

Kubernetes Secrets Management

Alex Soto Bueno
Andrew Block



MANNING



Securing Kubernetes Secrets V06

1. [MEAP VERSION 6](#)
2. [Welcome](#)
3. [1_Kubernetes_Secrets](#)
4. [2_Reintroducing_Kubernetes_and_Secrets](#)
5. [3_Securely_Storing_Secrets](#)
6. [4_Encrypting_Data_at_Rest](#)
7. [5_Hashicorp_Vault_and_Kubernetes](#)
8. [6_Accessing_Cloud_Secret_Stores](#)
9. [7_Kubernetes-Native_Continous_Integration_and_Secrets](#)
10. [8_Kubernetes-Native_Continuous_Delivery_and_Secrets](#)
11. [Appendix A. Tooling](#)
12. [Appendix B. Installing and Configuring yq](#)
13. [Appendix C. Installing and Configuring Pip](#)
14. [Appendix D. Installing and Configuring Git](#)
15. [Appendix E. Installing gpg](#)

Kubernetes Secrets Management

Alex Soto Bueno
Andrew Block

MEAP

MANNING



MEAP VERSION 6

 MANNING PUBLICATIONS

Welcome

Thanks for purchasing the MEAP of *Kubernetes Secrets Management*. I hope that what you'll get access to will be of immediate use to you and, with your help, the final book will be great!.

This book is written for developers and operators familiar with Kubernetes, CLI tooling, Git, and basic knowledge about software development and YAML files.

Security is a continuously evolving topic. Each week, news of a new vulnerability is reported along with intrusions being detected or achieved. One of the reasons for adopting a container native strategy is the many security benefits of cloud-native application development and operation. As containers are becoming more and more popular, Kubernetes becomes the de-facto container orchestrator.

By the conclusion of this book, you will better understand the role that Secrets play in a Kubernetes environment along with way technologies and approaches that can be used to manage sensitive resources within applications properly.

The book is divided into three parts. Part 1 will fly through the basics of Kubernetes that you'll need in the following chapters: a Pod, a Deployment, how to create ConfigMaps and Secrets and injecting them into a container, and how to deal with Volumes.

Part 2 will continue to go deep on securing secrets correctly, either when stored in the source control system or consumed in the production environment. Part 3 will introduce the concept of Kubernetes-Native continuous integration and delivery and how to manage secrets during the whole pipeline.

Please let me know your thoughts in the [liveBook Discussion forum](#) on what's been written so far and what you'd like to see in the rest of the book.

Your feedback will be invaluable in improving *Securing Kubernetes Service*.

Thanks again for your interest and for purchasing the MEAP!

- Alex Soto Bueno & Andrew Block

In this book

[MEAP VERSION 6](#) [About this MEAP](#) [Welcome](#) [Brief Table of Contents 1](#)
[Kubernetes Secrets 2](#) [Reintroducing Kubernetes and Secrets 3](#) [Securely Storing Secrets 4](#) [Encrypting Data at Rest 5](#) [HashiCorp Vault and Kubernetes 6](#) [Accessing Cloud Secret Stores 7](#) [Kubernetes-Native Continous Integration and Secrets 8](#) [Kubernetes-Native Continuous Delivery and Secrets Appendix A. Tooling](#) [Appendix B. Installing and Configuring yq](#) [Appendix C. Installing and Configuring Pip](#) [Appendix D. Installing and Configuring Git](#) [Appendix E. Installing gpg](#)

1 Kubernetes Secrets

This chapter covers

- A focus on Security
- Taking Full Advantage of the Kubernetes Ecosystem
- Everything is not a Secret
- Bringing it all together
- What you will learn
- Tools to get Started

“I’ve got a secret”; the name of the 1950’s American game show for which a contestant hid an interesting detail of their life from a panel of celebrities attempting to uncover the fact. Technology features many of the same parallels as the contestant from the game show as there are a wide range of attributes that make up any system, including those that may be more sensitive and should not be seen or known to outside parties. While we have come a long way from the days where computers filled entire rooms (like the ones that most likely produced the aforementioned game show), one constant that has remained the same is that systems will always rely on some form of configuration data to support their normal operation.

