ready to accept connections. For more information about readiness and liveness probes, take a look at the official Kubernetes documentation on the topic.

Setting up readiness and liveness probes is not only a best practice, it's paramount to an Argo CD Application. Argo CD Application health is based on the collective health of the each of manifests being deployed. **Without proper readiness and liveness probes, Argo CD might mark resources as "Healthy" and "Synced" when in fact, they might still be deploying.**

Let's take the scenario of a MySQL database. If we deploy the MySQL StatefulSet without any probes, Argo CD will mark the MySQL StatefulSet as "healthy" even though it might be going through its setup process. Furthermore, it will also be marked as "healthy" when the StatefulSet isn't even ready to start receiving requests! To that end, you can see how adding probes can help when deploying resources with Argo CD. Here is an example of adding probes for MySQL:

```
spec:
  template:
    spec:
      containers:
        - image: mysql:5.6.51
          name: mysql
          livenessProbe:
            tcpSocket:
              port: 3306
            initialDelaySeconds: 12
            periodSeconds: 10
          readinessProbe:
            exec:
              command: ["mysql", "-h", "127.0.0.1", "-e", "SELECT 1"]
            initialDelaySeconds: 12
            periodSeconds: 10
```

In the example above, Kubernetes considers the MySQL StatefulSet as "alive" when port 3306 responds to requests and it will consider it "ready" when a query executes successfully.

Argo CD Health Checks

Argo CD doesn't only rely on the generic Kubernetes health status for the objects it's managing, but it also provides built-in health checks for a multitude of Kubernetes types, which are then surfaced to the overall health status of an Application health

status. Health checks are written in Lua, and you can see the current built-in checks in the Argo CD GitHub repo.

There are times where there's a need to add or customize these health checks. For example, if you're working with a Kubernetes Operator (perhaps because you have either written one for your organization or because you're using a relatively new one), you might need to add these custom health checks in the `resource.customizations` field in the `argocd-cm` ConfigMap. The format looks like the following:

```
data:
  resource.customizations: |
    <group/kind>:
      health.lua: |
```

For example, here is what the health check for the `cert-manager.io/Certificate` object would look like in the `argocd-cm` ConfigMap.

```
data:
  resource.customizations: |
    cert-manager.io/Certificate:
      health.lua: |
        hs = {}
        if obj.status ~= nil then
          if obj.status.conditions ~= nil then
            for i, condition in ipairs(obj.status.conditions) do
              if condition.type == "Ready" and condition.status == "False" then
                hs.status = "Degraded"
                hs.message = condition.message
                return hs
              end
              if condition.type == "Ready" and condition.status == "True" then
                hs.status = "Healthy"
                hs.message = condition.message
                return hs
              end
            end
          end
        end

        hs.status = "Progressing"
        hs.message = "Waiting for certificate"
        return hs
```

To read more about Argo CD Health Checks please refer to the official documentation.

Application Health

Another important thing to note is that the health check for the Argo CD Application CRD has been removed in Argo CD 1.8 (see issue *https://github.com/argoproj/argo-cd/issues/3781* for more information). This is an important thing to keep in mind,

especially in the case of orchestrating Argo CD Application deployments that rely on each other. Since some of the patterns we're going to go through rely on the Argo CD Application health check's presence, we'll need to add it to the `argocd-cm` ConfigMap. This is easily done. Here's an example:

```
data:
  resource.customizations: |
    argoproj.io/Application:
      health.lua: |
        hs = {}
        hs.status = "Progressing"
        hs.message = ""
        if obj.status ~= nil then
          if obj.status.health ~= nil then
            hs.status = obj.status.health.status
            if obj.status.health.message ~= nil then
              hs.message = obj.status.health.message
            end
          end
        end
        return hs
```

With all these considerations (not only are they general best practices, **but they're also prerequisites for the upcoming use cases**) in place, we can start exploring different patterns on how to get Argo CD Application dependencies.

Eventual Consistency

One of the patterns worth mentioning for Argo CD Application orchestration is to rely on the fact that things will eventually be consistent with retries. This can easily be setup using the Argo CD Application manifest itself and also by using Argo CD Sync Option annotation. Here's an example Application manifest.

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: simple-go
spec:
  destination:
    name: in-cluster
    namespace: demo
  source:
    path: deploy/overlays/default
    repoURL: 'https://github.com/christianh814/simple-go'
    targetRevision: main
  project: default
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
```

```
    syncOptions:
      - CreateNamespace=true
      - Validate=false
    retry:
      limit: 5
      backoff:
        duration: 5s
        maxDuration: 3m0s
        factor: 2
```

Notice under `.spec.syncPolicy.syncOptions` that the `Validate=false` option is present. This disables resource validation (equivalent to running `kubectl apply --validate=false`). There are also retries set in this section to tell Argo CD to retry when an error occurs. You can (and probably should) also add the following annotation to resources that are dependant on others resources being present (like a CR of a CRD)

```
metadata:
  annotations:
    argocd.argoproj.io/sync-options: SkipDryRunOnMissingResource=true
```

A "dry run" will be performed if the dependent resource is present.

These two settings, when configured together, will make Argo CD "keep retrying until success or until the retries are exhausted" (whichever comes first). In this way, Argo CD handles deployment orchestration by not handling the specific details, but instead attempting to apply resources.

There are some drawbacks to this pattern. The biggest detriment is that disabling validation can lead to eventual consistency never happening if a resource with an invalid manifest is used. However, more importantly, some things just cannot happen before others. An example of this is when installing the Istio Service Mesh and ensuring that Istio is up and running so that it can inject sidecars in your application before your application starts.

Use Case Setup

Before going through the use cases, we'll need to set up the aforementioned prerequisites in order for orchestration to work properly. This is a one time setup that not only enables you to perform the following use cases; they are also, as stated before, best general practices when using Argo CD. We will be working out of the root directory of the Git repository that accompanies this book.

Verifying Probes

The manifests that we will be deploying are already set up with readiness and liveness probes. You can verify these configurations by using `yq` to inspect these resources.

You can find more information about `yq` at *https://mikefarah.gitbook.io/yq*.

From the root directory run the following commands:

```
## To view the liveness probe
$ yq .spec.template.spec.containers.0.livenessProbe \
ch10/apps/golist-api/golist-api-deployment.yaml

## To view the readiness probe
$ yq .spec.template.spec.containers.0.readinessProbe \
ch10/apps/golist-api/golist-api-deployment.yaml
```

You can verify the other deployment manifest by running the same command against the `ch10/apps/golist-frontend/golist-frontend-deployment.yaml` file

Since we are also deploying a helm chart, you'll need to run helm template to the `ch10/apps/golist-db` directory to verify the presence of those probes.

Adding Argo CD Healthchecks

As described previously in this chapter, Argo CD Applications health checks are disabled by default. You will need to enable the health checks in order to proceed with the use cases in the upcoming sections. We have added a convenient patch file to enable this configuration. From the root directory of the repository, run the following.

```
$ kubectl patch cm/argocd-cm -n argocd --type=merge --patch-file \
ch10/argocd-cm-patchfile.yaml
```

You can verify Argo CD has been updated

```
$ kubectl get -n argocd cm/argocd-cm -o \
jsonpath='{.data.resource\.customizations\.health\.argoproj\.io_Application}'
```

With the probes verified and Argo CD Application health check in place, you can now start with the first use case.

Use Case: App of Apps With Syncwaves

Originally conceived as a method of bootstrapping Argo CD, the App of Apps pattern is basically an Argo CD Application that consists of other Argo CD Applications (Since an Argo CD Application is nothing but a Kubernetes CRD). In Chapter 7, you were introduced with the App-of-Apps pattern and how it can be used to bootstrap Argo CD including how to deploy Argo CD Applications using Argo CD itself..

Extending beyond just bootstrapping, users found other advantages of using this pattern thanks to also having access to other features that Argo CD provides natively. Notably, Argo CD orchestration features, like Syncwaves and Sync Phases. When setting up probes and Argo CD Application Health, you will now have everything you need to set up Argo CD Application deployment orchestration using App-of-Apps and Syncwaves.

Let's take a look at a use case of deploying a 3 tiered application. We will have one Argo CD Application that deploys a frontend app, a backend app, and also a database. We want to have these managed by a "parent" Argo CD Application and we want to deploy these in the following order.

1. Database
2. Backend
3. Frontend

As you're going through examples, you might get some name collisions (duplicate Application names). You may delete former samples from your setup or run the following on a different Kubernetes cluster.

In order to achieve this architecture, we'll have to use Syncwaves with our App of Apps. We first apply the `argocd.argoproj.io/sync-wave` annotation to the Argo CD Application that deploys the "database" application. Taking a look at the annotations for the `ch10/argocd/applications/golist-db.yaml` file you should see the annotation set to "1".

Keep in mind that lower numbers get higher priority.

