Terraform changes

The `terraform` directory now contains two more files:

- `argocd.tf`: This contains the Terraform configuration for deploying Argo CD
- `app.tf`: This contains the Terraform configuration for configuring Argo CD apps

Let's explore both files in detail.

argocd.tf

This files starts with the `time_sleep` resource with an explicit dependency on the `google_container_cluster` resource. It will sleep for 30 seconds after the cluster is created so that it is ready to serve requests:

```
resource "time_sleep" "wait_30_seconds" {
  depends_on = [google_container_cluster.main]
  create_duration = "30s"
}
```

To connect with GKE, we will use the `gke_auth` module provided by `terraform-google-modules/kubernetes-engine/google//modules/auth`. We will add an explicit dependency to the `time_sleep` module so that authentication happens 30 seconds after the cluster is created:

```
module "gke_auth" {
  depends_on           = [time_sleep.wait_30_seconds]
  source               = "terraform-google-modules/kubernetes-engine/google//modules/auth"
  project_id           = var.project_id
  cluster_name         = google_container_cluster.main.name
  location             = var.location
  use_private_endpoint = false
}
```

Now that we've authenticated with the GKE cluster, we need to apply manifests to deploy Argo CD to the cluster. We will use the `gavinbunney/kubectl` plugin (`https://registry.terraform.io/providers/gavinbunney/kubectl/latest/docs`) for that.

We start by defining some data sources to help generate Kubernetes manifests that we will apply to install Argo CD. We will create two `kubectl_file_documents` data sources for the namespace and Argo CD app that point to the corresponding `namespace.yaml` and `install.yaml` files within the `manifests/argocd` directory:

```
data "kubectl_file_documents" "namespace" {
  content = file("../manifests/argocd/namespace.yaml")
}
data "kubectl_file_documents" "argocd" {
  content = file("../manifests/argocd/install.yaml")
}
```

Using these data sources, we can create two `kubectl_manifest` resources for the namespace and Argo CD app. These resources will apply the manifests within the GKE cluster:

```
resource "kubectl_manifest" "namespace" {
  for_each  = data.kubectl_file_documents.namespace.manifests
  yaml_body = each.value
  override_namespace = "argocd"
}
resource "kubectl_manifest" "argocd" {
  depends_on = [
    kubectl_manifest.namespace,
  ]
  for_each  = data.kubectl_file_documents.argocd.manifests
  yaml_body = each.value
  override_namespace = "argocd"
}
```

Now that we've added the configuration to install Argo CD, we also need to configure argo CD Applications. To do that, we have the `app.tf` file.

app.tf

Similar to the Argo CD configuration, we have a `kubectl_file_documents` data source reading from the `manifests/argocd/apps.yaml` file; the `kubectl_manifest` resource will apply the manifest to the Kubernetes cluster:

```
data "kubectl_file_documents" "apps" {
    content = file("../manifests/argocd/apps.yaml")
}
resource "kubectl_manifest" "apps" {
  depends_on = [
    kubectl_manifest.argocd,
  ]
  for_each  = data.kubectl_file_documents.apps.manifests
  yaml_body = each.value
  override_namespace = "argocd"
}
```

We've also modified the `provider.tf` file, so we'll explore that next.

provider.tf

Within this file, we have included the `kubectl` provider, as follows:

```
...
provider "kubectl" {
  host                   = module.gke_auth.host
  cluster_ca_certificate = module.gke_auth.cluster_ca_certificate
  token                  = module.gke_auth.token
  load_config_file       = false
```

```
}
terraform {
  required_providers {
    kubectl = {
      source  = "gavinbunney/kubectl"
      version = ">= 1.7.0"
    }
  }...
}
```

Now, let's inspect the manifests directory.

The Kubernetes manifests

The manifests directory contains Kubernetes manifests that we will apply to the Kubernetes cluster. As we're setting up Argo CD first, it only contains the `argocd` directory at the moment; however, we will extend this to add further directories later in this chapter.

The `manifests/argocd` directory contains the following files:

- `namespace.yaml`: The manifest to create the `argocd` namespace where Argo CD will run.
- `install.yaml`: The manifest to create the Argo CD application. The manifest is downloaded from the official Argo CD release URL.
- `apps.yaml`: This contains an Argo CD **ApplicationSet** configuration.

While the `namespace.yaml` and `install.yaml` files are self-explanatory, let's discuss the `apps.yaml` file and the Argo CD ApplicationSet resource in more detail.

Argo CD Application and ApplicationSet

To manage applications declaratively, Argo CD uses the **Application** resource. An Application resource defines the configuration required for Argo CD to access Kubernetes deployment configuration stored in the Git repository using the `source` attribute and where it needs to apply them using the `target` attribute. An Application resource caters to one application. For example, to deploy our Blog App, we will need to create an Application resource like the following:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: blog-app
  namespace: argocd
spec:
  project: default
```

```
  source:
    repoURL: https://github.com/<your_github_repo>/mdo-environments.git
    targetRevision: HEAD
    path: manifests/nginx
  destination:
    server: https://kubernetes.default.svc
  syncPolicy:
    automated:
      selfHeal: true
```

This manifest defines an Argo CD Application resource with the following sections:

- `project`: We can organize applications into distinct projects. In this case, we will stick to the `default` project.
- `source`: This section defines the configuration Argo CD requires to track and pull the application configuration from the Git repository. It typically contains `repoURL`, `targetRevision`, and the `path` value where the application manifests are located.
- `destination`: This section defines the `target` value to which we want to apply the manifest. It typically contains the `server` section and contains the Kubernetes cluster's URL.
- `syncPolicy`: This section defines any policies Argo CD should apply while syncing the Blog App from the Git repository and what to do when it detects a drift. In the preceding configuration, it would try to correct any drift from the Git repository automatically as `selfHeal` is set to `true`.

We can very well go ahead and define multiple application manifests for each application. However, for larger projects, it might turn out to be an overhead. To manage this, Argo CD provides a generic way of creating and managing applications via the `ApplicationSet` resource.

The `ApplicationSet` resource provides us with a way to dynamically generate Application resources by using a defined pattern. In our case, we have the following structure:

```
manifests
└── argocd
|   ├── apps.yaml
|   ├── install.yaml
|   └── namespace.yaml
└── blog-app
|   └── manifest.yaml
└── <other-app>
    └── manifest.yaml
```

So, logically, for every subdirectory of the `manifests` directory, we would need to create a new application with the directory name. The respective application configuration should source all manifests from the subdirectory.

We've defined the following `ApplicationSet` within the `apps.yaml` file:

```
apiVersion: argoproj.io/v1alpha1
kind: ApplicationSet
metadata:
  name: argo-apps
  namespace: argocd
spec:
  generators:
  - git:
      repoURL: https://github.com/<your_github_repo>/mdo-environments.git
      revision: HEAD
      directories:
      - path: manifests/*
      - path: manifests/argocd
        exclude: true
  template:
    metadata:
      name: '{{path.basename}}'
    spec:
      project: default
      source:
        repoURL: https://github.com/<your_github_repo>/mdo-environments.git
        targetRevision: HEAD
        path: '{{path}}'
      destination:
        server: https://kubernetes.default.svc
      syncPolicy:
        automated:
          selfHeal: true
```

`ApplicationSet` has the following sections:

- `generators`: This section defines how Argo CD should generate Application resources. We've used the `git` generator, which contains the `repoURL`, `revision`, and `directories` sections. The `directories` section defines the directory from where we would want to source our applications. We've set that to `manifests/*`. So, it will look for every subdirectory within the `manifests` directory. We have also defined an exclude directory called `manifests/argocd`. This is because we don't want Argo CD to manage the configuration to deploy itself.

- `templates`: This section defines the template for creating the application. As we can see, the contents are very similar to an Application resource definition. For `metadata.name`, we specified `{{path.basename}}`, which means it will create Application resources with the subdirectory name as we intended. The `template.spec.source.path` attribute contains the source path of the corresponding application manifests, so we've set that to `{{path}}` – that is, the subdirectory. So, we will have `blog-app` and `<other-app>` applications based on the preceding directory structure. The rest of the attributes are the same as those for the Application resource we discussed previously.

Now that we've configured everything we need to install and set up Argo CD, let's commit and push this configuration to the remote repository by using the following commands:

```
$ git add --all
$ git commit -m "Added argocd configuration"
$ git push
```

We will see that GitHub will run the Actions workflow on update and deploy Argo CD. Once the workflow is successful, we can go ahead and access the Argo CD Web UI.

Accessing the Argo CD Web UI

Before we can access the Argo CD Web UI, we must authenticate with the GKE cluster. To do so, run the following command:

```
$ gcloud container clusters get-credentials \
 mdo-cluster-dev --zone us-central1-a --project $PROJECT_ID
```

To utilize the Argo CD Web UI, you will require the external IP address of the `argo-server` service. To get that, run the following command:

```
$ kubectl get svc argocd-server -n argocd
NAME          TYPE        EXTERNAL-IP  PORTS          AGE
argocd-server LoadBalaner 34.122.51.25 80/TCP,443/TCP 6m15s
```

We now know that Argo CD is accessible at `https://34.122.51.25/`. Upon visiting this link, you'll notice that username and password are required for authentication:

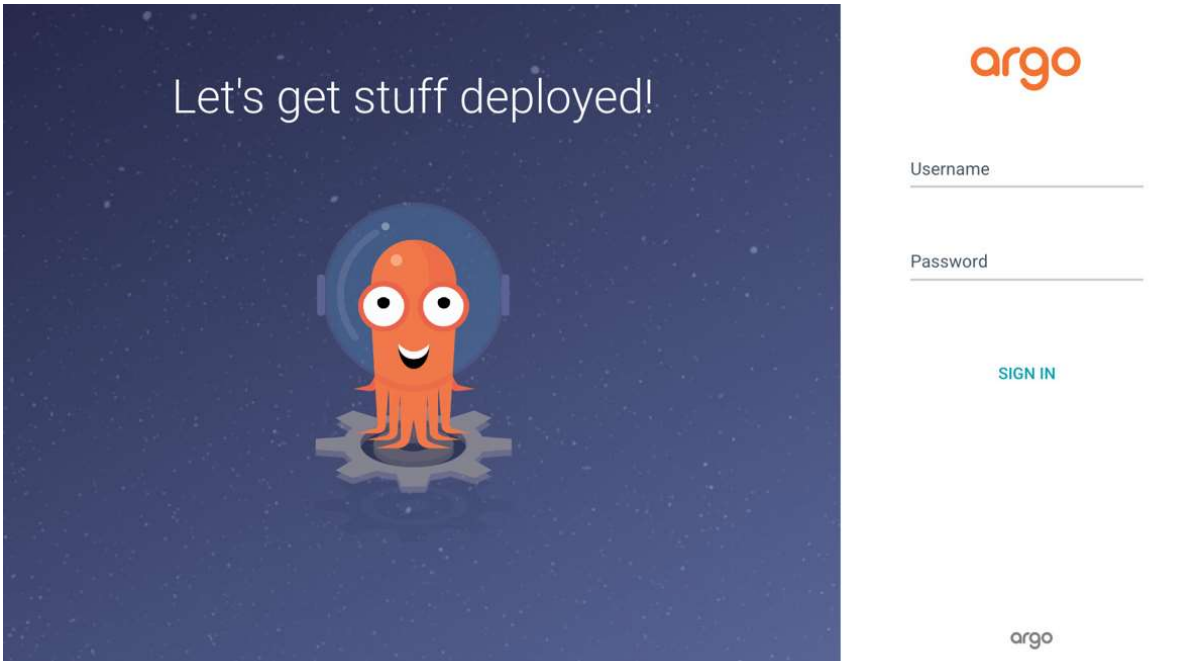

Figure 12.10 – Argo CD Web UI – login page

Argo CD provides an initial `admin` user by default, and the password for this user is stored in the `argocd-initial-admin-secret` **Secret** resource as plaintext. While you can use this default setup, it's worth noting that it is generated from the publicly available YAML manifest. Therefore, it's advisable to update it. To do so, execute the following command:

```
$ kubectl patch secret argocd-secret -n argocd \
-p '{"data": {"admin.password": null, "admin.passwordMtime": null}}'
$ kubectl scale deployment argocd-server --replicas 0 -n argocd
$ kubectl scale deployment argocd-server --replicas 1 -n argocd
```

Now, allow two minutes for the new credentials to be generated. After that, execute the following command to retrieve the password:

```
$ kubectl -n argocd get secret argocd-initial-admin-secret \
-o jsonpath="{.data.password}" | base64 -d && echo
```

Now that you have the necessary credentials, log in and you will see the following page:

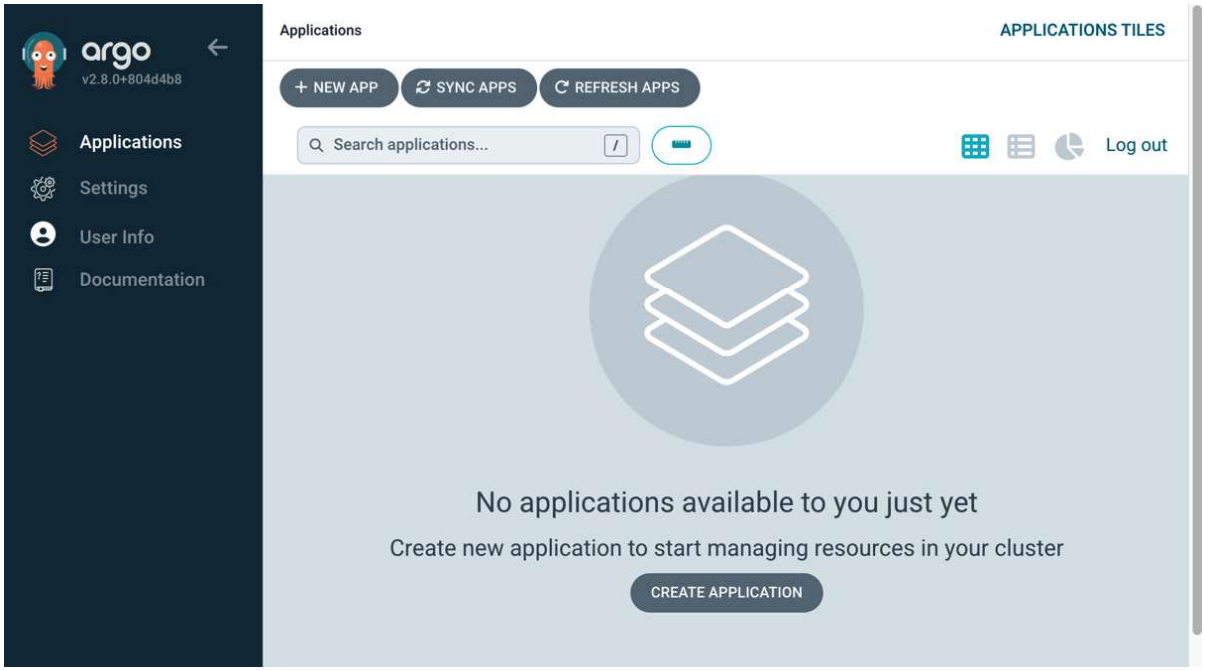

Figure 12.11 – Argo CD Web UI – home page

We've successfully set up Argo CD. The next step is to deploy our application; however, as we know that our application uses Kubernetes Secrets, which we cannot store on Git, we will have to find a mechanism to store it securely. To solve that problem, we have Bitnami's **SealedSecret** resource. We'll look at that in the next section.

Managing sensitive configurations and Secrets

Sealed Secrets solves the problem of *I can manage all my Kubernetes config in Git, except Secrets*. Sealed Secrets function as secure containers for your sensitive information. When you require a storage solution for secrets, such as passwords or keys, you place them in these specialized packages. Only the Sealed Secrets controller within Kubernetes can unlock and access the contents. This ensures the utmost security and protection for your valuable secrets. Created by *Bitnami Labs* and open sourced, they help you encrypt your Kubernetes Secrets into Sealed Secrets using asymmetric cryptography that only the Sealed Secrets controller running on the cluster can decrypt. This means you can store the Sealed Secrets in Git and use GitOps to set up everything, including Secrets.

Sealed Secrets comprises two components:

- A client-side utility called `kubeseal` helps us generate Sealed Secrets from standard Kubernetes Secret YAML

- A cluster-side Kubernetes controller/operator unseals your secrets and provides the key certificate to the client-side utility

The typical workflow when using Sealed Secrets is illustrated in the following diagram:

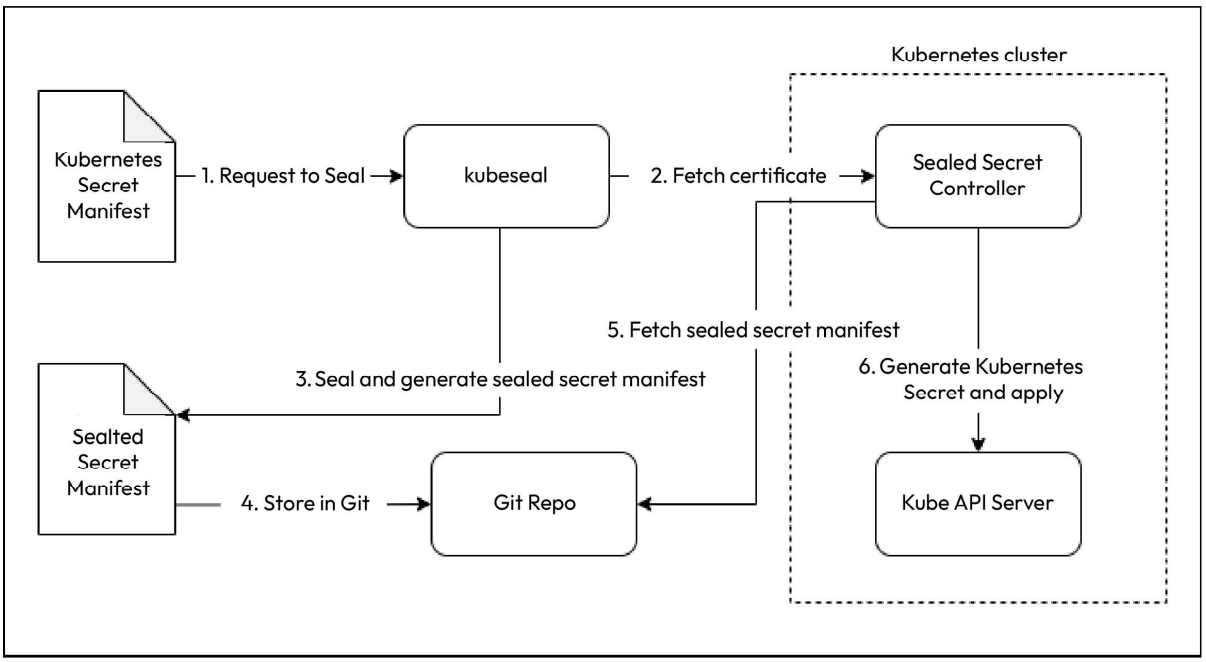

Figure 12.12 – Sealed Secrets workflow

Now, let's go ahead and install the Sealed Secrets operator.

Installing the Sealed Secrets operator

To install the **Sealed Secrets operator**, all you need to do is download the controller manifest from the latest release at `https://github.com/bitnami-labs/sealed-secrets/releases`. At the time of writing this book, `https://github.com/bitnami-labs/sealed-secrets/releases/download/v0.23.1/controller.yaml` is the latest controller manifest.

Create a new directory called `sealed-secrets` within the `manifest` directory and download `controller.yaml` using the following commands:

```
$ cd ~/mdo-environments/manifests & mkdir sealed-secrets
$ cd sealed-secrets
$ wget https://github.com/bitnami-labs/sealed-secrets\
/releases/download/v0.23.1/controller.yaml
```

Then, commit and push the changes to the remote repository. After about five minutes, Argo CD will create a new application called **sealed-secrets** and deploy it. You can visualize this in the Argo CD Web UI as follows:

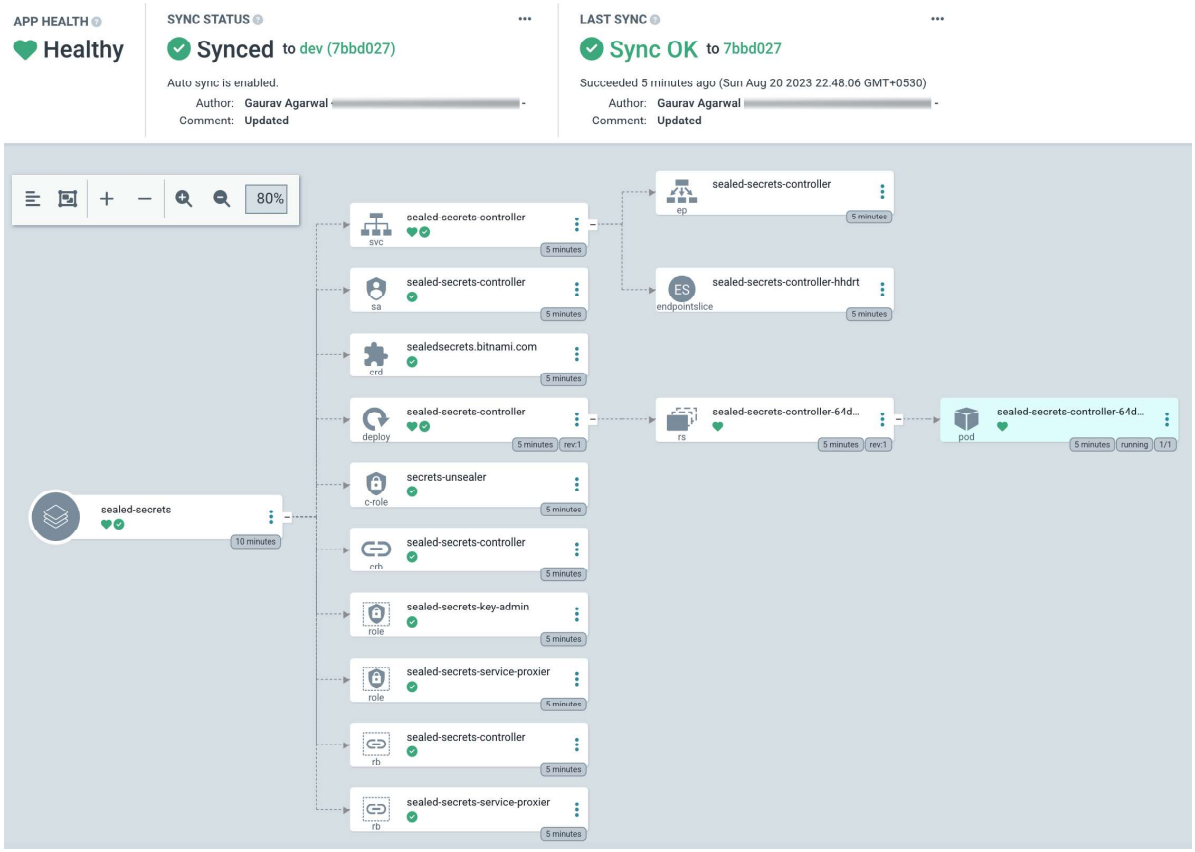

Figure 12.13 – Argo CD Web UI – Sealed Secrets

In the Kubernetes cluster, `sealed-secrets-controller` will be visible in the `kube-system` namespace. Run the following command to check this:

```
$ kubectl get deployment -n kube-system sealed-secrets-controller
NAME                     READY UP-TO-DATE AVAILABLE AGE
sealed-secrets-controller 1/1   1          1         6m4s
```

As we can see, the controller is running and ready. We can now install the client-side utility – `kubeseal`.

Installing kubeseal

To install the client-side utility, you can go to `https://github.com/bitnami-labs/sealed-secrets/releases` and get the `kubeseal` installation binary link from that page. The following commands will install `kubeseal 0.23.1` on your system:

```
$ KUBESEAL_VERSION='0.23.1'
$ wget "https://github.com/bitnami-labs/sealed-secrets/releases/download\
/v${KUBESEAL_VERSION:?}/kubeseal-${KUBESEAL_VERSION:?}-linux-amd64.tar.gz"
$ tar -xvzf kubeseal-${KUBESEAL_VERSION:?}-linux-amd64.tar.gz kubeseal
$ sudo install -m 755 kubeseal /usr/local/bin/kubeseal
$ rm -rf ./kubeseal*
```

To check whether `kubeseal` has been installed successfully, run the following command:

```
$ kubeseal --version
kubeseal version: 0.23.1
```

Since `kubeseal` has been installed, let's go ahead and create a Sealed Secret for the `blog-app`.

Creating Sealed Secrets

To create a Sealed Secret, we have to define the Kubernetes Secret resource. The `mongodb-creds` Secret should contain some key-value pairs with the `MONGO_INITDB_ROOT_USERNAME` key with a value of `root` and the `MONGO_INITDB_ROOT_PASSWORD` key with any value you want as the password.

As we don't want to store the plaintext Secret as a file, we will first create the Kubernetes secret manifest called `mongodb-creds` using the `--dry-run` and `-o yaml` flags and then pipe the output directly to `kubeseal` to generate the `SealedSecret` resource using the following command:

```
$ kubectl create secret generic mongodb-creds \
  --dry-run=client -o yaml --namespace=blog-app \
  --from-literal=MONGO_INITDB_ROOT_USERNAME=root \
  --from-literal=MONGO_INITDB_ROOT_PASSWORD=<your_pwd> \
  | kubeseal -o yaml > mongodb-creds-sealed.yaml
```

This generates the `mongodb-creds-sealed.yaml` Sealed Secret, which looks like this:

```
apiVersion: bitnami.com/v1alpha1
kind: SealedSecret
metadata:
  name: mongodb-creds
```

```
  namespace: blog-app
spec:
  encryptedData:
    MONGO_INITDB_ROOT_PASSWORD: AgB+tyskf72M/…
    MONGO_INITDB_ROOT_USERNAME: AgA95xKJg8veOy8v/…
  template:
    metadata:
      name: mongodb-creds
      namespace: blog-app
```

As you can see, the Sealed Secret is very similar to the Secret manifest. Still, instead of containing a Base64-encoded secret value, it has encrypted it so that only the Sealed Secrets controller can decrypt it. You can easily check this file into source control. Let's go ahead and do that. Move the Sealed Secret YAML file to the `manifests/blog-app` directory using the following command:

```
$ mkdir -p ~/mdo-environments/manifests/blog-app/
$ mv mongodb-creds-sealed.yaml ~/mdo-environments/manifests/blog-app/
```

Now that we've successfully generated the Sealed Secret and moved it to the `manifests/blog-app` directory, we'll set up the rest of our application in the next section.

Deploying the sample Blog App

To deploy the sample Blog App, we need to define application resources. We've already discussed what our app is composed of. We have defined the application bundle as a Kubernetes manifest file called `blog-app.yaml`. We need to copy this YAML to the `manifests/blog-app` directory using the following command:

```
$ cp ~/modern-devops/ch12/blog-app/blog-app.yaml \
 ~/mdo-environments/manifests/blog-app/
```

I've prebuilt the microservices and used the required `git-sha` as the tag, as we did in the previous chapter. You can edit the YAML and replace it with your image for each application.

Once done, commit and push the changes to the `mdo-environments` repository.

As soon as you push the changes, you should notice that the `blog-app` application starts appearing in the Argo CD UI in less than five minutes:

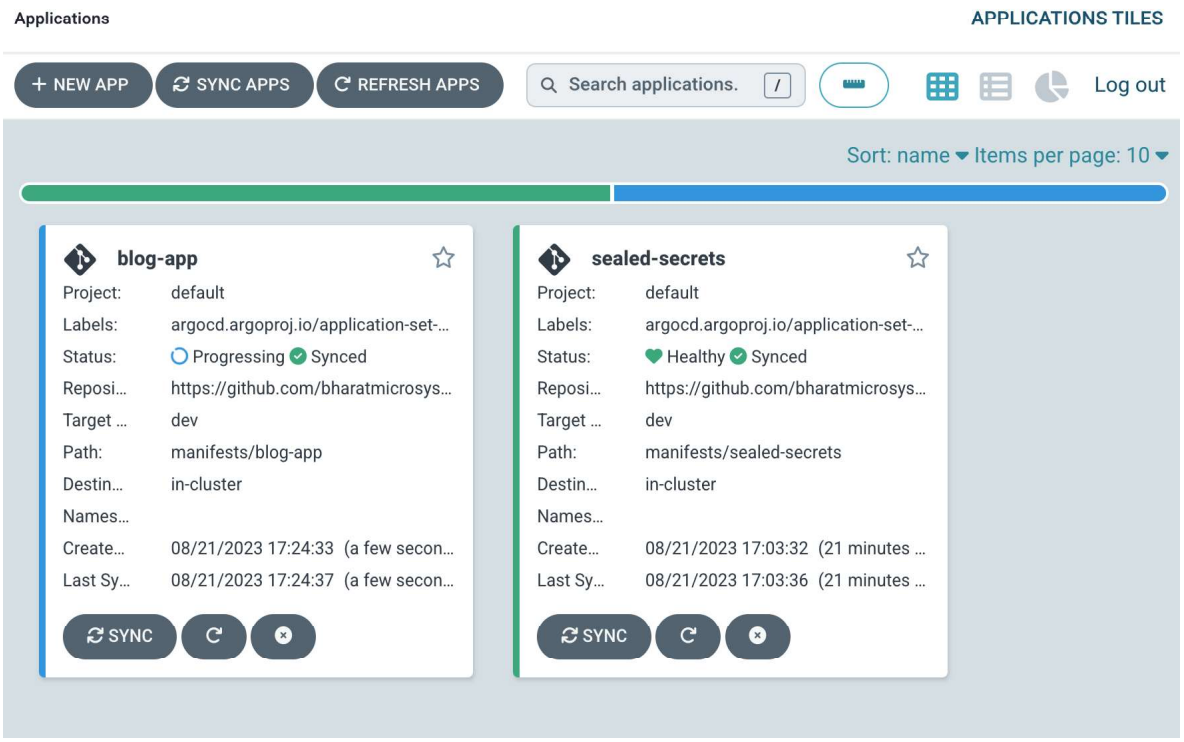

Figure 12.14 – Argo CD Web UI – Applications

Wait for the application to progress. Once it turns green, you should see the following within the application:

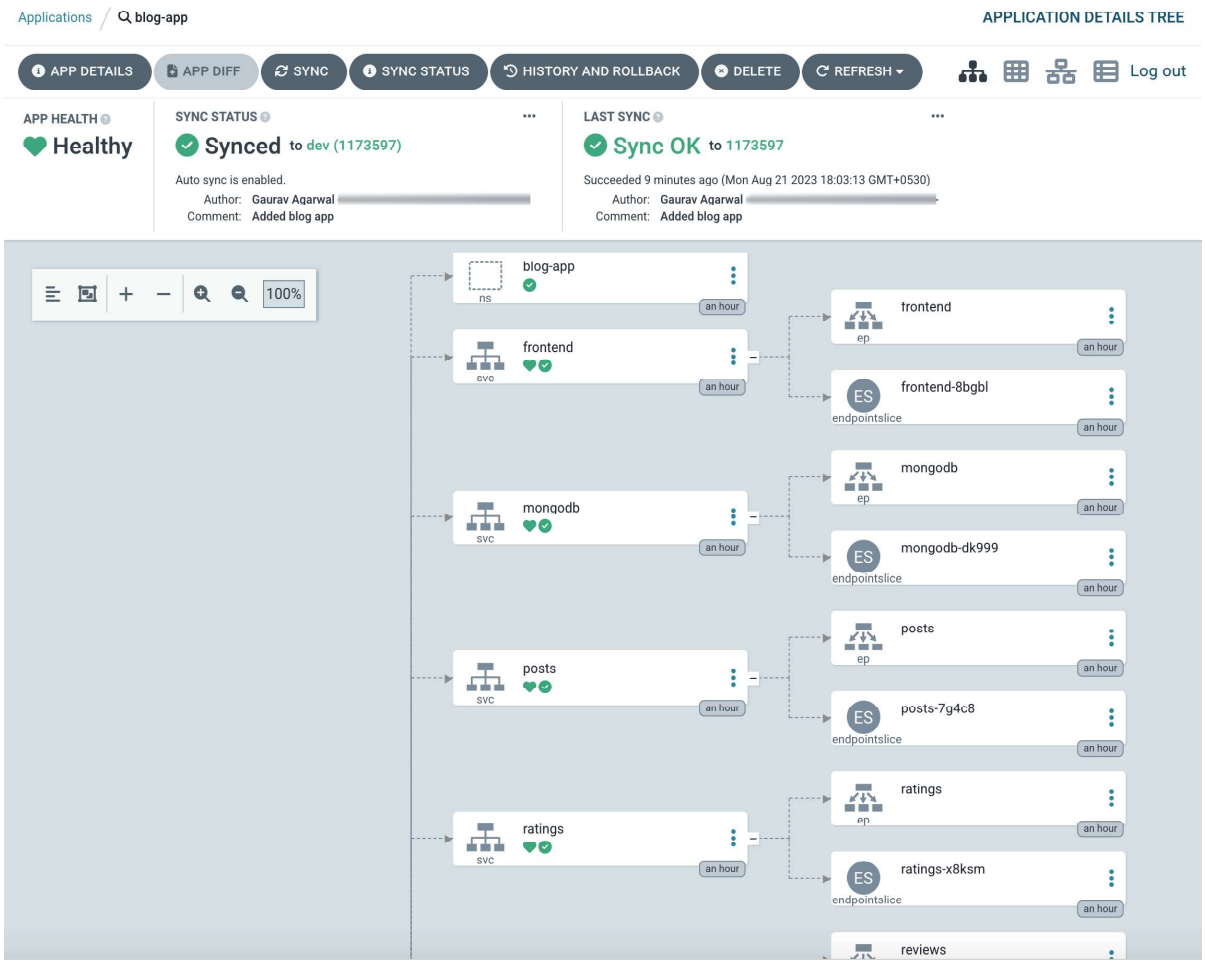

Figure 12.15 – Argo CD Web UI – blog-app

Now that the application is all synced up, we can check the resources that were created within the `blog-app` namespace. Let's list the Services first using the following command:

```
$ kubectl get svc -n blog-app
NAME     TYPE          CLUSTER-IP     EXTERNAL-IP PORT(S)
frontend LoadBalancer 10.71.244.154 34.68.221.0 80:3203/TCP
mongodb  ClusterIP     None           <none>      27017/TCP
posts    ClusterIP     10.71.242.211 <none>      5000/TCP
ratings  ClusterIP     10.71.244.78  <none>      5000/TCP
reviews  ClusterIP     10.71.247.128 <none>      5000/TCP
users    ClusterIP     10.71.241.25  <none>      5000/TCP
```

As we can see, it lists all the Services that we've defined. Note that the *frontend* service is of the `LoadBalancer` type and has an **External IP**. Note down this External IP as we will use it to access our application.

Now, let's list the **pods** to see whether all the microservices are running fine:

```
$ kubectl get pod -n blog-app
NAME                          READY   STATUS    RESTARTS
frontend-7cbdc4c6cd-4jzdw     1/1     Running   0
mongodb-0                     1/1     Running   0
posts-588d8bcd99-sphpm        1/1     Running   0
ratings-7dc45697b-wwfqd       1/1     Running   0
reviews-68b7f9cb8f-2jgvv      1/1     Running   0
users-7cdd4cd94b-g67zw        1/1     Running   0
```

As we can see, all pods are running fine. Note that the `mongodb-0` pod contains a numeric prefix, but the rest of the pods have random UUIDs. You might recall that when we created a **StatefulSet**, the pods always maintained a unique ID and were created in order. At first glance, the application seems to be set up correctly. Let's list the Secrets as well to see whether the `mongodb-creds` secret has been created:

```
$ kubectl get secret -n blog-app
NAME            TYPE     DATA   AGE
mongodb-creds   Opaque   2      80s
```

Here, we can see that the `mongodb-creds` Secret has been created. This shows us that SealedSecret is working fine.

Now, let's go ahead and access our application by opening `http://<frontend-svc-external-ip>`. If you see the following page, the application was deployed correctly:

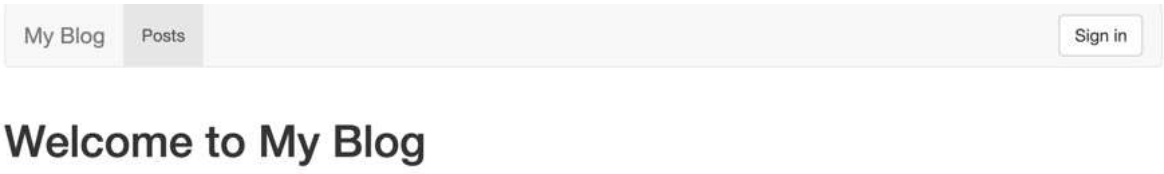

Figure 12.16 – Blog App home page

As an exercise, play around with the application by clicking on **Sign In** > **Not a user? Create an Account** and then fill in the details to register. You can *create* a new **Post**, add **Reviews**, and provide **Ratings**. You can also *update* your reviews, *delete* them, update ratings, and more. Try out the app to see whether all aspects are working correctly. You should be able to see something like the following:

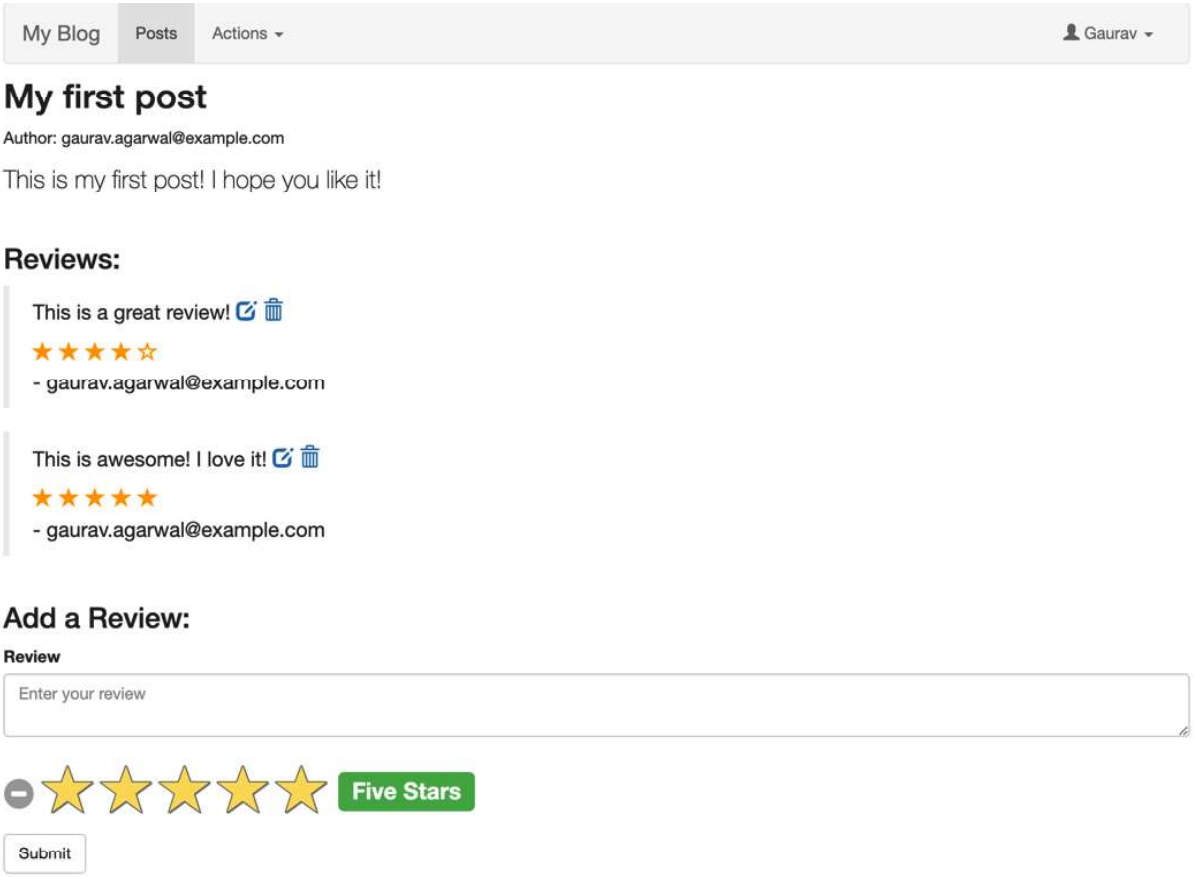

Figure 12.17 – Blog App posts

As we're happy with the application, we can raise a pull request from the `dev` branch to the `prod` branch. Once you merge the pull request, you will see that similar services will emerge in the production environment. You can use pull request-based gating for CD as well. This ensures that your environments remain independent while being sourced from the same repository, albeit from different branches.

Summary

This chapter has covered continuous deployment and delivery, and we understood the need for CD and the basic CD workflow for a container application. We discussed several modern deployment strategies and how CI tools cannot fulfill those responsibilities. Using the GitOps principles, we created an Environment repository and deployed our GKE-based environment using GitHub Actions by employing the push-based model. Then, we looked at using Argo CD as our CD tool and installed it. To avoid committing sensitive information in Git, such as secrets, we discussed Bitnami's Sealed Secrets. We then deployed the sample Blog App using Argo CD, using GitOps all the while.

In the next chapter, we will explore another vital aspect of modern DevOps – securing the deployment pipeline.

Questions

Answer the following questions to test your knowledge of this chapter:

1. Which of the following are CD tools? (Choose three)

   A. Spinnaker

   B. GitHub

   C. Argo CD

   D. AWS Code Deploy

2. CD requires human input for deployment to production. (True/False)
3. Argo CD supports Blue/Green deployments out of the box. (True/False)
4. What would you use to initiate deployment using Argo CD?

   A. Trigger the pipeline manually

   B. Check in changes to your Git repository

   C. Use CI to trigger Argo CD pipelines

   D. Argo CD pipelines don't react to external stimuli

5. An Argo CD ApplicationSet helps generate applications based on templates. (True/False)
6. What branch names should you prefer for your Environment repository?

   A. `dev`, `staging`, and `prod`

   B. `feature`, `develop`, and `master`

   C. `release` and `main`

7. Which of the following deployment models does Argo CD use?

   A. Push model

   B. Pull model

   C. Staggering model

8. You should use Terraform to install Argo CD as you can store all configurations in Git. (True/False)
9. Argo CD can sync resources from which of the following sources? (Choose two)

   A. Git repository

   B. Container Registry

   C. JFrog Artifactory's raw repository

10. What would Argo CD do if you manually changed a resource outside Git?

    A. Argo CD would change the resource so that it matches the Git configuration

    B. Argo CD would notify you that a resource has changed outside Git

    C. Argo CD would do nothing

11. You can check in Sealed Secrets to a Git repository. (True/False)

Answers

Here are the answers to this chapter's questions:

1. A, C, and D
2. True
3. True
4. B
5. True
6. A
7. B
8. True
9. A, B
10. A
11. True

13
Securing and Testing Your CI/CD Pipeline

In the previous chapters, we looked at **Continuous Integration** (**CI**) and **Continuous Deployment/Delivery** (**CD**) with GitOps as the central concept. Both concepts and the tooling surrounding them help us deliver better software faster. However, one of the most critical aspects of technology is security and quality assurance. Though security was not considered in DevOps' initial days, with the advent of **DevSecOps**, modern DevOps now places a great emphasis on it. In this chapter, we'll try to understand the concepts surrounding container applications' security and testing and how to apply them within CI and CD.

In this chapter, we're going to cover the following main topics:

- Securing and testing CI/CD pipelines
- Revisiting the Blog Application
- Container vulnerability scanning
- Managing secrets
- Binary authorization
- Release gating with pull requests and deploying our application in production
- Security and testing best practices for modern DevOps pipelines

Technical requirements

For this chapter, we will spin up a cloud-based Kubernetes cluster, **Google Kubernetes Engine** (**GKE**), for the exercises. Currently, **Google Cloud Platform** (**GCP**) provides a free $300 trial for 90 days, so you can go ahead and sign up for one at `https://console.cloud.google.com/`.

You will also need to clone the following GitHub repository for some of the exercises:

`https://github.com/PacktPublishing/Modern-DevOps-Practices-2e`.

You can use the Cloud Shell offering available on GCP to follow this chapter. Go to Cloud Shell and start a new session. Run the following commands to clone the repository into your home directory to access the required resources:

```
$ git clone https://github.com/PacktPublishing/Modern-DevOps-Practices-2e.git \
modern-devops
```

We also need to set the project ID and enable a few GCP APIs that we will use in this chapter. To do so, run the following commands:

```
$ PROJECT_ID=<YOUR_PROJECT_ID>
$ gcloud services enable iam.googleapis.com \
 container.googleapis.com \
 binaryauthorization.googleapis.com \
 containeranalysis.googleapis.com \
 secretmanager.googleapis.com \
 cloudresourcemanager.googleapis.com \
 cloudkms.googleapis.com
```

Now, in the next section, let's look at how to secure and test CI/CD pipelines.

Securing and testing CI/CD pipelines

With continuous cyber threats and the ongoing war between cybersecurity experts and cybercriminals, security has always been the top priority for most organizations, and it also forms a significant part of a mature organization's investment.

However, security comes with its costs. Most organizations have cybersecurity teams that audit their code regularly and give feedback. However, that process is generally slow and happens when most of the code is already developed and difficult to modify.

Similarly, while most organizations significantly emphasize automated testing, many still heavily depend on manual testing. Manual testing is not only labor-intensive but also lacks repeatability. DevOps places great importance on automating tests to ensure that they can be repeated with every release, enabling the detection of existing issues and thorough testing of new features. Additionally, automation is essential for efficiently conducting regression testing on bug fixes, as manual testing in such cases is inefficient.

Therefore, embedding security and testing at the early stages of development is an essential goal for modern DevOps. Embedding security with DevOps has led to the concept of DevSecOps, where developers, cybersecurity experts, and operations teams work together to create better and more secure software faster.

Securing and testing your software using CI/CD pipelines offers various significant business advantages. Firstly, it ensures security by protecting sensitive data, preventing vulnerabilities, and ensuring compliance. Secondly, it improves quality and reliability through early issue detection, consistency, and higher product quality. This leads to cost reduction by reducing rework, speeding up time to market, and optimizing resource usage. Additionally, it mitigates risks by increasing resilience and enabling stress testing. Moreover, it ensures business continuity through disaster recovery and efficient rollback procedures. Furthermore, it provides a competitive advantage by fostering faster innovation and market responsiveness. Finally, it enhances reputation and customer trust by building confidence in your products and services and safeguarding your brand's reputation. In essence, securing and testing CI/CD pipelines is both a technical necessity and a strategic business imperative that enhances security, quality, and reliability while reducing costs and risks, ultimately leading to improved customer satisfaction, business continuity, and a competitive edge in the market.

There are many ways of embedding security within the software supply chain. Some of these might include static code analysis, security testing, and applying organization-specific security policies within the process, but the idea of security is not to slow down development. Instead of human input, we can always use tools that can significantly improve the security posture of the software we develop. Similarly, testing need not be manual and slow and, instead, should use automation to plug in seamlessly with the CI/CD process.

CI/CD pipelines are one of the essential features of modern DevOps, and they orchestrate all processes and combine all tools to deliver better software faster, but how would you secure them? You may want to ask the following questions:

- How do I scan a container image for vulnerabilities?
- How do I store and manage sensitive information and secrets securely?
- How do I ensure that my application is tested before deployment to production?
- How do I ensure that only tested and approved container images are deployed in production?

Throughout this chapter, we will try to answer these using best practices and tooling. For reference, look at the following workflow diagram:

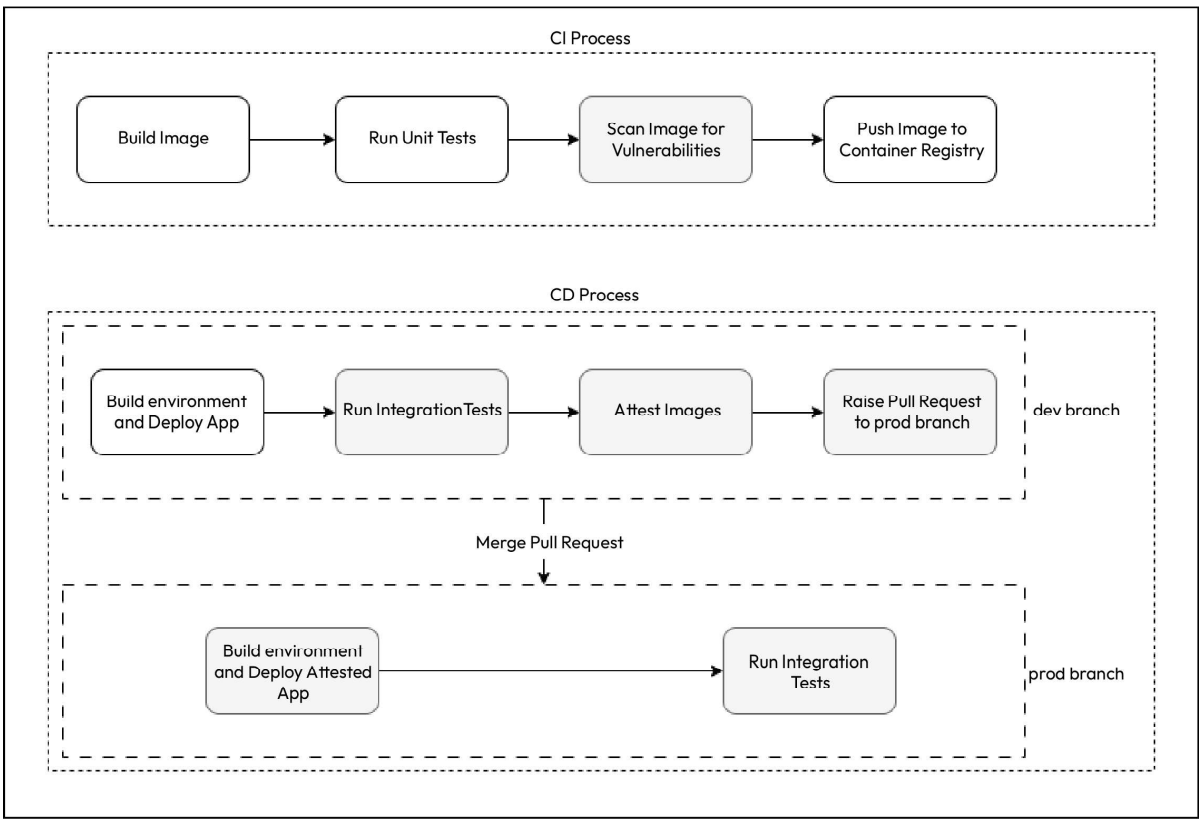

Figure 13.1 – Secure CI/CD workflow

As depicted in the previous figure, we need to modify the CI pipeline to include an additional step for vulnerability scanning. We also require two CD pipelines, one for the Dev environment and another for Prod. To enhance reusability, we'll restructure our GitHub Actions workflow. We'll divide the workflows into parent and child workflows. Let's begin by examining the CD workflow for the Dev environment to get an overview:

```
name: Dev Continuous Delivery Workflow
on:
  push:
    branches: [ dev ]
jobs:
  create-environment-and-deploy-app:
    name: Create Environment and Deploy App
    uses: ./.github/workflows/create-cluster.yml
    secrets: inherit
  run-tests:
    name: Run Integration Tests
    needs: [create-environment-and-deploy-app]
    uses: ./.github/workflows/run-tests.yml
```

```
    secrets: inherit
  binary-auth:
    name: Attest Images
    needs: [run-tests]
    uses: ./.github/workflows/attest-images.yml
    secrets: inherit
  raise-pull-request:
    name: Raise Pull Request
    needs: [binary-auth]
    uses: ./.github/workflows/raise-pr.yml
    secrets: inherit
```

The workflow begins with a `name`, followed by a declaration of `on push branches dev`. This configuration ensures that the workflow triggers with every push to the `dev` branch. We define multiple jobs in sequence, each depending on the previous one using the `needs` attribute. Each job invokes a child workflow specified by the `uses` attribute, and it provides GitHub secrets to these child workflows by setting `inherit` for the `secrets` attribute.

The workflow accomplishes the following tasks:

1. Sets up the Dev Kubernetes cluster, configures Argo CD and supporting tools to establish the environment, and deploys the sample Blog App.
2. Executes integration tests on the deployed Blog App.
3. If the tests pass, it utilizes binary authorization (more details to follow) to attest images, ensuring that only tested artifacts are allowed for deployment to production.
4. Initiates a pull request for deployment to the Prod environment.

In a similar manner, we have the following Prod CD Workflow file:

```
name: Prod Continuous Delivery Workflow
on:
  push:
    branches: [ prod ]
jobs:
  create-environment-and-deploy-app:
    name: Create Environment and Deploy App
    uses: ./.github/workflows/create-cluster.yml
    secrets: inherit
  run-tests:
    name: Run Integration Tests
    needs: [create-environment-and-deploy-app]
    uses: ./.github/workflows/run-tests.yml
    secrets: inherit
```

This workflow is similar to the Dev workflow but does not include the `binary-auth` and `raise-pull-request` steps, as they are unnecessary at this stage. To understand it better, let's begin by examining the Dev workflow. The initial step of the Dev workflow involves creating the environment and deploying the application. However, before we proceed, let's revisit the Blog App in the next section.

Revisiting the Blog Application

As we already discussed the Blog App in the last chapter, let's look at the services and their interactions again in the following diagram:

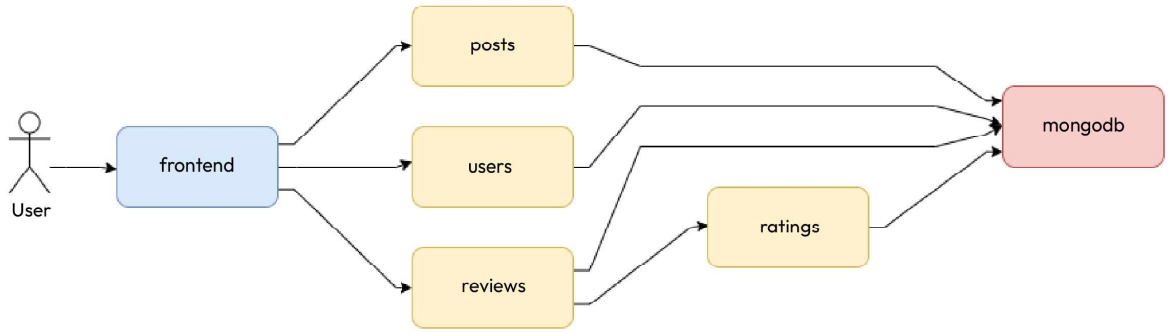

Figure 13.2 – The Blog App services and interactions

We've already created CI and CD pipelines for building, testing, and pushing our Blog Application microservices containers using GitHub Actions and deploying them using Argo CD in a GKE cluster.

If you remember, we created the following resources for the application to run seamlessly:

- **MongoDB** – We deployed an auth-enabled MongoDB database with root credentials. The credentials were injected via environment variables sourced from a Kubernetes **Secret** resource. To persist our database data, we created a **PersistentVolume** mounted to the container, which we provisioned dynamically using a **PersistentVolumeClaim**. As the container is stateful, we used a **StatefulSet** to manage it and, therefore, a headless Service to expose the database.
- **Posts, reviews, ratings, and users** – The posts, reviews, ratings, and users microservices interacted with MongoDB through the root credentials injected via environment variables sourced from the same **Secret** resource as MongoDB. We deployed them using their respective **Deployment** resources and exposed all of them via individual **ClusterIP** Services.
- **Frontend** – The frontend microservice does not need to interact with MongoDB, so there was no interaction with the **Secret** resource. We deployed this service as well using a **Deployment** resource. As we wanted to expose the service on the internet, we created a **LoadBalancer** Service for it.

We can summarize them in the following diagram:

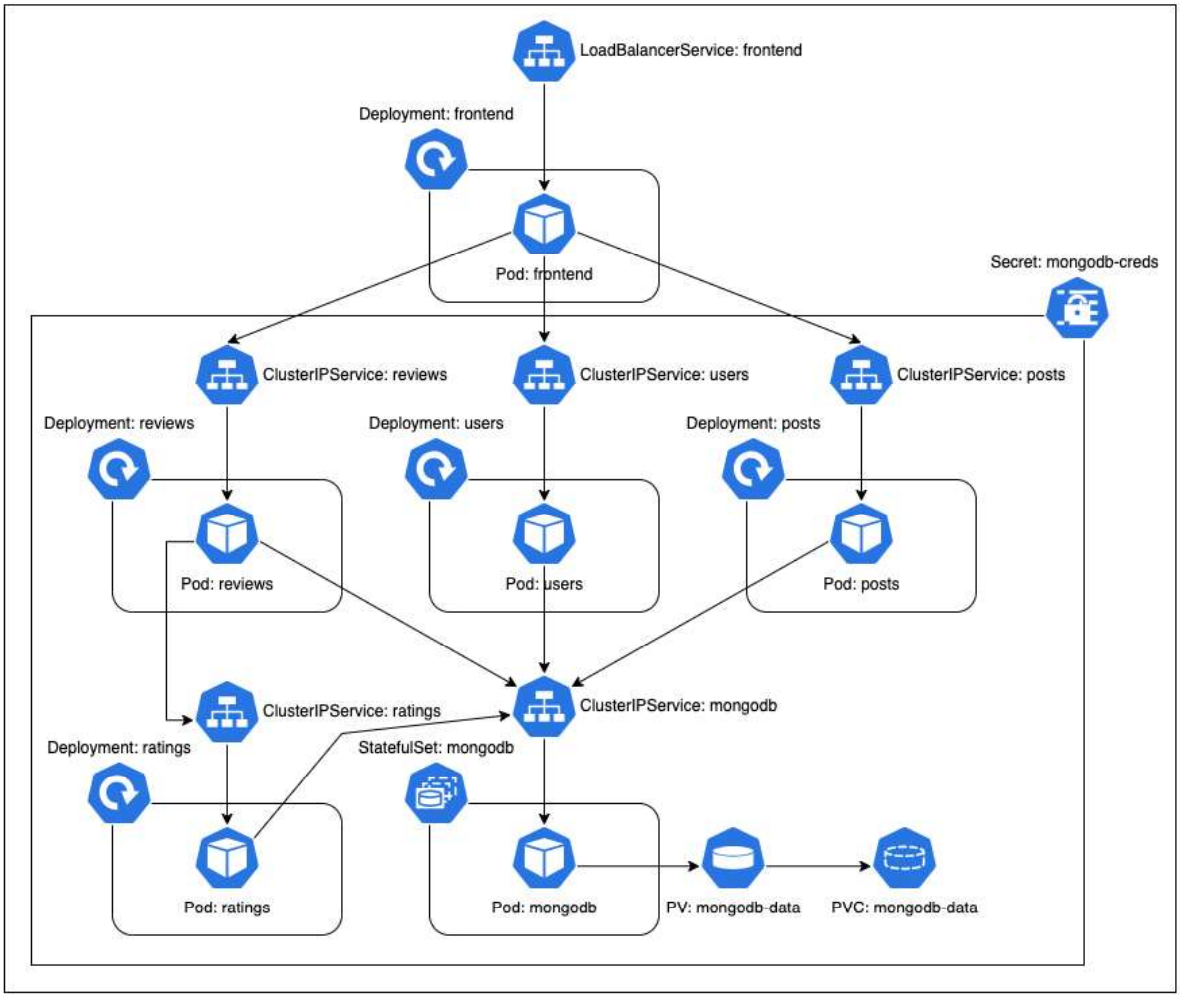

Figure 13.3 – The Blog App – Kubernetes resources and interactions

In subsequent sections, we will cover all aspects of implementing this workflow, starting with vulnerability scanning.

Container vulnerability scanning

Perfect software is costly to write and maintain, and every time someone makes changes to running software, the chances of breaking something are high. Apart from other bugs, changes also add a lot of software vulnerabilities. You cannot avoid these as software developers. Cybersecurity experts and cybercriminals are at constant war with each other, evolving with time. Every day, a new set of vulnerabilities are found and reported.

In containers, vulnerabilities can exist on multiple fronts and may be completely unrelated to what you're responsible for. Well, developers write code, and excellent ones do it securely. Still, you never know whether a base image may contain vulnerabilities your developers might completely overlook. In modern DevOps, vulnerabilities are expected, and the idea is to mitigate them as much as possible. We should reduce vulnerabilities, but doing so manually is time-consuming, leading to toil.

Several tools are available on the market that provide container vulnerability scanning. Some of them are open source tools such as **Anchore**, **Clair**, **Dagda**, **OpenSCAP**, Sysdig's **Falco**, or **Software-as-a-Service** (**SaaS**) services available with **Google Container Registry** (**GCR**), **Amazon Elastic Container Registry** (**ECR**), and **Azure Defender**. For this chapter, we'll discuss **Anchore Grype**.

Anchore Grype (`https://github.com/anchore/grype`) is a container vulnerability scanner that scans your images for known vulnerabilities and reports their severity. Based on that, you can take appropriate actions to prevent vulnerabilities by including a different base image or modifying the layers to remove vulnerable components.

Anchore Grype is a simple **Command-Line Interface** (**CLI**)-based tool that you can install as a binary and run anywhere—within your local system or your CI/CD pipelines. You can also configure it to fail your pipeline if the vulnerability level increases above a particular threshold, thereby embedding security within your automation—all this happening without troubling your development or security team.

Now, let's go ahead and see Anchore Grype in action.

Installing Anchore Grype

As we want to implement vulnerability scanning within our CI pipelines, let's modify the `mdo-posts` repository we created in *Chapter 11*.

Let's clone the repository first using the following command and `cd` into the `workflows` directory:

```
$ git clone git@github.com:<your_github_user>/mdo-posts.git
$ cd mdo-posts/.github/workflows/
```

Anchore Grype offers an installation script within its GitHub repository that you can download and run, and it should set it up for you. We'll modify the `build.yaml` file to include the following step before the `Login to Docker Hub` step so that we can install Grype within our CI workflow:

```
- name: Install Grype
  id: install-grype
  run: curl -sSfL https://raw.githubusercontent.com/anchore/grype/main/install.sh | sh -s
-- -b /usr/local/bin
```

Next, we need to use Grype to scan our images for vulnerabilities.

Scanning images

To run container vulnerability scanning, we can use the following command:

```
$ grype <container-image>
```

This will report a list of vulnerabilities with severities—`Negligible`, `Low`, `Medium`, `High`, `Critical`, or `Unknown`—within the image. We can also set a threshold within Grype to fail when any vulnerabilities are equal to or worse than it. For example, if we don't want to allow any `Critical` vulnerabilities in the container, we can use the following command:

```
$ grype -f critical <container-image>
```

To do so, we will add the following step within the `build.yaml` file after the `Build the Docker image` step:

```
- name: Scan Image for Vulnerabilities
  id: vul-scan
  run: grype -f critical ${{ secrets.DOCKER_USER  }}/mdo-posts:$(git rev-parse --short
"$GITHUB_SHA")
```

As we've made all the changes, let's push the modified CI pipeline using the following commands:

```
$ cp ~/modern-devops/ch13/grype/build.yaml .
$ git add --all
$ git commit -m "Added grype"
$ git push
```

As soon as we push the image, we will see the following in the GitHub Actions tab:

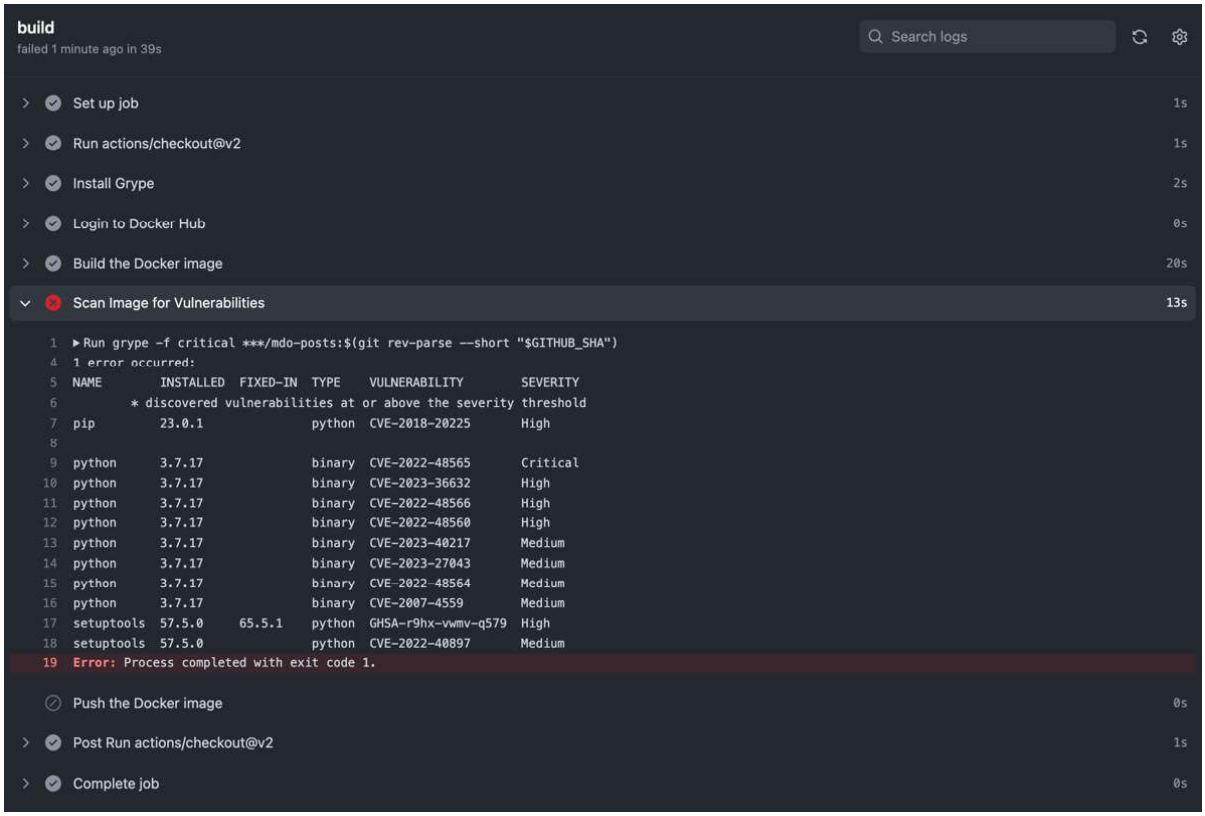

Figure 13.4 – Vulnerability scan failure

As we can see, Grype has reported several vulnerabilities with one being `Critical`. It has also failed the CI pipeline. That is automated vulnerability scanning in action. This will discover vulnerabilities and only allow builds to end up in your container registry if they meet minimum security standards.

We need to fix the issue here, so let's look at a more recent image and see whether it can fix the problem. Therefore, instead of using `python:3.7-alpine`, we will use `python:alpine3.18`. Let's do that and push our code to GitHub using the following commands:

```
$ cd ~/mdo-posts && cp ~/modern-devops/ch13/grype/Dockerfile .
$ git add --all
$ git commit -m "Updated base image"
$ git push
```

Let's revisit GitHub Actions and see what we get in the `build` output:

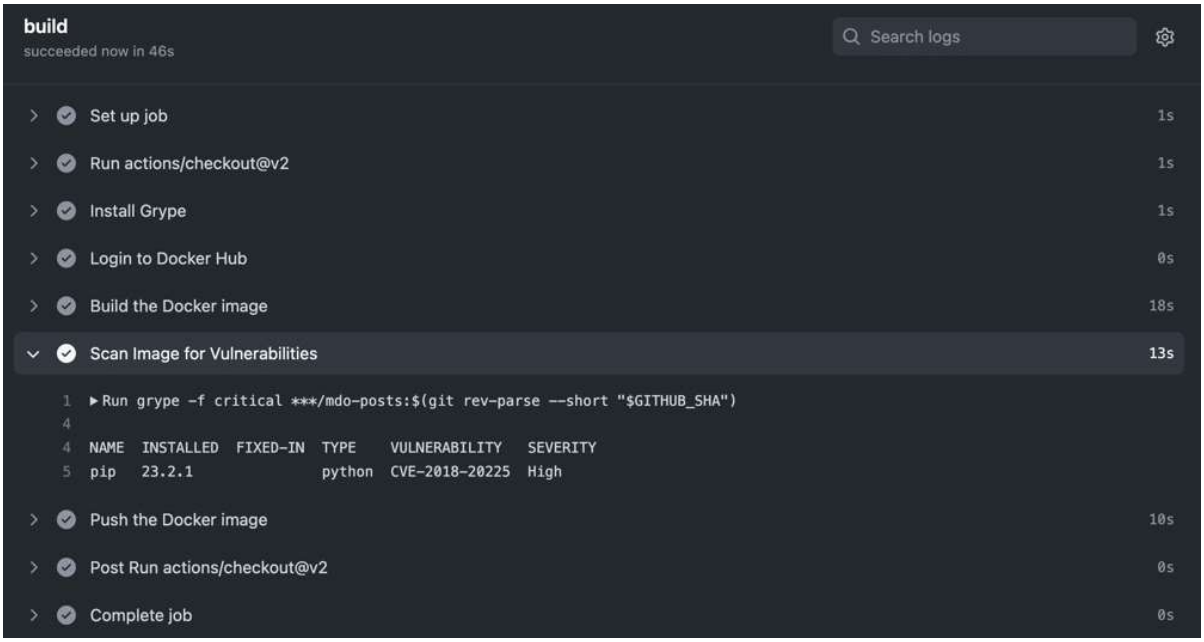

Figure 13.5 – Vulnerability scan success

The vulnerability scan did not stop our CI build this time, as no `Critical` vulnerabilities were found.

> **Tip**
>
> Continually update the base image with time, as newer ones contain fewer vulnerabilities and fix older ones.

Now that we've secured the image for vulnerabilities, our CI pipeline is complete. You can replicate this process for other microservices as needed. Let's proceed to discuss CD pipelines.

If you remember, in the last chapter, following the GitOps model, we stored the manifests of all resources on Git. However, due to security concerns with Kubernetes Secrets, we used **SealedSecrets** to manage them securely.

However, this may not be the ideal solution for all teams due to the following inherent issues:

- SealedSecrets are reliant on the controller that encrypts them. If we lose this controller, we also lose the ability to recreate the secret, essentially losing the Secret forever.
- Access to the Secret is limited to logging in to the cluster and using `kubectl`, which doesn't provide non-admins with the ability to manage secrets. While this approach might suit some teams, it may not suit others.

Therefore, we will explore managing secrets using a Secrets management tool to establish a standardized method for centrally managing secrets with more granular control over access. Let's delve into this topic in the next section.

Managing secrets

Software always requires access to sensitive information such as user data, credentials, **Open Authorization** (**OAuth**) tokens, passwords, and other information known as secrets. Developing and managing software while keeping all these aspects secure has always been a concern. The CI/CD pipelines might deal with them as they build and deliver working software by combining code and other dependencies from various sources that may include sensitive information. Keeping these bits secure is of utmost importance; therefore, the need arises to use modern DevOps tools and techniques to embed security within the CI/CD pipelines themselves.

Most application code requires access to sensitive information. These are called **secrets** in the DevOps world. A secret is any data that helps someone prove their identity, authenticate, and authorize privileged accounts, applications, and services. Some of the potential candidates that constitute secrets are listed here:

- Passwords
- API tokens, GitHub tokens, and any other application key
- **Secure Shell** (**SSH**) keys
- **Transport Layer Security** (**TLS**), **Secure Sockets Layer** (**SSL**), and **Pretty Good Privacy** (**PGP**) private keys
- One-time passwords

A good example could be a container requiring access to an API key to authenticate with a third-party API or a username and password to authenticate with a backend database. Developers need to understand where and how to store secrets so that they are not exposed inadvertently to people who are not supposed to view them.

When we run a CI/CD pipeline, it becomes imperative to understand how we place those secrets as, in CI/CD pipelines, we build everything from the source. "*Do not store secrets with code*" is a prominent piece of advice we've all heard.

> **Tip**
>
> Never store hardcoded secrets within CI/CD pipelines or store secrets in a source code repository such as Git.

How can we access secrets without including them in our code to run a fully automated GitOps-based CI/CD pipeline? Well, that's something we need to figure out.

> **Tip**
> When using containers, the thing to avoid is baking the secrets within an image. While this is a prominent piece of advice, many developers do this inadvertently, leading to many security holes. It is very insecure, and you should avoid doing it at all costs.

You can overcome this problem by using some form of **secrets management solution**. A secrets management solution or a **key management solution** helps store and manage your secrets and secure them with encryption at rest and in transit. There are secrets management tools within cloud providers, such as **Secret Manager** in GCP and **Amazon Web Services** (**AWS**), or you can use a third-party tool, such as **HashiCorp Vault**, if you want to go cloud agnostic. All these solutions provide APIs to create and query secrets at runtime, and they secure the API via HTTPS to allow encryption in transit. That way, you don't need to store your secrets with code or bake it within an image.

In this discussion, we'll use the **Secret Manager** solution offered by GCP to store secrets, and we will access them while running the CI/CD pipeline. Secret Manager is Google Cloud's secrets management system, which helps you store and manage secrets centrally. It is incredibly secure and uses **Hardware Security Modules** (**HSMs**) to harden your secrets further.

In this chapter, we will look at improving the CI/CD pipeline of our Blog Application, which we discussed in the last chapter, and will use the same sample application. Therefore, let's go ahead and create the `mongodb-creds` Secret in Google Cloud Secret Manager.

Creating a Secret in Google Cloud Secret Manager

Let's create a secret called `external-secrets`, where we will pass the MongoDB credentials in JSON format. To do so, run the following command:

```
$ echo -ne \
'{"MONGO_INITDB_ROOT_USERNAME": "root", "MONGO_INITDB_ROOT_PASSWORD": "itsasecret"}'  \
| gcloud secrets create external-secrets --locations=us-central1 \
--replication-policy=user-managed --data-file=-
Created version [1] of the secret [external-secrets].
```

In the preceding command, we echo a JSON containing `MONGO_INITDB_ROOT_USERNAME` and `PASSWORD` directly into the `gcloud secrets create` command. We have specified a particular location to avoid replicating it in other regions as a cost-saving measure. However, it's highly recommended to replicate secrets to prevent potential loss in case of a zonal outage. The JSON is stored as a new version of our secret. Secret Manager utilizes versioning for secrets, so any new value assigned to the secret (`external-secrets`) is versioned and stored within Secret Manager. You can reference a specific version either by its version number or by using the `latest` keyword to access the most recent version.

As seen in the output, we've created the first version of our secret (`version 1`). Typically, this is done during development and should remain outside the CI/CD process. Instead of storing the Secret resource manifest in your source code repository, you can keep it in Secret Manager.

Now that we've created the secret, we must access it within our application. To achieve this, we require a tool to access the secret stored in Secret Manager from the Kubernetes cluster. For this purpose, we will use **External Secrets Operator**.

Accessing external secrets using External Secrets Operator

External Secrets Operator (`https://external-secrets.io/latest/`) is a Kubernetes operator used in Kubernetes clusters to manage external secrets securely. It is designed to automate the retrieval and management of secrets stored in external secret stores such as AWS Secret Manager, GCP Secret Manager, Hashicorp Vault, and so on, and inject them into Kubernetes pods as Kubernetes Secrets. Operators are a way to extend Kubernetes functionality and automate tasks.

How it works

External Secrets Operator serves as a bridge between the Kubernetes cluster and external secret management systems. We define an `ExternalSecret` custom resource within the Kubernetes cluster, which the operator monitors. When an `ExternalSecret` resource is created or updated, the operator interacts with the external secret store specified in the `ClusterSecretStore` CRD to retrieve the secret data. It then creates or updates the corresponding Kubernetes Secrets. This process is illustrated in the following diagram:

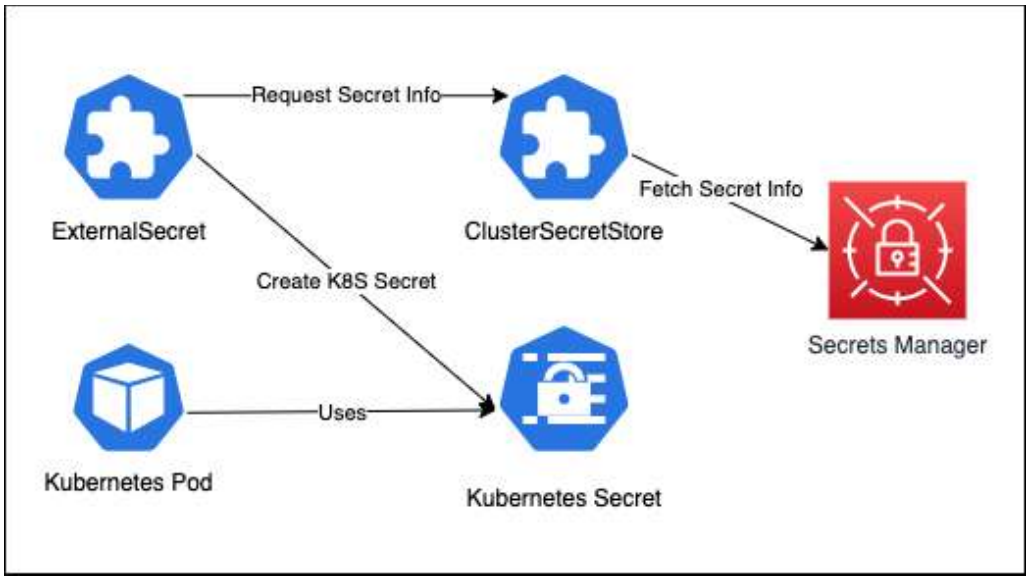

Figure 13.6 – External Secret Operator

Now, this process has a lot of benefits, some of which are as follows:

- **Enhanced Security**: Secrets remain in a dedicated, secure secret store
- **Automation**: Automates the retrieval and rotation of secrets
- **Simplified Deployment**: Eases the management of secrets within Kubernetes applications
- **Compatibility**: Works with various external secret stores, making it versatile

Now, let's go ahead and install External Secrets Operator on our Kubernetes cluster.

Installing External Secrets Operator

External Secrets Operator is available as a **Helm chart**, and Argo CD supports it. A Helm chart is a collection of preconfigured Kubernetes resources (such as Deployments, Services, ConfigMaps, and more) organized into a package that makes it easy to deploy and manage applications in Kubernetes. Helm is a package manager for Kubernetes that allows you to define, install, and upgrade even the most complex Kubernetes applications in a repeatable and standardized way. Therefore, we must create an Argo CD application pointing to the Helm chart to install it. To do so, we will create the following `manifests/argocd/external-secrets.yaml` manifest file:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: external-secrets
  namespace: argocd
spec:
  project: default
  source:
    chart: external-secrets/external-secrets
    repoURL: https://charts.external-secrets.io
    targetRevision: 0.9.4
    helm:
      releaseName: external-secrets
  destination:
    server: "https://kubernetes.default.svc"
    namespace: external-secrets
```

The application manifest creates an `external-secrets` application on the `argocd` namespace within the `default` project. It downloads the `0.9.4` revision from the `external-secrets` Helm chart repository and deploys the chart on the Kubernetes cluster on the `external-secrets` namespace.

To install this application, we need to apply this manifest using Terraform. Therefore, to do so, we make the following entry in the `app.tf` file:

```
data "kubectl_file_documents" "external-secrets" {
    content = file("../manifests/argocd/external-secrets.yaml")
}
```

```
resource "kubectl_manifest" "external-secrets" {
  depends_on = [
    kubectl_manifest.argocd,
  ]
  for_each  = data.kubectl_file_documents.external-secrets.manifests
  yaml_body = each.value
  override_namespace = "argocd"
}
```

To deploy this, we must check these files into source control. Let's clone the `mdo-environments` repository that we created in the last chapters.

If you haven't followed the last chapters, you can do the following to set a baseline. Feel free to skip the next section if you've already set up your environment in *Chapter 12, Continuous Deployment/Delivery with Argo CD*.

Setting up the baseline

To ensure continuity with the last chapters, let's start by creating a service account for Terraform to interact with our GCP project using the following commands:

```
$ gcloud iam service-accounts create terraform \
 --description="Service Account for terraform" \
 --display-name="Terraform"
$ gcloud projects add-iam-policy-binding $PROJECT_ID \
--member="serviceAccount:terraform@$PROJECT_ID.iam.gserviceaccount.com" \
--role="roles/editor"
$ gcloud iam service-accounts keys create key-file \
--iam-account=terraform@$PROJECT_ID.iam.gserviceaccount.com
```

You will see a file called `key-file` created within your working directory. Now, create a new repository called `mdo-environments` with a `README.md` file on GitHub, rename the `main` branch to `prod`, and create a new branch called `dev` using GitHub. Navigate to `https://github.com/<your_github_user>/mdo-environments/settings/secrets/actions/new` and create a secret named `GCP_CREDENTIALS`. For the value, print the `key-file` file, copy its contents, and paste it into the **values** field of the GitHub secret.

Next, create another secret, `PROJECT_ID`, and specify your GCP project ID within the **values** field.

Next, we need to create a GCS bucket for Terraform to use as a remote backend. To do this, run the following command:

```
$ gsutil mb gs://tf-state-mdo-terraform-${PROJECT_ID}
```

So, now that all the prerequisites are met, we can clone our repository and copy the baseline code. Run the following commands to do this:

```
$ cd ~ && git clone git@github.com:<your_github_user>/mdo-environments.git
$ cd mdo-environments/
$ git checkout dev
$ cp -r ~/modern-devops/ch13/baseline/* .
$ cp -r ~/modern-devops/ch13/baseline/.github .
```

As we're now on the baseline, let's proceed further to install external secrets with Terraform.

Installing external secrets with Terraform

Let's configure our local repository to install the external secrets manifest. To do so, copy the application manifest and `app.tf` file using the following commands:

```
$ cp ~/modern-devops/ch13/install-external-secrets/app.tf terraform/app.tf
$ cp ~/modern-devops/ch13/install-external-secrets/external-secrets.yaml \
  manifests/argocd/
```

Now that we're all set up and ready, let's go ahead and commit and push our code using the following commands:

```
$ git add --all
$ git commit -m "Install external secrets operator"
$ git push
```

As soon as we push the code, we'll see that the GitHub Actions workflow has been triggered. To access the workflow, go to `https://github.com/<your_github_user>/mdo-environments/actions`. Soon, the workflow will apply the configuration, create the Kubernetes cluster, and deploy Argo CD, the Sealed Secrets controller, and External Secrets Operator.

Once the workflow is successful, we can do the following to access the Argo Web UI.