Configurations can take many forms and be applied in a variety of ways depending on the use case and context. In a software application, examples of configurable properties could include details to support the application framework, or for the normal operation of the program. While many of these properties are intended to be viewed by any party, there are certain attributes, such as passwords, that should only be seen or accessed by certain individuals. Kubernetes is a container orchestration platform, and as one might expect with any complex system, a myriad of properties are available that can be applied to support the normal operation of the foundational infrastructure as well as any end user actions; some of which may contain sensitive information that if exposed, risk the integrity of the entire platform.

Over time, an increasing number of options have become available for

managing configurations in Kubernetes, but most trace their history to the two primary methods for storing configurations within the platform: ConfigMaps and Secrets. While each resource provides a way for storing configuration material through the use of key/value pairs, the primary difference between the two is that Secrets are intended to be used by assets that are meant to be secretive. As a result, additional mechanisms are employed by the Kubernetes Secret construct itself in order to obscure the content of the underlying property.

However, the included Kubernetes Secret resource is just the tip of the iceberg as it relates to what it takes to properly manage how sensitive resources are stored and used within Kubernetes. Additional tools and approaches have evolved over time to supplement or provide alternate options. One of the motivations is partially due to the fact that the native Kubernetes Secrets resource does not provide the level of security that one might need or expect from a secrets management system. Instead of making use of a proper encryption algorithm, values stored within Kubernetes Secrets are merely Base64 encoded, where their values can be easily decoded by a malicious attacker. But, what truly makes these alternate tools more superior to the native Kubernetes Secrets resource? Is it a more robust encryption mechanism, ease of use, or how it integrates into the target system or end application? These are all factors that need to be considered at the onset, and in many cases, there is no “correct” answer. Everyone, whether it be an individual developer, or multinational organization, has a different classification for how they assess security. What may be fine for one may not be fine for the other, especially when there are required regulations that must be followed pertaining to security.

1.1 A focus on Security

Security is a continuously evolving topic. Each week, news of a new vulnerability is reported along with intrusions being detected or achieved. One of the reasons for adopting a container native strategy is due to the many security benefits that come along cloud-native application development and operation. The fundamental shift to how systems are built and deployed gives organizations an opportunity to reflect on their current security practices and contemplate how they want to design and deliver their strategies and policies

moving forward. Adopting concepts, like the Principle of Least Privilege, which limits the amount of access granted to resources to only the minimum necessary privileges in order to accomplish the task, embraces the importance of managing access to sensitive resources. Applying Role Based Access Control (RBAC) to restrict access to authorized users is one of the most common approaches for applying this principle. It is important that these policies be reviewed on an ongoing basis to confirm that any actor still needs access to the desired resource. This continual assessment not only employs good practices, but increases the overall security level.

1.2 Taking Full Advantage of the Kubernetes Ecosystem

As one can probably imagine, proper secret management in a Kubernetes environment goes far beyond the simple deployment of Pods and Services. More advanced topics, such as sidecars and admission webhooks push Kubernetes to the limit in order to achieve secure solutions. The true extensibility of the platform is demonstrated by extending the base set of API resources found within the platform and enabling the ability to define new resources dedicated to the management of secure assets. Through the hard work of members from the Open Source community and organizations dedicated to provide solutions in this space, additional options and approaches have become available to provide the most secure operating environment available.