```
apiVersion: argoproj.io/v1alpha1
kind: Application
```

```
  metadata:
    annotations:
      argocd.argoproj.io/sync-wave: "1"
    name: database
    namespace: argocd
```

Since we want the backend to become available afterwards, we'll annotate that Application with a higher number. In this case, taking a look at the `ch10/argocd/applications/golist-api.yaml` file, note the annotation value is set to "2".

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  annotations:
    argocd.argoproj.io/sync-wave: "2"
  name: backend
  namespace: argocd
```

Finally, we can see in the `ch10/argocd/applications/golist-frontend.yaml` file that the annotation for the frontend Application is set with a higher number than the database and backend so that it comes up last. In our case, it's annotated with "3".

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  annotations:
    argocd.argoproj.io/sync-wave: "3"
  name: frontend
  namespace: argocd
```

The "parent" Application, being just another Argo CD Application, will create the resources in the specified order. Taking a look at the `ch10/argocd/applications/parent.yaml` file, you'll see the following:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: parent
  namespace: argocd
  finalizers:
  - resources-finalizer.argocd.argoproj.io
spec:
  source:
    path: argocd/applications
    repoURL: 'https://github.com/sabre1041/argocd-up-and-running-book'
    targetRevision: main
  destination:
    namespace: argocd
    name: in-cluster
  project: default
  syncPolicy:
    automated:
```

```
    prune: true
    selfHeal: true
  retry:
    limit: 5
    backoff:
      duration: 5s
      maxDuration: 3m0s
      factor: 2
  syncOptions:
    - CreateNamespace=true
```

Once this "parent" Argo CD Application is applied, Argo CD will apply the "child" Argo CD Applications in the order it was annotated with. To start the process, apply the "parent" Argo CD Application by running the following command.

```
$ kubectl apply -n argocd -f \
ch10/argocd/applications/parent.yaml
```

Walking through the process, you will notice the following in the Argo CD UI dashboard:

First, the "parent" Argo CD Application is created and begins the sync process which includes deploying the "database" Application (since it's annotated with a "1"). You can see this in Figure 9-1.

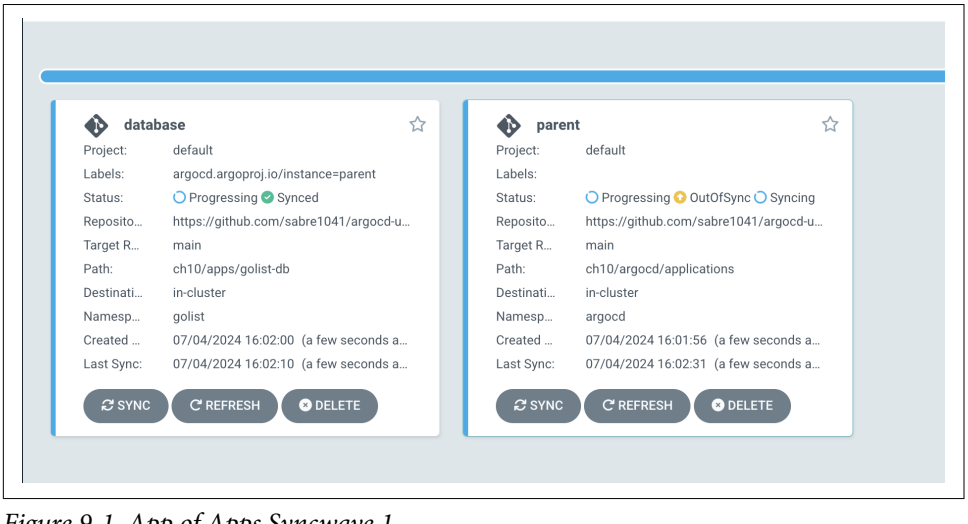


Figure 9-1. App of Apps Syncwave 1

Once the "database" Application is synced and healthy, Argo CD will apply the "backend" Application (as it is annotated with a "2"). You can see this in Figure 9-2

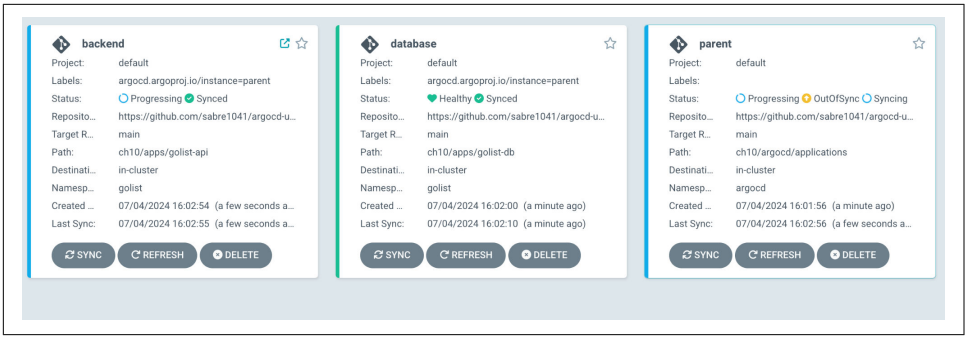


Figure 9-2. App of Apps Syncwave 2

Once the "backend" Application is synced and healthy, Argo CD will finally apply the "frontend" Application (as it is annotated with a "3"). This is represented in Figure 9-3.

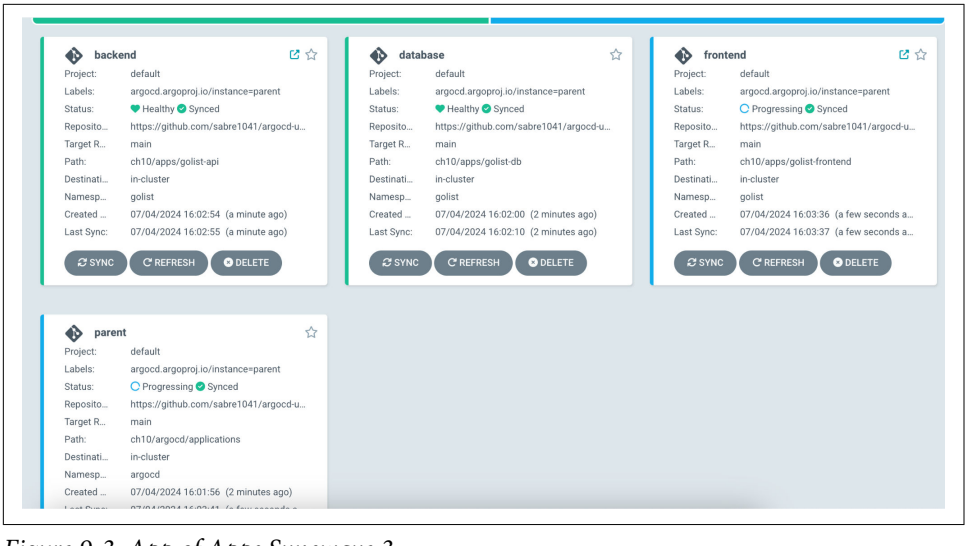


Figure 9-3. App of Apps Syncwave 3

In the end, all 3 Applications that make up this workload, plus the "parent" Application, are synced and healthy. Figure 9-4 should represent the current state in your environment.

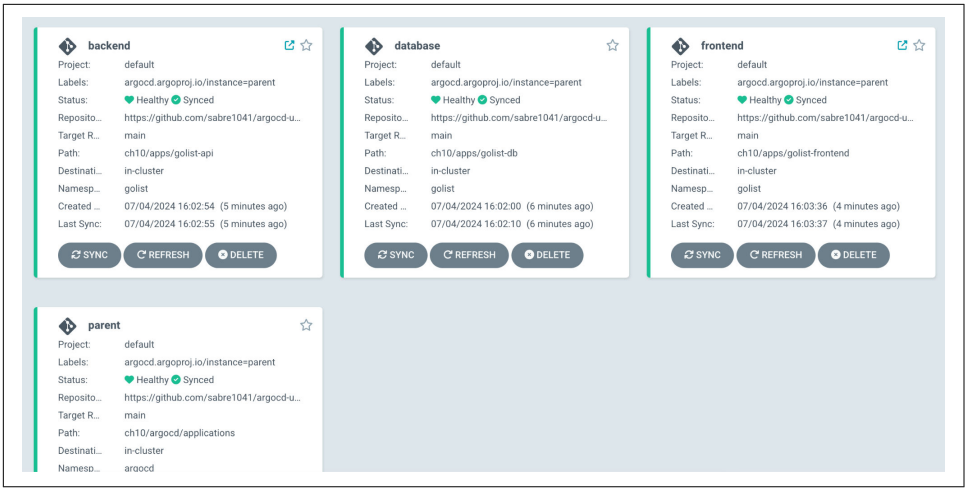


Figure 9-4. App of Apps Syncwave Finished

As you can see, this approach provides a powerful method of setting up Application dependencies, bootstrapping, and performing custom Application deployment orchestration and it's, currently, the recommended way of doing it. **It's worth reiterating that this is all possible because all readiness/liveness probes were set up and Argo CD was configured with the proper LUA health checks.**

ApplicationSets

With all the power that Argo CD gives you with Argo CD Applications and the App-of-Apps pattern, there was still a need to templatize the creation of Argo CD Applications. Yes, we can manage Argo CD Application deployments in a controlled manner. But, we still needed to create those Application manifests.