We must first authenticate with the GKE cluster. To do so, run the following command:

```
$ gcloud container clusters get-credentials \
 mdo-cluster-dev --zone us-central1-a --project $PROJECT_ID
```

To utilize the Argo CD Web UI, you will require the external IP address of the `argo-server` service. To get that, run the following command:

```
$ kubectl get svc argocd-server -n argocd
NAME          TYPE        EXTERNAL-IP  PORTS          AGE
argocd-server LoadBalaner 34.122.51.25 80/TCP,443/TCP 6m15s
```

So, now we know that Argo CD is accessible on `https://34.122.51.25/`.

Next, we will run the following commands to reset the admin password:

```
$ kubectl patch secret argocd-secret -n argocd \
-p '{"data": {"admin.password": null, "admin.passwordMtime": null}}'
$ kubectl scale deployment argocd-server --replicas 0 -n argocd
$ kubectl scale deployment argocd-server --replicas 1 -n argocd
```

Now, allow two minutes for the new credentials to be generated. After that, execute the following command to retrieve the password:

```
$ kubectl -n argocd get secret argocd-initial-admin-secret \
 -o jsonpath="{.data.password}" | base64 -d && echo
```

As we now have the credentials, log in, and you will see the following page:

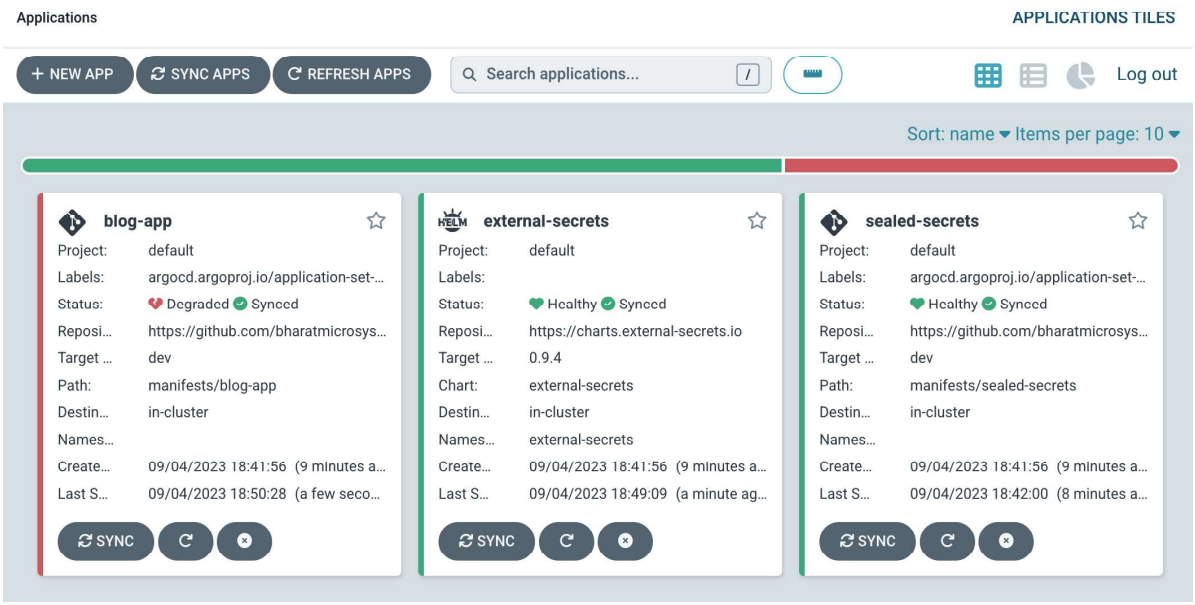

Figure 13.7 – Argo CD Web UI – home page

As we can see, there are three applications – **sealed-secrets**, **external-secrets**, and **blog-app**. While the **sealed-secrets** and **external-secrets** apps are all synced up and green, **blog-app** has degraded. That is because, in my case, I've started fresh and created a new cluster. Therefore, there is no way the Sealed Secrets operator can decrypt the `SealedSecret` manifest that we created in the last chapter, as it was generated by a different Sealed Secrets controller.

We don't need the Sealed Secrets operator; we will use Google Cloud Secret Manager instead. So, let's remove it from our cluster using the following commands:

```
$ rm -rf manifests/sealed-secrets
$ git add --all
$ git commit -m "Removed sealed secrets"
$ git push
```

We've removed the Sealed Secrets operator, and the Argo CD Web UI should reflect that shortly. However, the Blog Application will remain degraded as the `mongodb-creds` Secret is still missing. In the next section, we will use External Secrets Operator to generate the `mongodb-creds` Secret.

Generating the MongoDB Kubernetes Secret using External Secrets Operator

To generate the `mongodb-creds` secret, we would need to create the following resources:

- A `Secret` resource – This is a standard Kubernetes Secret resource containing the service account credentials for Kubernetes to connect with GCP Secret Manager.
- A `ClusterSecretStore` resource – This resource contains configuration for connecting with the secret store (GCP Secret Manager in this case) and uses the `Secret` resource for the service account credentials.
- An `ExternalSecret` resource – This resource contains configuration to generate the required Kubernetes Secret (`mongodb-creds`) out of the extracted Secret from the secret store.

So, let's go ahead and define the `Secret` resource first:

To create the `Secret` resource, we first need to create a GCP service account to interact with Secret Manager using the following commands:

```
$ cd ~
$ gcloud iam service-accounts create external-secrets
```

As we're following the principle of least privilege, we will add the following role-binding to provide access only to the `external-secrets` secret, as follows:

```
$ gcloud secrets add-iam-policy-binding external-secrets \
 --member "serviceAccount:external-secrets@$PROJECT_ID.iam.gserviceaccount.com" \
 --role "roles/secretmanager.secretAccessor"
```

Now, let's generate the service account key file using the following command:

```
$ gcloud iam service-accounts keys create key.json \
--iam-account=external-secrets@$PROJECT_ID.iam.gserviceaccount.com
```

Now, copy the contents of the `key.json` file into a new GitHub Actions secret called `GCP_SM_CREDENTIALS`. We will use GitHub Actions to set this value during runtime dynamically; therefore, the following secret manifest will contain a placeholder:

```
apiVersion: v1
data:
  secret-access-credentials: SECRET_ACCESS_CREDS_PH
kind: Secret
metadata:
  name: gcpsm-secret
type: Opaque
```

Let's look at the `ClusterSecretStore` resource next:

```
apiVersion: external-secrets.io/v1alpha1
kind: ClusterSecretStore
metadata:
  name: gcp-backend
spec:
  provider:
      gcpsm:
        auth:
          secretRef:
            secretAccessKeySecretRef:
              name: gcpsm-secret
              key: secret-access-credentials
        projectID: PROJECT_ID_PH
```

The manifest defines the following:

- A `ClusterSecretStore` resource called `gcp-backend`
- A provider configuration of the `gcpsm` type using auth information in the `gcpsm-secret` secret we defined before

Now, let's look at the `ExternalSecret` resource manifest:

```
apiVersion: external-secrets.io/v1alpha1
kind: ExternalSecret
metadata:
  name: mongodb-creds
  namespace: blog-app
spec:
  secretStoreRef:
    kind: SecretStore
    name: gcp-backend
  target:
    name: mongodb-creds
  data:
  - secretKey: MONGO_INITDB_ROOT_USERNAME
    remoteRef:
```

```
      key: external-secrets
      property:  MONGO_INITDB_ROOT_USERNAME
  - secretKey: MONGO_INITDB_ROOT_PASSWORD
    remoteRef:
      key: external-secrets
      property:  MONGO_INITDB_ROOT_PASSWORD
```

The manifest defines an `ExternalSecret` resource with the following specs:

- It is named `mongodb-creds` in the `blog-app` namespace.
- It refers to the `gcp-backend` `ClusterSecretStore` that we defined.
- It maps `MONGO_INITDB_ROOT_USERNAME` from the `external-secrets` Secret Manager secret to the `MONGO_INITDB_ROOT_USERNAME` key of the `mongodb-creds` Kubernetes secret. It does the same for `MONGO_INITDB_ROOT_PASSWORD`.

Now, let's deploy these resources by using the following commands:

```
$ cd ~/mdo-environments
$ cp ~/modern-devops/ch13/configure-external-secrets/app.tf terraform/app.tf
$ cp ~/modern-devops/ch13/configure-external-secrets/gcpsm-secret.yaml \
manifests/argocd/
$ cp ~/modern-devops/ch13/configure-external-secrets/mongodb-creds-external.yaml \
manifests/blog-app/
$ cp -r ~/modern-devops/ch13/configure-external-secrets/.github .
$ git add --all
$ git commit -m "Configure External Secrets"
$ git push
```

This should trigger a GitHub Actions workflow again, and soon, we should see `ClusterSecretStore` and `ExternalSecret` created. To check that, run the following commands:

```
$ kubectl get secret gcpsm-secret
NAME           TYPE     DATA   AGE
gcpsm-secret   Opaque   1      1m
$ kubectl get clustersecretstore gcp-backend
NAME          AGE   STATUS   CAPABILITIES   READY
gcp-backend   19m   Valid    ReadWrite      True
$ kubectl get externalsecret -n blog-app mongodb-creds
NAME          STORE       REFRESHINTERVAL STATUS       READY
mongodb-creds gcp-backend 1h0m0s          SecretSynced True
$ kubectl get secret -n blog-app mongodb-creds
NAME            TYPE     DATA   AGE
mongodb-creds   Opaque   2      4m45s
```

The same should be reflected in the `blog-app` application on Argo CD, and the application should come up clean, as shown in the following screenshot:

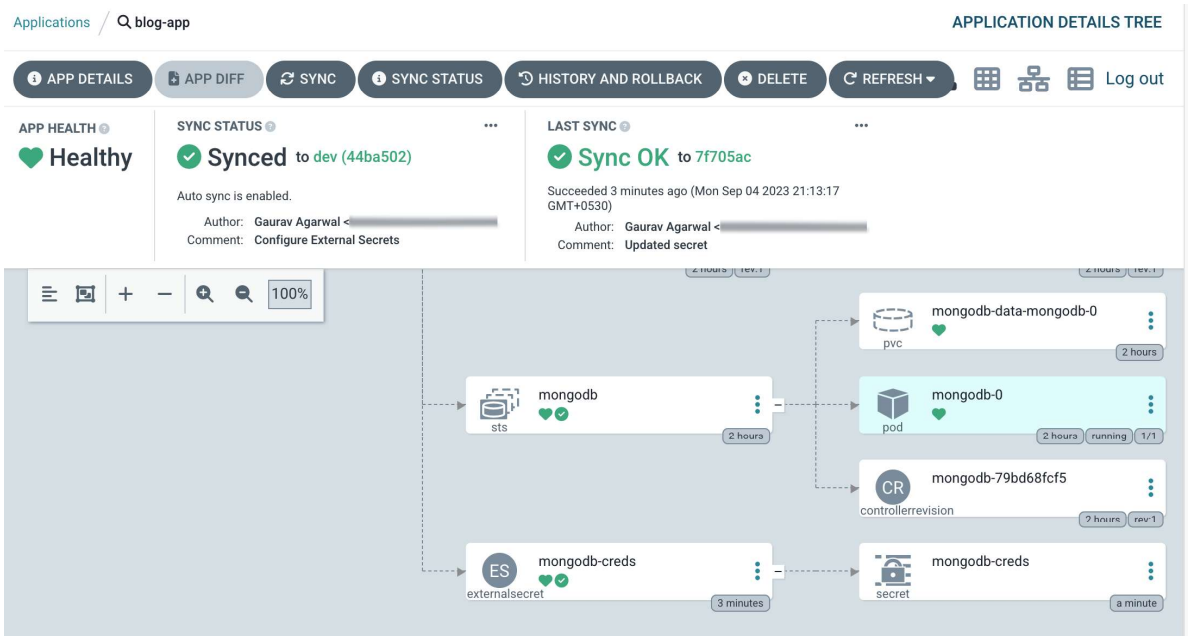

Figure 13.8 – blog-app showing as Healthy

You can then access the application by getting the frontend service external IP using the following command:

```
$ kubectl get svc -n blog-app frontend
NAME     TYPE          EXTERNAL-IP   PORT(S)       AGE
frontend LoadBalancer 34.122.58.73 80:30867/TCP 153m
```

You can access the application by visiting `http://<EXTERNAL_IP>` from a browser:

Welcome to My Blog

Figure 13.9 – Blog App home page

And as we can see, we can access the Blog App successfully. That is proper secret management, as we did not store the secret in the source code repository (Git). We did not view or log the secret while applying it, meaning there is no trace of this secret anywhere in the logs, and only the application or people who have access to the namespace where this application is running can access it. Now, let's look at another crucial aspect: testing your application.

Testing your application within the CD pipeline

Until now, we've deployed our application on a Kubernetes cluster and manually verified that it is running. We have two options moving forward: either proceed with manual testing or create automated tests, also known as a **test suite**. While manual testing is the traditional approach, DevOps heavily emphasizes automating tests to integrate them into your CD pipeline. This way, we can eliminate many repetitive tasks, often called **toil**.

We've developed a Python-based integration test suite for our application, covering various scenarios. One significant advantage of this test suite is that it treats the application as a black box. It remains unaware of how the application is implemented, focusing solely on simulating end user interactions. This approach provides valuable insights into the application's functional aspects.

Furthermore, since this is an integration test, it assesses the entire application as a cohesive unit, in contrast to the unit tests we ran in our CI pipeline, where we tested each microservice in isolation.

Without further delay, let's integrate the integration test into our CD pipeline.