1.3 Everything is not a Secret

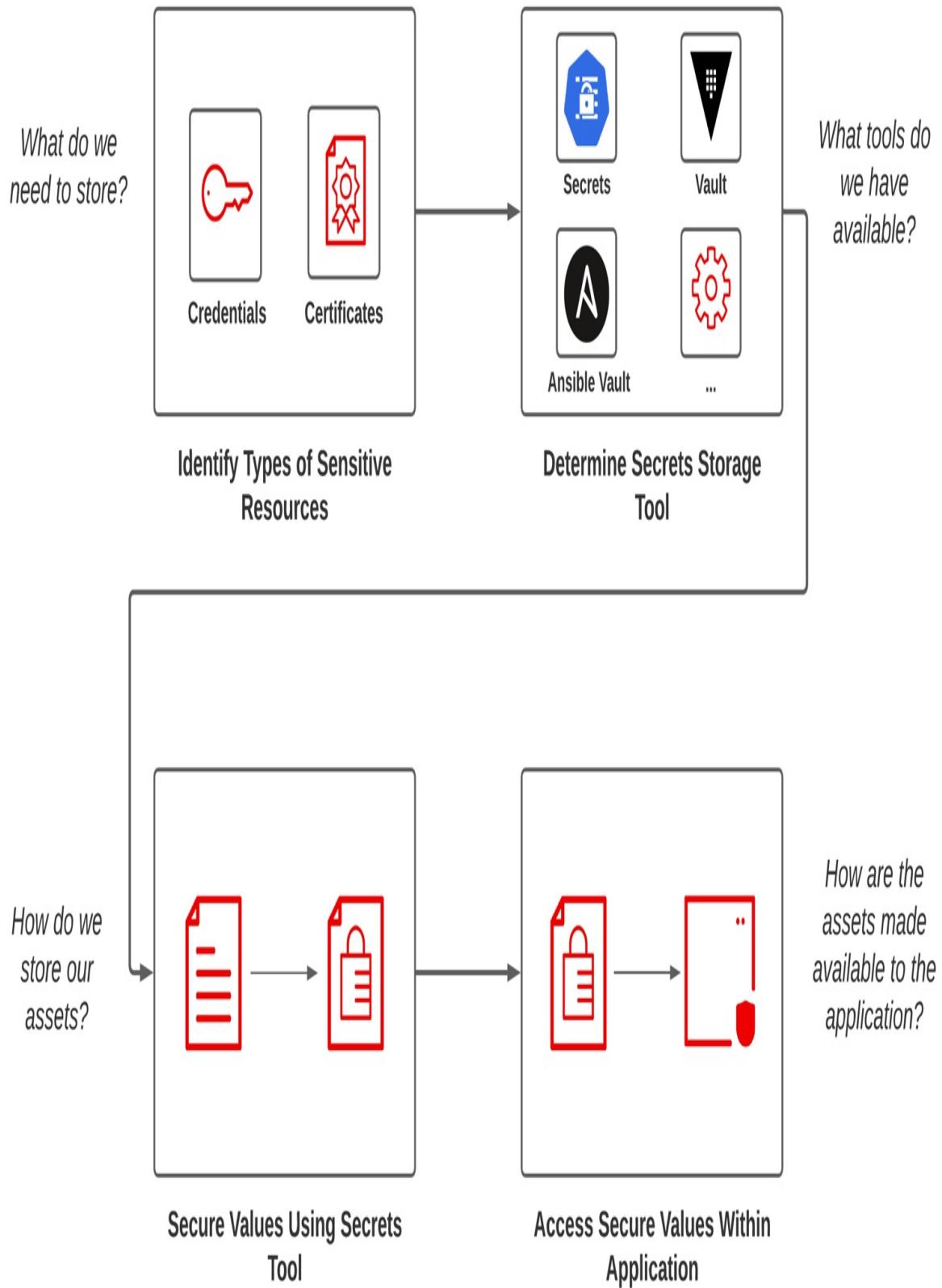
While it is important to have a security first mindset, one must be considerate of the effort that it takes to properly perform secret management. For example, if you developed an application that sends alerts based on the current temperature once a configurable threshold was reached, such as if the temperature of a refrigerator reached 5 degrees celsius, would this threshold of 5 degrees celsius be truly considered a sensitive resource? Not really. The time, resources, and effort to fully secure and manage assets must be taken into account as the administrative overhead may outweigh the benefits. The mindset that every configurable value should be treated in the same regard is

a common fallacy faced by many teams. Determining which values can remain in plain text while others that require protection is an exercise that should be performed by all teams. Defining a standardized method for identifying sensitive resources will help align not only how application teams approach secret management, but also provide guidance for them to make more informed decisions and to reduce wasted time.

1.4 Bringing it all together

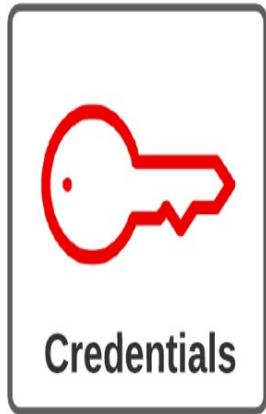
Secrets management has been a challenge long before the days of Kubernetes. The process can be laborious and will most likely involve careful planning and consideration. So, what are these steps?