In Chapter 7, we introduced Argo CD ApplicationSets; which can be seen as an Application "factory". The sole purpose of the Argo CD ApplicationSet controller is to create Argo CD Applications. As you saw in Chapter 7, this gives us the ability to not only create multiple Applications at the same time using a single manifest, but it also allows us to deploy many applications to many destination clusters.

Progressive Syncs

The one drawback of ApplicationSets is that it just generates Applications. There was no built-in mechanism to order or have dependencies. That was until ApplicationSets ProgressiveSyncs was introduced.

The ProgressiveSync feature aims to deploy the Applications in an ApplicationSet in the specified order, while also taking Application health into consideration (meaning

it won't sync Applications unless the previous one is synced and healthy). While a great way to use ApplicationSets ProgressiveSyncs, there are a few things to keep in mind:

1. Generated Applications will have autosync disabled.
2. This is an Alpha feature and will be subject to change. This also means that the feature needs to be explicitly enabled.
3. If an Application has been in a "Pending" state for more that the allotted progressing timeout (default 300 seconds), the ApplicationSet controller will mark it as "Healthy"

Even with ProgressiveSyncs enabled, you still need to set up your readiness/liveness probes and Argo CD Application health. With all these things in mind, let's go over the same example as was described previously, except with the ProgressiveSync feature.

Use Case: Using Progressive Syncs

We can have similar behavior of Application dependency management using ApplicationSet Progressive Sync that you got with the App-of-Apps use case. The biggest advantage of using Progressive Syncs over App-of-Apps is that you only need to manage one manifest.

There are other advantages of using Progressive Syncs. There are features of being able to group many Applications in each deployment phase, but also include things like specifying `maxUpdate`, which allows to only deploy a percentage of Applications at a time in each phase. This is helpful in the situation where you have thousands of applications and want to prevent a "broadcast storm" of syncs happening.

Let's take a look at the same use case of deploying the same 3 tiered application. This time, we will use an Argo CD ApplicationSet that uses a Progressive Sync to deploy that frontend app, along with the backend app, and also that same database we used in the previous use case.

Before anything else, you'll need to remove the existing Applications related to the 3 tier deployment. This can be easily accomplished by deleting the "parent" application.

```
$ kubectl delete application parent -n argocd
```

Since the "parent" Application controls the other Applications (via a finalizer), it will also delete the "children" Applications.

Next, you need to explicitly enable Progressive Sync in Argo CDs. A patch file is included to simplify this process.

```
$ kubectl patch cm/argocd-cmd-params-cm -n argocd --type=json \
--patch-file ch10/argocd-cmd-params-cm-patchfile.yaml
```

Next, the ApplicationSet Controller Deployment must be resarted to pick up the updated configuration.

```
$ kubectl rollout restart deploy/argocd-applicationset-controller -n argocd
```

Any ApplicationSet can use Progressive Syncs. The only configuration difference is the labels that will be added and a new section under `.spec.strategy` in the ApplicationSet YAML. If you take a look at the `ch10/argocd/appsets/progressive sync.yaml` file, you'll see a List generator used with the following strategy:

```
spec:
  # ...omitted for brevity...
  strategy:
    type: RollingSync
    rollingSync:
      steps:
        - matchExpressions:
            - key: golist-component
              operator: In
              values:
                - database
        - matchExpressions:
            - key: golist-component
              operator: In
              values:
                - backend
        - matchExpressions:
            - key: golist-component
              operator: In
              values:
                - frontend
```

Take note of the `.spec.strategy` section ait includes "steps". This is how ordering is accomplished, similar to the App-of-Apps with Syncwaves method. This section allows you to group Applications by the labels present on the generated Application resources. When the ApplicationSet changes, the changes will be propagated to each group of Application resources sequentially. ProgressiveSyncs uses the familiar `matchExpressions` that is found in various standard Kubernetes resources. You can potentially group together hundreds of Applications in each "step"

The next section that we want to call attention to is the `.spec.template.meta data.labels` section in the same ApplicationSet manifest.

```
spec:
  # ...omitted for brevity...
```

```
  template:
    metadata:
      name: '{{srv}}'
      labels:
        golist-component: '{{srv}}'
```

This section will apply the label to the corresponding Argo CD Application that this ApplicationSet creates. Then the ProgressiveSync operation will use these labels to determine which Argo CD Application gets synced in each step.

To start the process, apply the ApplicationSet in the `progressivesync.yaml` file

```
$ kubectl apply -n argocd -f ch10/argocd/appsets/progressivesync.yaml
```

Once applied, it will create all 3 of the Argo CD Applications at once (in contrast to App-of-Apps pattern where they are created as they are synced); but they will remain "missing / out of sync". Then it will progress to syncing the first "database" Application. You can see this in the Argo CD UI as depicted by Figure 9-5.

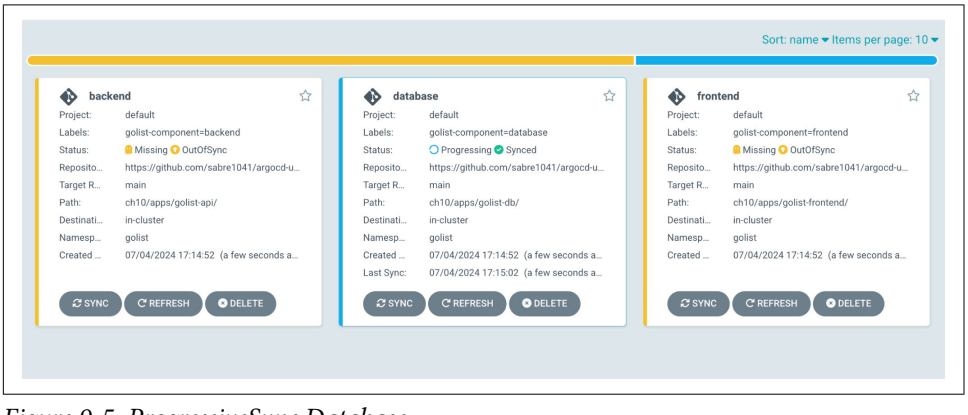


Figure 9-5. ProgressiveSync Database

Once the "database" is synced, the "backend" Application will start syncing. A similar representation appears in Figure 9-6.

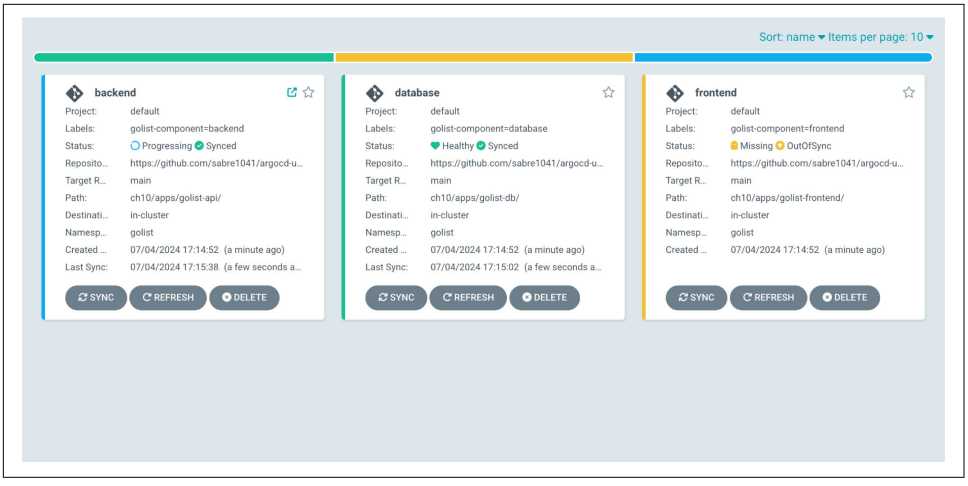


Figure 9-6. ProgressiveSync Backend

When the "backend" Application becomes synced, the "frontend" Application will begin to sync. A state similar to Figure 9-8 will be present in the Argo CD UI.