CD workflow changes

Till now, we have the following within our CD workflow:

```
.
├── create-cluster.yml
├── dev-cd-workflow.yaml
└── prod-cd-workflow.yaml
```

Both the Dev and Prod CD workflows contain the following jobs:

```
jobs:
  create-environment-and-deploy-app:
    name: Create Environment and Deploy App
    uses: ./.github/workflows/create-cluster.yml
    secrets: inherit
```

As we can see, the step calls the `run-tests.yml` workflow. That is the workflow that will be doing the integration tests. Let's look at the workflow to understand it better:

```
name: Run Integration Tests
on: [workflow_call]
jobs:
  test-application:
    runs-on: ubuntu-latest
    defaults:
      run:
        working-directory: ./tests
    steps:
    - uses: actions/checkout@v2
    - name: Extract branch name
      run: echo "branch=${GITHUB_HEAD_REF:-${GITHUB_REF#refs/heads/}}" >> $GITHUB_OUTPUT
      id: extract_branch
    - id: gcloud-auth
      name: Authenticate with gcloud
      uses: 'google-github-actions/auth@v1'
      with:
        credentials_json: '${{ secrets.GCP_CREDENTIALS }}'
    - name: Set up Cloud SDK
      id: setup-gcloud-sdk
      uses: 'google-github-actions/setup-gcloud@v1'
    - name: Get kubectl credentials
      id: 'get-credentials'
      uses: 'google-github-actions/get-gke-credentials@v1'
      with:
        cluster_name: mdo-cluster-${{ steps.extract_branch.outputs.branch }}
        location: ${{ secrets.CLUSTER_LOCATION }}
    - name: Compute Application URL
      id: compute-application-url
      run: external_ip=$(kubectl get svc -n blog-app frontend --output jsonpath='{.status.
loadBalancer.ingress[0].ip}') && echo ${external_ip} && sed -i "s/localhost/${external_
ip}/g" integration-test.py
    - id: run-integration-test
      name: Run Integration Test
      run: python3 integration-test.py
```

The workflow performs the following tasks:

1. It is triggered exclusively through a `workflow` call.
2. It has the `./tests` working directory.
3. It checks out the committed code.
4. It installs the `gcloud` CLI and authenticates with Google Cloud using the `GCP_CREDENTIALS` service account credentials.
5. It connects `kubectl` to the Kubernetes cluster to retrieve the application URL.
6. Using the application URL, it executes the integration test.

Now, let's proceed to update the workflow and add tests using the following commands:

```
$ cp -r ~/modern-devops/ch13/integration-tests/.github .
$ cp -r ~/modern-devops/ch13/integration-tests/tests .
$ git add --all
$ git commit -m "Added tests"
$ git push
```

This should trigger the Dev CD GitHub Actions workflow again. You should see something like the following:

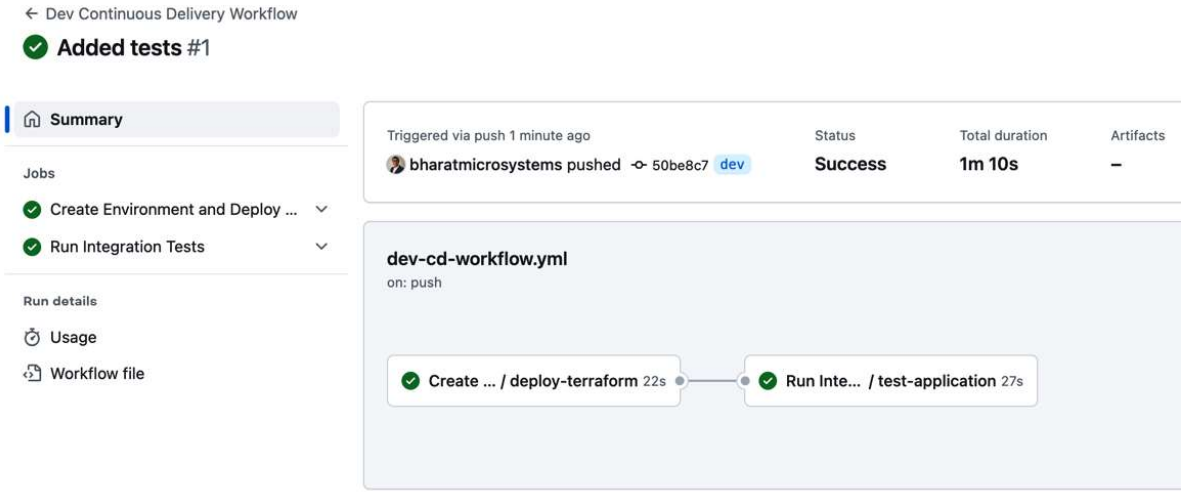

Figure 13.10 – Added tests workflow run

As we can see, there are two steps in our workflow, and both are now successful. To explore what was tested, you can click on the **Run Integration Tests** step, and it should show you the following output:

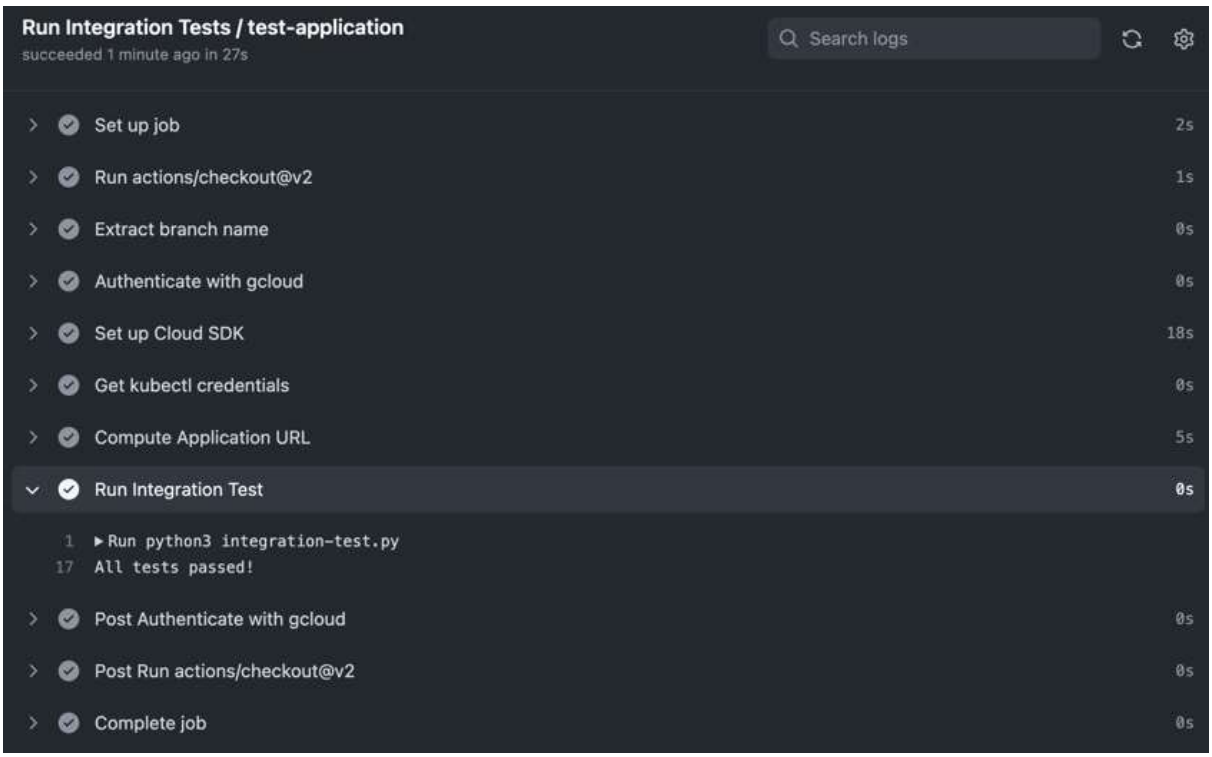

Figure 13.11 – The Run Integration Tests workflow step

As we can see, the **Run Integration Tests** step reports that all tests have passed.

While images are being built, deployed, and tested using your CI/CD toolchain, there is nothing in between to prevent someone from deploying an image in your Kubernetes cluster. You might be scanning all your images for vulnerabilities and mitigating them, but somewhere, someone might bypass all controls and deploy containers directly to your cluster. So, how can you prevent such a situation? The answer to that question is through binary authorization. Let's explore this in the next section.

Binary authorization

Binary authorization is a deploy-time security mechanism that ensures that only trusted binary files are deployed within your environments. In the context of containers and Kubernetes, binary authorization uses signature validation and ensures that only container images signed by a trusted authority are deployed within your Kubernetes cluster.

Using binary authorization gives you tighter control over what is deployed in your cluster. It ensures that only tested containers and those approved and verified by a particular authority (such as security tooling or personnel) are present in your cluster.

Binary authorization works by enforcing rules within your cluster via an admission controller. This means you can create rulesets only to allow images signed by an attestation authority to be deployed in your cluster. Your **quality assurance** (**QA**) team can be a good attestor in a practical scenario. You can also embed the attestation within your CI/CD pipelines. The attestation means your images have been tested and scanned for vulnerabilities and have passed a minimum standard to be ready to be deployed to the cluster.

GCP provides binary authorization embedded within GKE, based on the open source project **Kritis** (`https://github.com/grafeas/kritis`). It uses a **public key infrastructure** (**PKI**) to attest and verify images—so your images are signed by an attestor authority using the private key, and Kubernetes verifies the images by using the public key. The following diagram explains this beautifully:

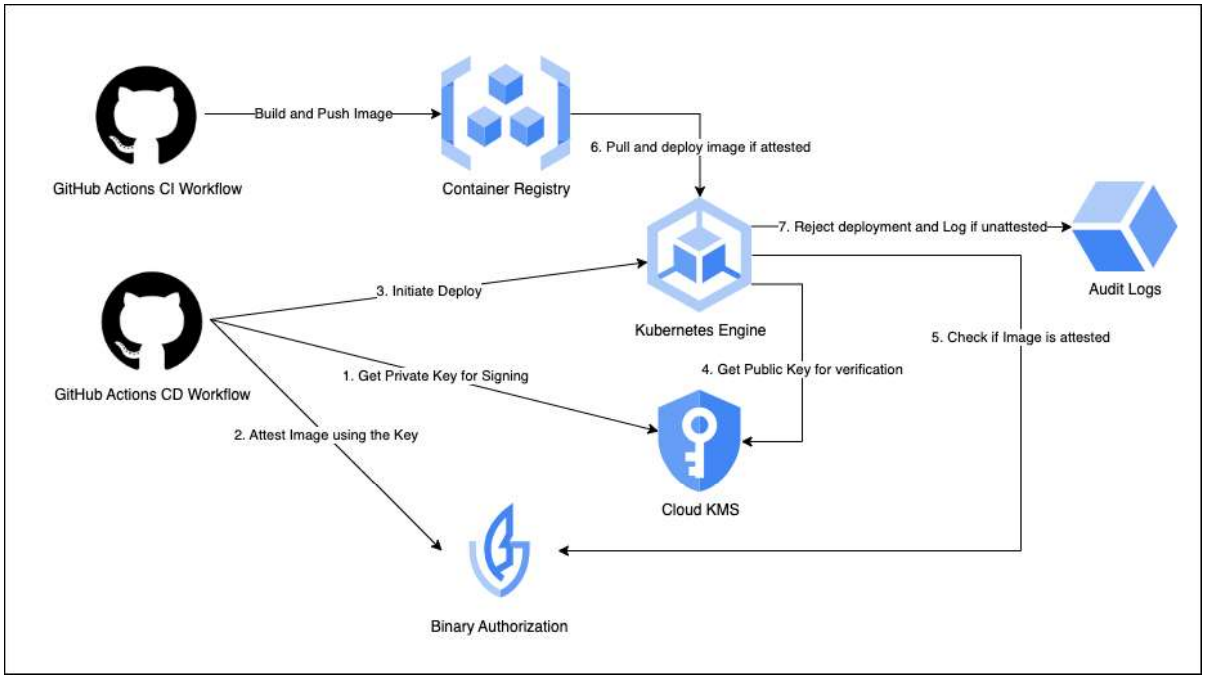

Figure 13.12 – Binary authorization process

In the hands-on exercise, we will set up binary authorization and a PKI using Google Cloud KMS. Next, we will create a QA attestor and an attestation policy for all binary auth-enabled GKE clusters, ensuring that only attested images can be deployed. Since our application is now tested, the next step is to attest the tested images. So, let's proceed to set up binary authorization within our Dev CD workflow in the next section.

Setting up binary authorization

As we're using GitOps right from the beginning, we will use Terraform to set up binary authorization for us. We'll start by setting up some GitHub Actions secrets. Go to `https://github.com/<your_github_user>/mdo-environments/settings/secrets/actions` and create the following secrets:

```
ATTESTOR_NAME=quality-assurance-attestor
KMS_KEY_LOCATION=us-central1
KMS_KEYRING_NAME=qa-attestor-keyring
KMS_KEY_NAME=quality-assurance-attestor-key
KMS_KEY_VERSION=1
```

We'll then create a `binaryauth.tf` file with the following resources.

We'll begin by creating a Google KMS key ring. Since binary authorization utilizes PKI for creating and verifying attestations, this key ring will enable our attestor to digitally sign attestations for images. Please note the `count` attribute defined in the following code. This ensures that it is created exclusively in the `dev` environment, where we intend to use the attestor for attesting images after testing our app:

```
resource "google_kms_key_ring" "qa-attestor-keyring" {
  count = var.branch == "dev" ? 1 : 0
  name     = "qa-attestor-keyring"
  location = var.region
  lifecycle {
    prevent_destroy = false
  }
}
```

We will then use a Google-provided `binary-authorization` Terraform module to create our `quality-assurance` attestor. That attestor uses the Google KMS key ring we created before:

```
module "qa-attestor" {
  count = var.branch == "dev" ? 1 : 0
  source = "terraform-google-modules/kubernetes-engine/google//modules/binary-
authorization"
  attestor-name = "quality-assurance"
  project_id    = var.project_id
  keyring-id    = google_kms_key_ring.qa-attestor-keyring[0].id
}
```

Finally, we will create a binary authorization policy that specifies the cluster's behavior when deploying a container. In this scenario, our objective is to deploy only attested images. However, we will make a few exceptions, allowing Google-provided system images, Argo CD, and External Secrets Operator images. We will set the `global_policy_evaluation_mode` attribute to `ENABLE` to avoid enforcing the policy on system images managed by Google.