Figure 1.1. Kubernetes Secrets Management Process

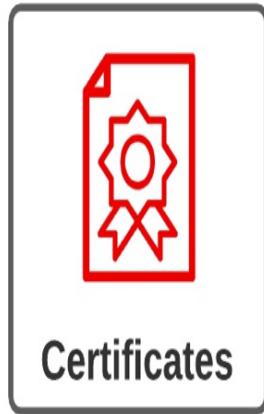


From this birdseye perspective, lets break down each section. First, prior to any sort of implementation or execution, there should be an agreement on what types of configurable material is present, as well as those that are to be deemed sensitive.

Figure 1.2. Identifying Kubernetes Secrets



Credentials



Certificates



Application Configurations

- Passwords
- API Keys

- TLS
- GPG Keys

- Runtime Arguments



What are types of sensitive resources?

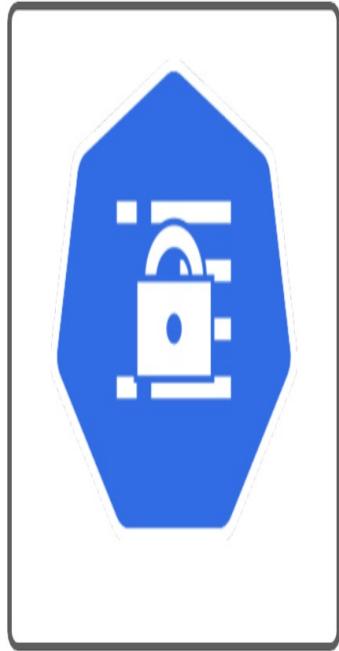
In some cases, the answer is fairly straightforward, such as a password to a database. For others, it may not be so easy. Take an example of a web application that makes a connection to the same backend database. Would the database hostname be considered sensitive material? To some, it may be, as there could be a desire to obfuscate the location in order to reduce the potential attack vector. However, if the location referred to a common shared database used by members of a team in a development environment, it may not. Many battles have been fought, won and lost throughout this entire process and the answer does in fact vary.

Once there has been an agreement on the types of properties that have been deemed sensitive, the next step is to determine how they should be stored. There are several factors that should be taken into account:

1. What are the available secrets management solutions available?
2. How is the sensitive asset intended to be used?
3. What application framework is being used and what are the options for injecting external configurations?

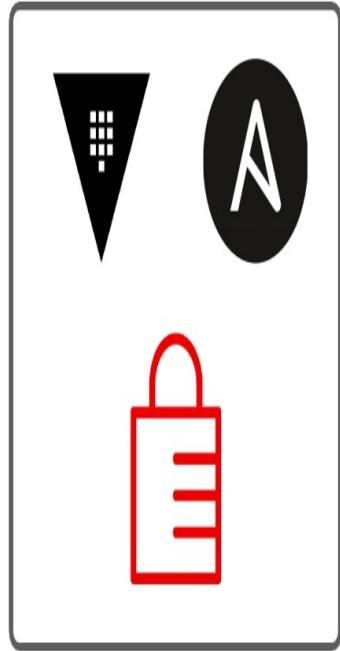
These areas are illustrated in the following diagram:

Figure 1.3. Secrets Management Utilities



Kubernetes Secrets

1. Kubernetes Secrets is the defacto method for storing sensitive material in Kubernetes



Third Party Tools

2. Alternate methods can be used to both store and secure sensitive material for use in Kubernetes

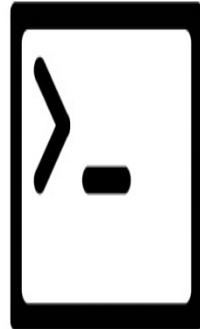
Secrets Management Utilities

In a Kubernetes environment, the Secret resource is a natural default as it does provide some form of protection as well as it is included in any distribution of Kubernetes. But, what else is available? If the framework for your application or component offers a secrets management system, it may be a viable option. Many cloud providers also offer their own Key Management System (KMS) for use as well for a managed service option. This is highly appetizing when operating in cloud environments. Other solutions, such as Hashicorp Vault, provide a Key Management system that can be deployed anywhere, whether it be in the public cloud or elsewhere.