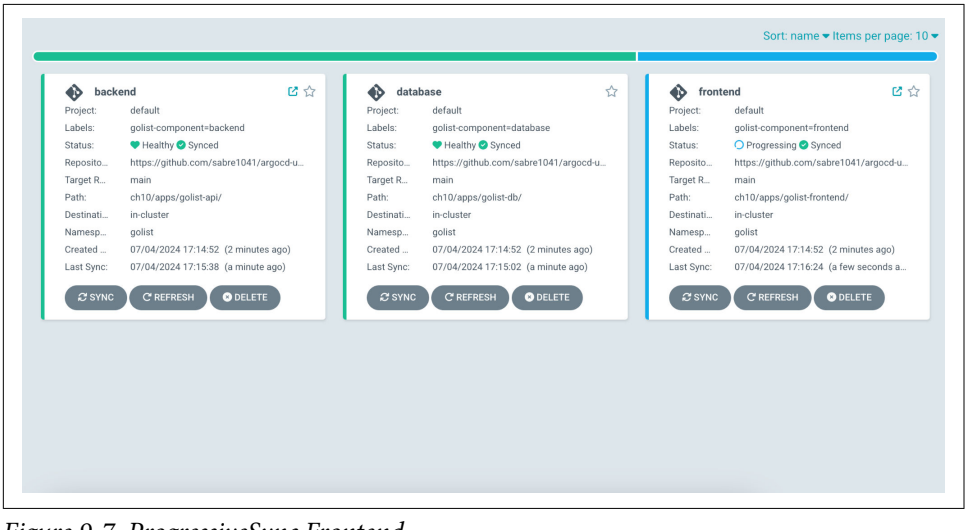


Figure 9-7. ProgressiveSync Frontend

In the end, it should appear similar to that of the "App of Apps" method, except there is no "parent" Application since we are using an ApplicationSet to deploy this workload. You will see all Applications synced and healthy in the Argo CD UI as in Figure 9-8.

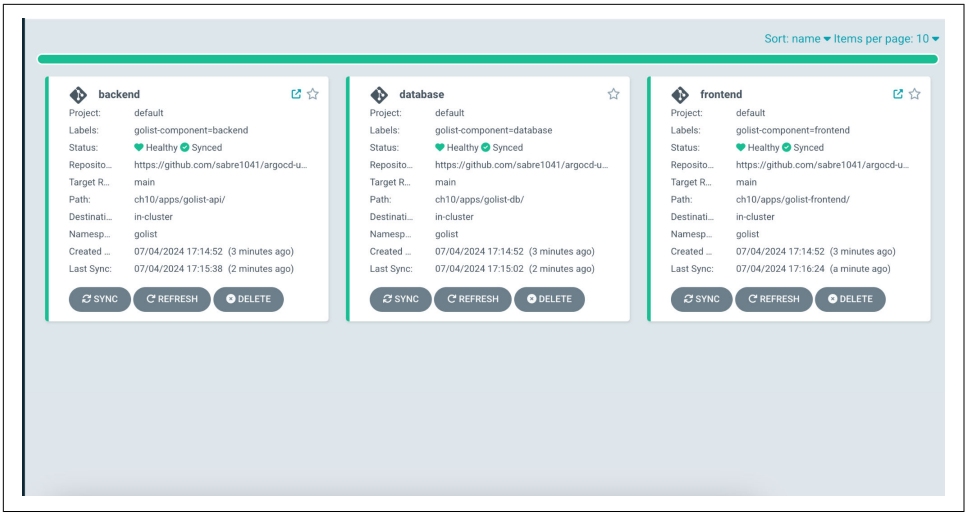


Figure 9-8. ProgressiveSync Finished

The end result is the same; except that with ProgressiveSyncs; there is only one manifest to create and manage.

It's important to note that it's not App of Apps vs ProgressiveSyncs. There are some situations where you could use both or a combination of both (Like "App of AppSets" or "AppSet of App of Apps"). It varies from organization to organization and the specific implementation.

Summary

In this chapter, we delved into the complexities and strategies of managing large-scale deployments with Argo CD. We reviewed the foundational concept of the Application Custom Resource Definition (CRD) object, introduced in Chapter 4, which helps in logically grouping Kubernetes manifests. Also in this chapter, we highlighted the challenges that arise due to the autonomous nature of Argo CD Applications, especially in microservices architectures where sequential deployment dependencies exist.

We provided an in depth look into various deployment patterns to tackle these challenges, such as the App-of-Apps with Syncwaves and ApplicationSets with Progressive Syncs. These patterns are designed to manage dependencies and orchestrate the deployment of multiple Argo CD Applications effectively. Additionally, we also went through the importance of readiness and liveness probes, Argo CD Application health checks, and resource health checks as best practices to ensure the reliable and predictable deployment of Applications. All of these concepts came together to support the deployment of a multi-tier application using these advanced patterns,

demonstrating how to set up and manage these dependencies, highlighting the nuances and benefits of each method in a scalable deployment environment.

CHAPTER 10
Extending Argo CD

> **A Note for Early Release Readers**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 11th chapter of the final book
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *jleonard@oreilly.com*.

Thus far, we have described many of the features that are included within Argo CD to not only manage resources effectively using GitOps patterns in Kubernetes, but to provide a rich set of options for end users to interact with the platform. By offering a way to integrate tools and frameworks common to Kubernetes, complex workflows can be developed to create a robust management strategy for infrastructure and applications. However, even with all of the supported set of capabilities, there may be a need to integrate an additional set of components that are not natively included within Argo CD or to customize the platform itself to better serve the needs of end users.

In this chapter, we will introduce several different mechanisms that can be used to extend the default configuration of Argo CD including the use of a pluggable framework to incorporate additional tools to support how Kubernetes resources are created. These options give end users the power to take Argo CD to the next level.

Config Management Plugins

Kubernetes resources in Argo CD can be created using a variety of methods. They may be declared using standalone manifests or incorporate one of the included set of config management tools, such as Helm, Kustomize or Jsonnet. While these tools represent some of the most common options available for managing Kubernetes resources, there became a need to provide a facility for which additional options were available to customize the generation of Kubernetes resources.

Kubernetes itself faced a similar challenge early on where it only provided a finite list of resource types and API's for users and systems to interact with. This limitation could have reduced the impact that Kubernetes would ultimately have on the IT industry. However, it was the introduction of Custom Resource Definitions (CRD's) which enabled the ability to extend the types of resources served by Kubernetes and unleash an entirely new way to work with the platform. Argo CD provides its own solution to the configuration management tool challenge through the use of Config Management Plugins which offers a flexible method for enabling additional options for facilitating the creation of Kubernetes resources.

If you recall, the repo server is the component responsible for building Kubernetes resources using one of the supported configuration management tools. For alternate tools to be used, it is within this location where tasks need to be executed.

A Config Management Plugin consists of two parts:

- A `ConfigManagementPlugin` manifest describing how and when the plugin should be used
- Tooling the enable the execution of the plugin

The use of alternate tools will typically require additional dependencies, such as binaries associated with the tool and scripts containing the logic employed by the plugin. While the repo server image could be extended to include these custom assets, the preferred approach is to package any of the necessary assets into a separate container and run this container alongside the repo server. This model is known as the sidecar pattern in Kubernetes as it has a number of benefits:

- Avoids conflicts between the plugin and the repo server
- Eliminates the need to manage the lifecycle of the repo server
- Own the entire lifecycle plugin and its components

With an understanding of the high level set of components involved when integrating Config Management plugins, let's explore each of these items in depth and how they can be used to implement a plugin within Argo CD.