After determining the secrets management solution to move forward with, the next step is to store assets within the tool. While the process of storing resources can vary depending on the tool, most expose an API based interface that can be used to integrate at a variety of levels. The API becomes the focal point for interacting with the secrets management solution and for convenience sakes, a Command Line Interface option or a user interface exposed through a web browser abstracts the underlying interaction with the API. These options all aid in the storage of secrets can be achieved in a manual fashion or integrated into a Continuous Integration or Continuous Delivery (CI/CD) process to achieve repeatably and consistency.

Figure 1.4. Interacting With Secrets Management Tools

CLI



Interactive way to work
with sensitive resources

API



Exposes integrations, such as
UI and RESTful services, to
manage sensitive resources

Working with Secrets

Finally, and arguably the most important step is retrieving the stored resource out of the Secrets Management tool for use by the application. Thanks to the power of Kubernetes, there are a variety of options and approaches that can be used. Ultimately, it boils down to two distinct steps:

1. Translating the value from its protected form to plain text
2. Exposing the value so that it can be consumed by the application

The process in which stored values are converted back to their plain text representation is dependent on the secrets management tool in use. In many cases, the same approach that was used to store the values can be used, just in the reverse fashion. Where things can get *interesting* is how these values are made available to applications. In its simplest form, such as with standard Kubernetes secrets, a reference is made to the stored Secret in the Kubernetes manifest being created, where the asset is then exposed to applications as either an environment variable or contained within a file on the filesystem of the application. However, if more advanced secrets management tools are used, or there is a desire for more dynamic capabilities to further restrict how values are exposed to applications, additional options may be available. Certain tools can be colocated alongside running applications, called sidecars, to interact with the secrets management store and inject sensitive values into applications for consumption. Alternatively, at deployment time, the Kubernetes manifest of the application can be modified by the platform to decouple how values are injected in a dynamic fashion. Furthermore, approaches are also available where the sensitive value is never exposed in plaintext and is accessed directly by the application in memory. From start to end, regardless of the approach, careful thought and planning needs to occur to assess the necessary tools that are available, requirements from the application, and then finally the overall time and effort that it will take to implement the solution.

1.5 What you will learn

By the conclusion of this book, you will have a better understanding of the role that Secrets play in a Kubernetes environment along way technologies and approaches that can be used to properly manage sensitive resources

within applications. This book is geared toward individuals that interact with Kubernetes environments on a regular basis to deploy their applications in a full stack capacity and oversee the full lifecycle of applications from development to production. Most importantly, they also have a desire to learn how to properly manage configurations in a secure platform. Starting with the base platform itself and working outward, you will first learn about the set of components that are natively included in all Kubernetes deployments to support storing and securing sensitive resources. From there, you will learn how to manage sensitive resources at rest including options for providing alternate methods for securing assets. Data at rest extends beyond the scope of deployed applications and also includes the primary datastore of the platform. Hashicorp Vault will then be introduced as an enterprise secrets management tool that can be used to protect both the most sensitive infrastructure resources as well as those deployed by any microservices application deployed to the platform. Finally, you will learn how to manage sensitive resources within a Kubernetes Native application along with the considerations that need to be taken into account as applications are delivered from development through to production as part of a Continuous Integration and Continuous Delivery (CI/CD) pipeline. By the end, you will have a thorough understanding of the various options available in the Kubernetes ecosystem for managing sensitive resources and be able to make informed choices based on your particular use case.

1.6 Tools to Get Started

To guide you along your journey throughout this book, you will make use of several tools that not only interact with a Kubernetes cluster, but any of the secrets management solutions that are being discussed. It is important that you have an environment that allows for the installation and configuration of software. At a minimum, to interact with a Kubernetes cluster and manage the Kubernetes Secret resource, the Kubernetes Command Line Tool (`kubectl`) will be needed. As we work through some of the alternate solutions in the secrets management space, additional tools will be introduced. So, without further delay, let's get started!

1.7 Summary

This chapter covered:

- Applications make use of configuration details that contain sensitive information for which protections should be applied.
- The importance of security in the technology industry.
- An overview of Kubernetes Secrets to aid in the storage of restricted material.
- Situations where it may not be necessary to use Kubernetes Secrets
- The end to end process for managing Secrets in Kubernetes.
- The tools to begin managing secure configurations in Kubernetes.