The ConfigManagementPlugin Manifest

The `ConfigManagementPlugin` manifest provides instructions to the repo server so that it understands when the plugin should be invoked and how it should be invoked. The following contains the structure of the manifest:

```
apiVersion: argoproj.io/v1alpha1
kind: ConfigManagementPlugin
metadata:
  # The name of the plugin must be unique within a given Argo CD instance.
  name: my-plugin
spec:
  # The version of your plugin. Optional. If specified, the Application's
spec.source.plugin.name field
  # must be <plugin name>-<plugin version>.
  version: v1.0
  # The init command runs in the Application source directory at the beginning
of each manifest generation. The init
  # command can output anything. A non-zero status code will fail manifest
generation.
  init:
    # Init always happens immediately before generate, but its output is not
treated as manifests.
    # This is a good place to, for example, download chart dependencies.
    command: [sh]
    args: [-c, 'echo "Initializing..."']
  # The generate command runs in the Application source directory each time
manifests are generated. Standard output
  # must be ONLY valid Kubernetes Objects in either YAML or JSON. A non-zero
exit code will fail manifest generation.
  # To write log messages from the command, write them to stderr, it will
always be displayed.
  # Error output will be sent to the UI, so avoid printing sensitive informa-
tion (such as secrets).
  generate:
    command: [sh, -c]
    args:
      - |
        echo "{\"kind\": \"ConfigMap\", \"apiVersion\": \"v1\", \"metadata\":
{ \"name\": \"$ARGOCD_APP_NAME\", \"namespace\": \"$ARGOCD_APP_NAMESPACE\",
\"annotations\": {\"Foo\": \"$ARGOCD_ENV_FOO\", \"KubeVersion\": \"$KUBE_VER-
SION\", \"KubeApiVersion\": \"$KUBE_API_VERSIONS\",\"Bar\": \"baz\"}}}"
  # The discovery config is applied to a repository. If every configured discov-
ery tool matches, then the plugin may be
  # used to generate manifests for Applications using the repository. If the
discovery config is omitted then the plugin
  # will not match any application but can still be invoked explicitly by
specifying the plugin name in the app spec.
  # Only one of fileName, find.glob, or find.command should be specified. If
multiple are specified then only the
  # first (in that order) is evaluated.
  discover:
```

```
    # fileName is a glob pattern (https://pkg.go.dev/path/filepath#Glob) that
is applied to the Application's source
    # directory. If there is a match, this plugin may be used for the Applica-
tion.
    fileName: "./subdir/s*.yaml"
    find:
      # This does the same thing as fileName, but it supports double-start
(nested directory) glob patterns.
      glob: "**/Chart.yaml"
      # The find command runs in the repository's root directory. To match, it
must exit with status code 0 _and_
      # produce non-empty output to standard out.
      command: [sh, -c, find . -name env.yaml]
  # The parameters config describes what parameters the UI should display for
an Application. It is up to the user to
  # actually set parameters in the Application manifest (in
spec.source.plugin.parameters). The announcements _only_
  # inform the "Parameters" tab in the App Details page of the UI.
  parameters:
    # Static parameter announcements are sent to the UI for _all_ Applications
handled by this plugin.
    # Think of the `string`, `array`, and `map` values set here as "defaults".
It is up to the plugin author to make
    # sure that these default values actually reflect the plugin's behavior if
the user doesn't explicitly set different
    # values for those parameters.
    static:
      - name: string-param
        title: Description of the string param
        tooltip: Tooltip shown when the user hovers the
        # If this field is set, the UI will indicate to the user that they must
set the value.
        required: false
        # itemType tells the UI how to present the parameter's value (or, for
arrays and maps, values). Default is
        # "string". Examples of other types which may be supported in the
future are "boolean" or "number".
        # Even if the itemType is not "string", the parameter value from the
Application spec will be sent to the plugin
        # as a string. It's up to the plugin to do the appropriate conversion.
        itemType: ""
        # collectionType describes what type of value this parameter accepts
(string, array, or map) and allows the UI
        # to present a form to match that type. Default is "string". This field
must be present for non-string types.
        # It will not be inferred from the presence of an `array` or `map`
field.
        collectionType: ""
        # This field communicates the parameter's default value to the UI.
Setting this field is optional.
        string: default-string-value
      # All the fields above besides "string" apply to both the array and map
```

```
type parameter announcements.
      - name: array-param
        # This field communicates the parameter's default value to the UI.
Setting this field is optional.
        array: [default, items]
        collectionType: array
      - name: map-param
        # This field communicates the parameter's default value to the UI.
Setting this field is optional.
        map:
          some: value
        collectionType: map
    # Dynamic parameter announcements are announcements specific to an Applica-
tion handled by this plugin. For example,
    # the values for a Helm chart's values.yaml file could be sent as parameter
announcements.
    dynamic:
      # The command is run in an Application's source directory. Standard out-
put must be JSON matching the schema of the
      # static parameter announcements list.
      command: [echo, '[{"name": "example-param", "string": "default-string-
value"}]']

  # If set to `true` then the plugin receives repository files with original
file mode. Dangerous since the repository
  # might have executable files. Set to true only if you trust the CMP plugin
authors.
  preserveFileMode: false
```

As depicted above, there are a wide range of options that a ConfigManagementPlugin manifest supports. However, there are only a few properties that you need to be concerned with whenever developing and using a Config Management Plugin as they dictate how the plugin will operate. A description of each of these key properties are listed below:

init:
: Optional parameter which performs any preparation steps that the plugin requires, such as downloading dependencies.

generate:
: Performs the primary function of the plugin. This action runs within the directory associated with the Argo CD Application and can be implemented in a variety of ways including executing a script, binary or printing arbitrary content. The only requirement is that the only output that is produced from this stage be a set of valid YAML or JSON formatted Kubernetes manifests.

discover:
: A set of rules that determines whether the Application is applicable for execution. Common examples include searching for the presence of a file in the application

source or executing a command to perform more complex capabilities. The exit code determines whether the plugin is applicable for the content.

They each are located directly underneath the `.spec` property and work hand in hand to determine the applicability of a plugin for the source Application and the steps necessary to produce Kubernetes manifests.

To determine whether a plugin should be executed for a given Application, two methods are available. First, the `discover` property within the ConfigManagementPlugin can either match the name of a file, or file based on a glob pattern in the content source or return a 0 exit code as a result of the execution of a command. Otherwise, the name of the plugin can be explicitly defined on the Application manifest as shown below:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: guestbook
spec:
  source:
    plugin:
      name: pluginName
```

In most cases, you will want to both abstract when a plugin is executed as well as the need for the end user to define the plugin within their `Application`. A common example for determining whether a plugin should be executed using the auto discovery capability is the presence of a particular file within the application source – such as a file called `Chart.yaml` for a Helm based application. An example of how the discover property can be configured to support this use case using the `fileName` option is found below:

```
discover:
  fileName: "Chart.yaml"
```

Once a match has been made using any of the methods described above, the next step is to perform any initialization steps that are required by the plugin. This step is optional and only executed when the `init` property has been defined. When working with Helm based applications within the context of a Config Management Plugin, a common initialization task may involve the need to manage any of the dependencies that the chart relies on. This way, when the Chart is processed within the main logic as defined in the `generate` property, all of the necessary resources will be available. The following is an example of how Helm dependencies can be handled within the `init` property:

```
init:
  command: ["/bin/sh", "-c"]
  args: ["helm dependency build || true"]
```

Finally, after defining any of the key optional properties, the primary plugin logic as defined by the `generate` property can be specified. Instead of providing a simple code example like we demonstrated previously for the `init` and `discovery` properties, let's use this as an opportunity to look into how a Config Management Plugin could be used and implemented in practice.

Kustomize and Helm on their own provide a powerful set of capabilities for templating Kubernetes manifest. But, why choose one tool over the other when you can utilize both at the same time. Kustomize includes support for inflating Helm charts and the two tools, working hand-in-hand, provide a powerful combination that provides a number of benefits, particularly when working with Argo CD. For example, a common challenge when consuming Helm charts from the community is that customizations are limited to only the options that the chart creator provides. When used with Kustomize, the rendered charts can be augmented using any of the Kustomize features including patching and transformation.

The challenge, where a Config Management Plugin can be beneficial, is that an additional flag (`--enable-helm`) must be provided to the underlying `kustomize` command to enable support for the Helm inflator. Argo CD does provide support for customizing the Kustomize build options. However, these configurations are enabled globally within the argo-cd ConfigMap and there may be either a desire to avoid setting configurations globally or the inability to modify Argo CD configurations at a global level due to access limitations.

To enable the Helm inflator feature within the context of a Config Management Plugin, the following can be specified in the `generate` property.