2 Reintroducing Kubernetes and Secrets

This chapter reviews some important Kubernetes concepts that we need to know to correctly create and manage secrets to a Kubernetes cluster to finally create our first secret. The concepts that we'll be covering in this chapter are

- Understanding the basic architecture of a Kubernetes cluster
- Deploying an application to Kubernetes
- Managing application configuration externally
- Using Kubernetes Secrets to store sensitive information

If you are already well versed in these concepts then you can skip the following sections and get straight to Secrets section.

The deployed application in this chapter is a very simple RESTful Web Service that returns a greeting message. The service returns a default greeting message but it can be configured externally too either by using an environment variable or a properties configuration file.

Kubernetes cluster

We'll need a Kubernetes cluster to run the examples of this book. You can use any Kubernetes distribution that is provided either by any public cloud or to run it locally.

In this book, the examples are tested using a Minikube cluster. Minikube allows us to run Kubernetes locally in a "single-node" Kubernetes cluster inside a Virtual Machine (VM) on a laptop. Check Appendix to learn how to install it.

2.1 Kubernetes Architecture

The first thing to understand about Kubernetes architecture is that there are two kinds of nodes, master and worker nodes, and in typical production deployments you might have several nodes of each kind.

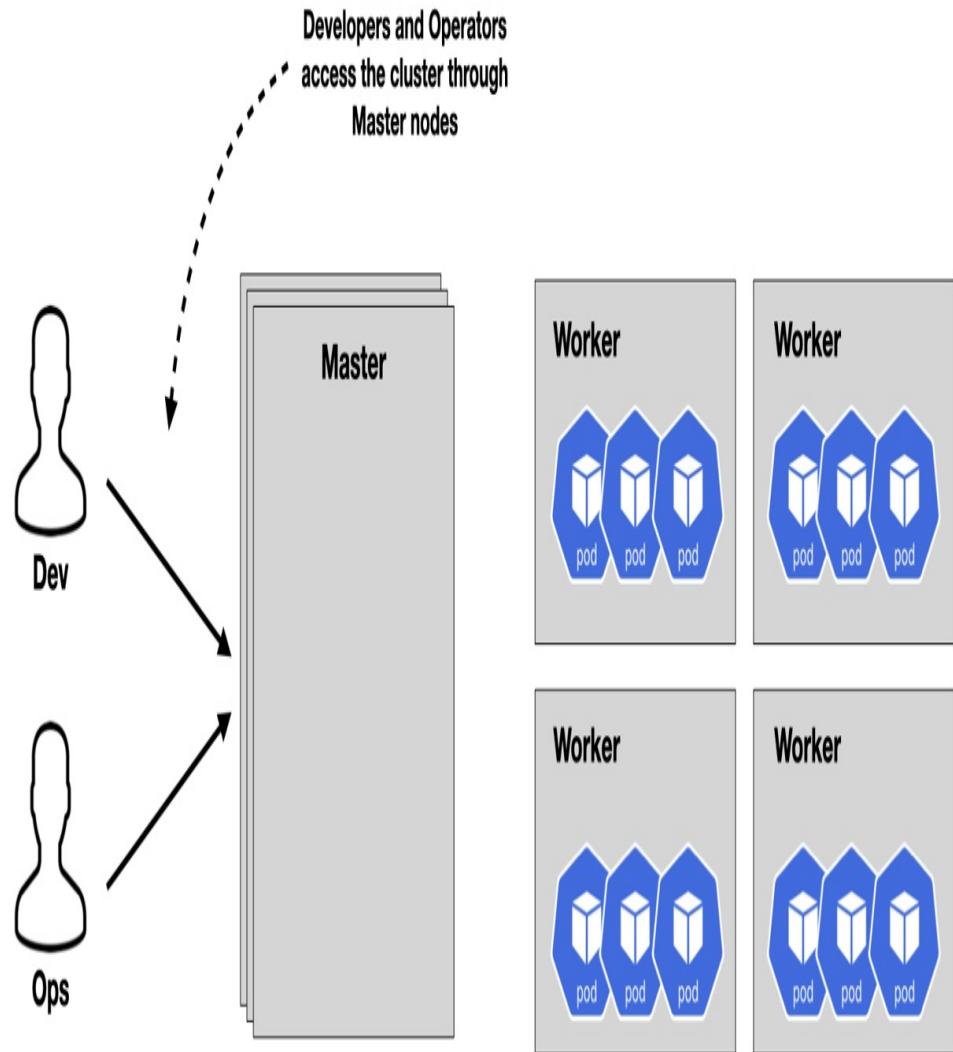
In [2.1](#), you can see an overview of a Kubernetes cluster and the relationship between master nodes and worker nodes.