```
generate:
  command: ["/bin/sh", "-c"]
  args: ["kustomize build —-enable-helm"]
```

Registering the Plugin

WIth the `generate` property now defined, we have all of the necessary steps to be able to utilize a ConfigManagementPlugin manifest. Now, while this resource may appear similar to a Kubernetes Custom Resource, it is just a configuration file that Argo CD understands. It is included within the plugin sidecar at a known location so that it can be discovered by the Argo CD server. The delivery of the file can be achieved using two methods:

- Inclusion within the image
- Injected at runtime as a ConfigMap

The injection method is preferred as the values contained within the configuration file may differ per environment which avoids having to build a new plugin image for

each variation. This approach also aligns with the principles of the twelve-factor app which emphasizes externalizing configurations within the operating environment – and in Kubernetes, this implies storage as a ConfigMap or Secret.

A ConfigMap containing the embedded `ConfigManagementPlugin` resource can be found in the `kustomize-helm-plugin.yml` file within the `ch11/configmanagement plugins` directory of the repository accompanying this book and also included below.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: kustomize-helm-plugin
  namespace: argocd
data:
  plugin.yaml: |
    apiVersion: argoproj.io/v1alpha1
    kind: ConfigManagementPlugin
    metadata:
      name: kustomize-helm
    spec:
      generate:
        command: ["/bin/sh", "-c"]
        args: ["kustomize build --enable-helm"]
```

Now, apply the ConfigMap to the `argocd` namespace using the following command from within the project repository directory:

```
kubectl apply -f ch11/configmanagementplugins/kustomize-helm-plugin.yml
```

Next, the plugin sidecar must be added to the Deployment of the repository server. The sidecar is represented by the following configuration:

```
containers:
  - name: kustomize-helm
    securityContext:
      runAsNonRoot: true
      runAsUser: 999
    image: registry.k8s.io/kustomize/kustomize:v4.5.7
    imagePullPolicy: IfNotPresent
    command: [/var/run/argocd/argocd-cmp-server]
    volumeMounts:
      - mountPath: /var/run/argocd
        name: var-files
      - mountPath: /home/argocd/cmp-server/plugins
        name: plugins
      - mountPath: /home/argocd/cmp-server/config/plugin.yaml
        subPath: plugin.yaml
        name: kustomize-helm-plugin
      - mountPath: /tmp
        name: cmp-tmp
volumes:
  - name: kustomize-helm-plugin
```

```
    configMap:
      name: kustomize-helm-plugin
  - emptyDir: {}
    name: cmp-tmp
```

While the definition of a Config Management Plugin sidecar can vary between each implementation, particularly as it relates to the associated image, there are certainly properties where their values must align to a certain set of rules as noted below:

- The sidecar must run as user `999` in order for the sidecar to access the files from the Application
- The `plugin.yaml` file must be located in the `/home/argocd/cmp-server/config` directory
- The Repository Server `Deployment` includes a series of volumes that should be mounted into the sidecar including `/var/run/argocd` which contains the `argocd-cmp-server` binary and `/home/argocd/cmp-server/plugins`

A patch file called `argocd-repo-server-kustomize-helm-plugin-patch.yaml` containing the sidecar definition above is also included in the `ch11/configmanagement plugins` directory of the repository accompanying this book.

Patch the repo-server `Deployment` by executing the following command:

```
kubectl -n argocd patch deployments/argo-cd-argocd-repo-server --patch-file
ch11/configmanagementplugins/argocd-repo-server-kustomize-helm-plugin-patch.yaml
```

With the patch applied, confirm that the updated repo-server `Deployment` now includes the *kustomize-helm-plugin* sidecar for a total of two running containers in the Pod.

```
$ kubectl get pods -n argocd -l=app.kubernetes.io/component=repo-server

NAME                                          READY   STATUS    RESTARTS   AGE
argo-cd-argocd-repo-server-9d947b457-pxs8l    2/2     Running   0          121m
```

Included in the `ch11/configmanagementplugins` directory are an additional set of assets that will be used to to demonstrate the use of the Helm inflator capability of Kustomize with an Argo CD Config Management plugin. First, the `charts` directory contains a simple Helm chart called *kustomize-helm* which produces a ConfigMap when rendered. And, as with any Kustomize application, there is also a Kustomization (`kustomization.yaml`) file present which invokes the Helm inflator using a set of properties prefixed with *helm*.

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

helmCharts:
  - name: kustomize-helm
```

```
      version: 0.1.0
      releaseName: kustomize-helm

  helmGlobals:
    chartHome: charts
```

The `helmCharts` property within the *Kustomization* file includes the majority of the configurations associated with the Helm inflator, such as the name of the chart and the version. Since the desired Helm chart is not in a location relative to the Kustomization file, the `chartHome` property within the `helmGlobals` property specifies where Helm charts should be sourced from. If a Helm chart is not available locally, they can originate from either a remote repository or an OCI registry.

To have Argo CD deploy the Kustomize based application within the Kubernetes cluster, create an Argo CD Application called *kustomize-helm* that is defined in a file called `kustomize-helm-app.yaml` within the `ch11/configmanagementplugins` directory.

```
kubectl apply -f ch11/configmanagementplugins/kustomize-helm-app.yaml
```

Using either the Argo CD CLI or the user interface, check on the status of the newly created Application.

Notice that the kustomize-helm Application is reporting an error with a message similar to the following:

```
Failed to load target state: failed to generate manifest for source 1 of 1: rpc
error: code = Unknown desc = `kustomize build <path to cached source>/ch11/con-
figmanagementplugins` failed exit status 1: Error: trouble configuring builtin
HelmChartInflationGenerator with config: ` chartHome: charts name: kustomize-
helm version: 0.1.0 `: must specify --enable-helm
```

The error message indicates that it is unable to render the Kustomize application as even though the Helm inflator capability is being used, it is not being enabled by including the `--enable-helm` flag.

Recall the two ways that an Argo CD Config Management Plugin can be triggered: either through dynamic activation or specified explicitly within the Application itself. Since neither option was used, the error being displayed is expected as the Helm inflator feature in Kustomize is not enabled by default in Argo CD.

To enable the Config Management Plugin that we configured previously, update the *kustomize-helm* `Application` to specify the name of the plugin within the `.spec.source` property using kubectl, the Argo CD CLI or the Argo CD user interface:

```
spec:
  source:
    plugin:
      name: kustomize-helm
```

Once the configuration of the Application has been updated, the previously seen error will be resolved and the Application will synchronize successfully.

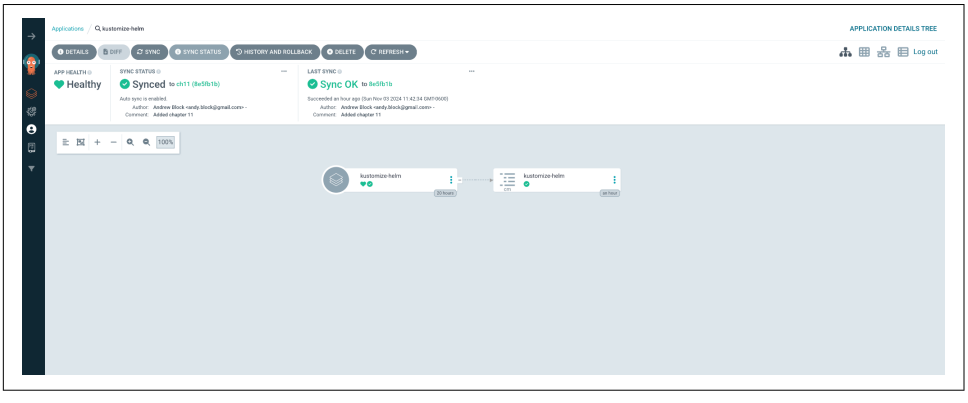


Figure 10-1. The kustomize-helm Application in the Argo CD user interface

If you investigate the contents of the *kustomize-helm* `ConfigMap` that was created from the Application, two properties are present:

```
apiVersion: v1
kind: ConfigMap
metadata:
  labels:
    app.kubernetes.io/instance: kustomize-helm
    app.kubernetes.io/managed-by: Helm
    app.kubernetes.io/name: kustomize-helm
    argocd.argoproj.io/instance: kustomize-helm
    helm.sh/chart: kustomize-helm-0.1.0
  name: kustomize-helm
  namespace: kustomize-helm
data:
  baseValue: Base Value
  specialValue: Added by Kustomize
```

The `baseValue` property is included by default from the kustomize-helm Helm chart. However, the `specialValue` property was added dynamically as a patch by Kustomize as defined in the `kustomization.yaml`:

```
patches:
- patch: |-
    apiVersion: v1
    kind: ConfigMap
    metadata:
      name: kustomize-helm
    data:
      specialValue: "Added by Kustomize"
```

The combination of Helm and Kustomize which is enabled as an opt-in capacity illustrates the benefits that are provided from a Config Management Plugin.

Customizing Plugin Execution

The execution of a Config Management Plugin can be enhanced at an `Application` level to curate their operation. They provide the end user both the ability to specify additional configurations at an Application level, but also awareness that certain options might be available to them. Two approaches of configuration are available:

- Environment Variables
- Parameters

Both of these methods are then exposed to plugins and it is the responsibility of the plugin author to handle the inputs accordingly.

Environment Variables

Environment variables are the primary method for which Config Management Plugins glean information about the operating environment and can originate from a variety of system and user defined sources. Much of the same information is also made available and utilized by the standard build tools, like Helm and Kustomize, and include the following:

- Operating System level environment variables from within the plugin sidecar
- Build environment variables including `ARGOCD_APP_NAME`, `ARGOCD_APP_NAMESPACE`, and `KUBE_VERSION`. The full list can be found at *https://argo-cd.readthedocs.io/en/stable/user-guide/build-environment*

In addition to the system defined environment variables, end users can explicitly specify their own set of environment variables within the `env` property of the `.spec.source.plugin` field as shown below:

```
spec:
  source:
    plugin:
      env:
        - name: FOO
          value: bar
```

User defined environment variables are prefixed with `ARGOCD_ENV_`. So, the value of the user defined environment variable above would be accessible within the plugin in the environment variable `ARGOCD_ENV_FOO`.

Parameters

Another method for customizing the execution of a Config Management Plugin is through the use of parameters. Parameters are also defined in the `.spec.source.plugin` field of an `Application` in the `parameters` property and they have several advantages when compared to environment variables:

- Support multiple data types aside from strings (string, array, or map are the supported data types)
- "Announced" within the Parameters tab of the Application within the User Interface
- "Announced" parameters can either be statically or dynamically defined within the `ConfigManagamentPlugin` manifest

Parameters are also exposed to plugins as environment variables and available in two formats:

- Individually with the prefix `PARAM_`. A parameter with the name `example-param` would be exposed as the environment variable `PARAM_EXAMPLE_PARAM`.
- A single `ARGOCD_APP_PARAMETERS` environment variable containing the content of the Application .spec.source.plugin field in JSON format.

Complex parameter types, such as arrays or maps, have a slightly different environment variable name format. For arrays, the environment variable is suffixed with the index (`PARAM_NAME_X` where `X` is the index) of the parameter while maps are suffixed with the key associated with the parameter (foo.bar becomes `PARAM_NAME_FOO_BAR`).

Aside from supporting more complex data types, another strength of plugin parameters is that they can be "announced" within the Argo CD user interface giving end users the awareness of specific parameters as well as the ability for parameters to be defined. The following schema defines how parameters can be exposed (announced):

```
# Name of the parameter
name: string-param
# Description of the parameter
title: ""
Tooltip shown when the user hovers over the field in the user interface
tooltip: ""
# Indicator for whether a parameter is required.
required: false
# Indicator for how the user interface should present the entry field (defaults
to "string")
itemType: ""
# Data type for non string values (map or array)
collectionType: ""
```

```
    # Optional default value
    string: default-string-value
```

Parameters that are consistent (static) for each execution of a particular plugin are announced in the `.spec.parameters.static` property of the `ConfigManagementPlugin`.

To illustrate how parameters are presented in the Argo CD user interface, define a parameter called *my-static-param* within the *kustomize-helm-plugin* `ConfigMap` containing the `ConfigManagementPlugin` as shown below:

```
spec:
  parameters:
    static:
      - name: my-static-param
        title: Example static parameter
```

Once applied, restart the repo-server pod to enable Argo CD to pick up on changes.

```
kubectl delete pod -n argocd -l=app.kubernetes.io/name=argocd-repo-server
```

With the repo-server restarted, navigate to the Argo CD user interface and select the *kustomize-helm* Application. Click on the **Details** button and then navigate to the **Parameters** tab.

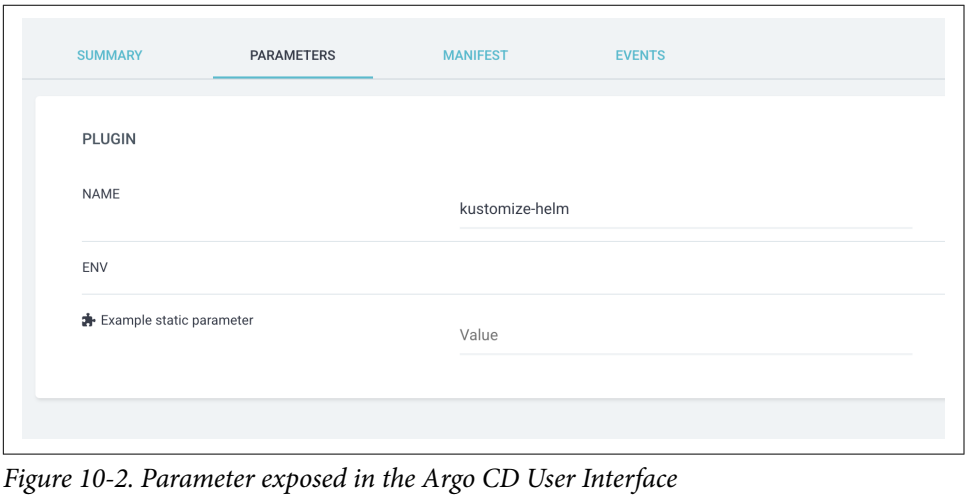


Figure 10-2. Parameter exposed in the Argo CD User Interface

Notice that the *my-static-param* parameter with the title "*Example static parameter*" as configured in the *kustomize-helm-plugin* `ConfigMap` is now available on the page as a field to specify.

It is important to note that even though parameters are exposed to the user interface, they do not become defined as environment variables for use by Config Management

Plugins until their values are specified either in the user interface or declaratively in the Application manifest.

Alternatively, instead of explicitly specifying parameters within the ConfigManagementPlugin, they can be sourced dynamically from the content within the Application source code. The use of dynamic parameters offloads responsibility for defining plugin parameters that are exposed within the Argo CD user interface from the Argo CD administrator as well as enabling parameters to be defined based on the content source associated with each Application. Dynamic parameters are defined within the `.spec.parameters.dynamic` property of the `ConfigManagementPlugin` which specifies a command that should be executed within the Application source which generates a structure representing the structure of static parameters in JSON format.

User Interface Customization

One of the primary reasons why Argo CD has gained such popularity in the Kubernetes community is due to its rich user interface. By simplifying the steps that a user needs to take to become productive as well as presenting an easy to understand visualization of the current state of GitOps based deployments and operations accelerates adoption and management concerns. In order to enable further productivity with the user interface, Argo CD provides several injection points for end users to customize the look and feel as well as to extend the baseline featureset. This section will highlight several of the available methods.

Banner Notifications

Proactive communication is one of the methods that can be used to enhance the overall experience for end users. One way that Argo CD supports this goal is through the use of banner notifications. When enabled, these messages, defined at a global level by Argo CD administrators, allows for important information to be presented to end users, such as upcoming maintenance periods or new features that are available on the platform. This feature is enabled by setting the `ui.bannercontent` property of the *argocd-cm* `ConfigMap` with the desired content. Additional options, such as the location of where the banner should appear, is set by specifying the `ui.banner positon` to be either "top" or "bottom" as well as whether the banner should be permanently displayed using the `ui.bannerpermanent` property. Finally, the text provided in `ui.bannercontent` property can also include a hyperlink to another location, such as a maintenance page when notifying users of upcoming changes to the environment. This option is set by specifying the `ui.bannerurl` property.

The following illustrates how a banner notification appears within the Argo CD user interface.

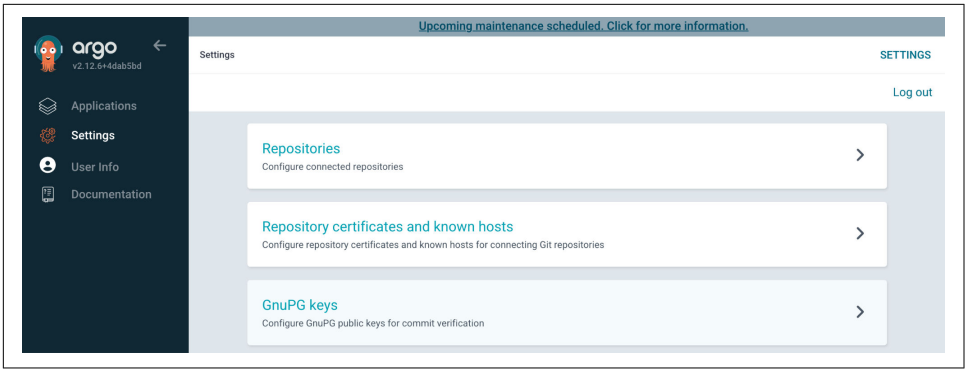


Figure 10-3. Notification banner displayed in the Argo CD User Interface

Custom Styles

Integral to any user experience is how content is presented and in modern web applications, the look and feel is driven primarily by Cascading Style Sheets (CSS). These resources are included as part of the argo-ui project (*https://github.com/argoproj/argo-ui*) and the Argo CD user interface leverages many of these elements when presenting content to the end user.

As Argo CD usage continues to expand to different environments and in enterprise organizations, there may be a desire to customize how some of the elements are presented. The Argo CD user interface supports including custom CSS content in order to supplement the baseline set of content provided by the argo-ui project. Examples of common customizations include replacing the Argo CD logo with a custom logo or setting the background of certain components to represent the operating environment that Argo CD is managing.

Custom stylesheets can be applied either by specifying the location of resources from a remote URL or from a location within the *argocd-server* container using the `ui.cssurl` property of the *argocd-cm* `ConfigMap`. For example, to reference an externally hosted CSS file from a remote resource, set the `ui.cssurl` property using the following format:

```
ui.cssurl: "https://www.example.com/my-styles.css"
```

One of the common uses for customizing the Argo CD user interface, as described previously, is to change background elements to represent the environment that Argo CD is managing. This small enhancement gives end users an extra level of assurance, particularly when multiple Argo CD instances have been deployed.

The `argocd-server` Deployment that was installed using the Argo CD Helm chart includes an optional volume mount leveraging a *ConfigMap* called `argocd-styles-cm` containing custom CSS styles to the location `/shared/app/custom` within the

container. This ConfigMap is not included in the set of resources when Argo CD is deployed, and since the volume is marked as *optional*, the container can start without any issue. If an alternate installation method was chosen and the volume for setting up mounting custom styles to the Argo CD Server container was not configured, a `volume` and associated `volumeMount` can be applied to the *argocd-server* `Deployment`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: argocd-server
  ...
spec:
  template:
    ...
    spec:
      containers:
      - command:
        ...
        volumeMounts:
        ...
        - mountPath: /shared/app/custom
          name: styles
      ...
      volumes:
      ...
      - configMap:
          name: argocd-styles-cm
        name: styles
```

To implement the use case for changing the background element of the Argo CD user interface, we can embed the custom CSS content within the `argocd-styles-cm` *ConfigMap* to achieve the desired goal.

The following CSS properties can be used to update the topbar of the Argo CD user interface to be the color red, potentially indicating that the Argo CD instance represents a production environment.

```
div.columns.small-9.top-bar__left-side {
    background: #fefefe;
}
div.columns.top-bar__left-side,
div.top-bar__title.text-truncate.top-bar__right-side {
    background: #EE0000;
    color: #fff;
}
.top-bar__breadcrumbs {
    color: #fff !important;
}
.top-bar__title {
    color: #fff !important;
```

}

The `argocd-styles-cm.yaml` file within the `ch11/ui` directory of the project repository contains the updated ConfigMap with the CSS classes illustrated previously already included.

Apply the changes to the ConfigMap by running the following command from within the project directory:

```
kubectl apply -f ch11/ui/argocd-styles-cm.yaml
```

Restart the argocd-server pod so that the changes to the ConfigMap can be picked up:

```
kubectl delete pod -n argocd -l=app.kubernetes.io/component=server
```

Once the pod is running and ready, reload the User Interface. Notice that the toolbar is now red confirming that the changes specified are being used.

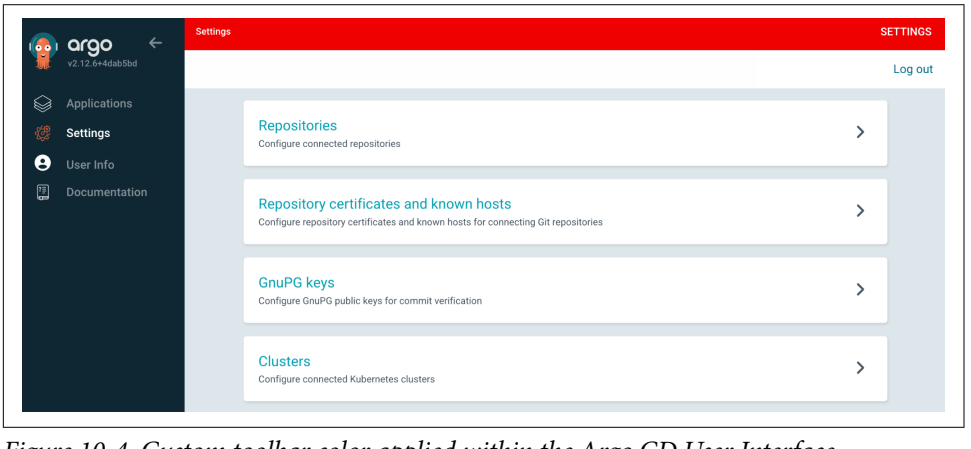


Figure 10-4. Custom toolbar color applied within the Argo CD User Interface

While modifying the toolbar is just a minor change, it just illustrates the potential options available for customizing the style of the Argo CD user interface.

UI Extensions

Not only can the look and feel of the Argo CD User Interface be customized, but entirely new elements can be added through the use of UI Extensions. Since the Argo CD User Interface is React based, Extensions are delivered as React Components within JavaScript files matching the pattern `extensions*.js` from within the `/tmp/extensions` directory of the *argocd-server* Pod.

Three types of UI extensions are available:

Resource Tab Extensions:
: Provides an additional tab within the sliding panel on the Argo CD Application details page

System Level Extensions:
: Adds new items to the sidebar that displays a new page with content with selected

Application Status Panel Extensions:
: Adds new items to the status panel of an Application

Extensions are registered using the exposed extensionsAPI global variable. Each extension type provides its own registration method along with a series of method parameters. For example, to register a System Level Extension, the following method is used

```
registerSystemLevelExtension(component: ExtensionComponent, title: string,
options: {icon?: string})
```

With a basic understanding of Argo CD UI Extensions including the types that can be defined, let's walk through the steps that it takes to create and implement a System Level Extension.

A System Level Extension, once again, exposes a link on the sidebar to a dedicated page with content. The following is the JavaScript that is needed to create a minimal extension:

```
((window) => {
  const component = () => {
    return React.createElement(
      "div",
      { style: { padding: "10px" } },
      "Argo CD Up and Running"
    );
  };
  window.extensionsAPI.registerSystemLevelExtension(
    component,
    "Argo CD Book",
    "/argocd-book",
    "fa-book"
  );
})(window);
```

When added to Argo CD, the User Interface will contain a new link called "Argo CD Book" with a book icon (Using a book icon from the content library https://fontawesome.com) that presents a page (component) with a simple line of text. Notice how the extension is registered to Argo Using the `registerSystemLevelExtension` method of the `extensionsAPI`.

There are two methods that UI extensions are typically delivered to the argocd-server Pod:

- Mounted as a Volume
- Loaded dynamically using the Argo CD Extension Installer Project (https://github.com/argoproj-labs/argocd-extension-installer)

In our case, we will use the former strategy and inject the extension within a ConfigMap as a volume. The ConfigMap containing the extension can be found in a file called `ui-extensions.yaml` within the `ch11/ui` directory of the project repository.

Create the ConfigMap by running the following command from the project repository directory:

```
kubectl apply -f ch11/ui/ui-extensions.yaml
```

Next, update the *argocd-server* Deployment with the contents of the `argo-cd-server-ui-extensions.yaml` file within the `ch11/ui` directory that will include the *ui-extensions* ConfigMap that will be mounted within the /tmp/extensions directory of the container by executing the following command:

```
kubectl apply -f ch11/ui/argo-cd-argocd-server.yaml --server-side=true
```

Wait until the server pod has restarted and becomes ready. Navigate to the User Interface and verify the new link exposed by the extension is present on the sidebar.

Figure 10-5. CAPTION

Clicking on the "Argo CD Book" link will present the minimal amount of content that was provided in the extension, but can easily be expanded upon as desired.

While this walk through provided a glimpse into the power provided by Argo CD UI Extensions, more fully featured extensions are available. One such example from the Argo Labs project is the ArgoCD Extension Metrics which exposes Prometheus metrics on the Resources tab of the user interface. It is projects, like ArgoCD Extension Metrics, that illustrate just how extensible Argo CD user interface has become.

Summary

In this chapter, we covered some of the ways that the base capabilities provided by Argo CD can be extended by end users. We first explored how Config Management

Plugins enable complete control for how manifests are rendered by Argo CD including how they are configured using a *ConfigManagementPlugin* and implemented as a sidecar to the Argo CD Repository Server. Then, we looked at the Argo CD user interface and how the look and feel can be customized through the use of Banner notifications and custom CSS styles. Finally, we saw how UI extensions allow end users to add elements, including custom components, at either the System, Resource or Application Status level, to extend the baseline set of capabilities that the Argo CD User Interface provides.

About the Authors

Andrew Block is a Distinguished Architect at Red Hat who works with organizations throughout the world to design and implement solutions leveraging cloud native technologies. He specializes in embracing security at every phase of the Software Development Lifecycle and delivering software in a repeatable and consistent manner. Andrew has authored several publications related to the cloud native ecosystem including *Managing Kubernetes Resources Using Helm* and *Kubernetes Secrets Management* in order to share his knowledge with others. He holds several roles in the Open Source community and is a core maintainer of Helm, a package manager for Kubernetes.

Christian Hernandez is a well-rounded technologist with experience in infrastructure engineering, systems administration, enterprise architecture, tech support, advocacy, and product management. Passionate about OpenSource and containerizing the world one application at a time. He is currently a maintainer of the OpenGitOps project, a member of the Argo Project, and currently is Head of Community at Akuity. He focuses on GitOps practices, DevOps, Kubernetes, and Containers.