The worker node(s) are responsible for running our workloads such as developed services or databases.

The master node(s) manages the worker nodes and is responsible for deciding worker nodes where workloads are deployed.

The minimum nodes required to conform to a Kubernetes is one master node acting as master and worker node. Although this might not be the typical use case in production, it is when developing in the local machine. Usually, in production, you might have from 3 to 5 master nodes and several worker nodes which might depend on the number of workloads to deploy and the degree of redundancy that you expect on your application.

Figure 2.1. Kubernetes Architecture



Let's explore what is inside master and worker nodes.

2.1.1 What is a Master Node?

A master node is responsible for controlling the whole cluster; for example, it decides where the workloads are deployed, detects and responds to abnormalities that might happen on the cluster (ie number of replicas are not satisfied),

stores information of the state of the cluster so we can monitor the state of the cluster and finally it is the entry point of the Kubernetes

A Kubernetes cluster must have at least one master node, but we will have more than one for redundancy in production environments.

We find the next four elements inside each master node:

- kube-apiserver
It is the front end for Kubernetes and exposes the Kubernetes API to the Kubernetes users. When an operator runs a command against the Kubernetes cluster, it does through the api-server.
- etcd
A key-value database used to store all cluster data. Every time you get information about the cluster, that data is retrieved from etcd.
- scheduler
It is the process responsible of selecting a node for running workloads on. Factors that are taken into consideration to select the node to deploy our workload might depend on its requirements such as hardware, policy constraints, affinity and anti-affinity, data locality, ...
- controllers

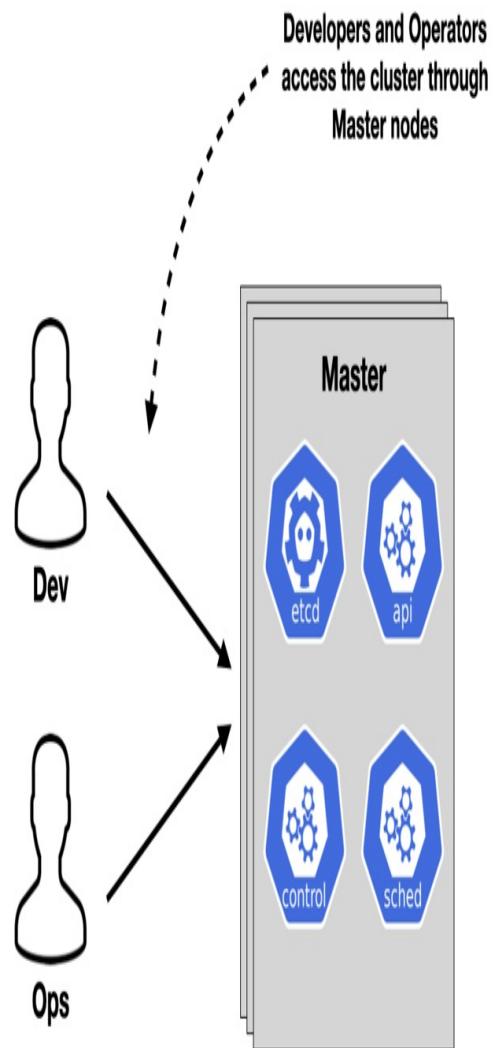
The main task of controllers are monitoring specific Kubernetes resources. There are 4 major controllers:

1. The **node controller** is responsible for monitoring and acting when any node goes down.

2. The **replication controller** is responsible for maintaining our workloads up and running all the time.
3. The **endpoint controller** makes possible to access the workloads with an static IP and DNS name.
4. The **service account and token controllers** creates default accounts and tokens for new namespaces.

In [2.2](#) you can see all the elements that conforms a master node.

Figure 2.2. Elements of a Master Node



Now that we know the parts of a master node, let's see the parts of a worker node.

2.1.2 What is a Worker Node?

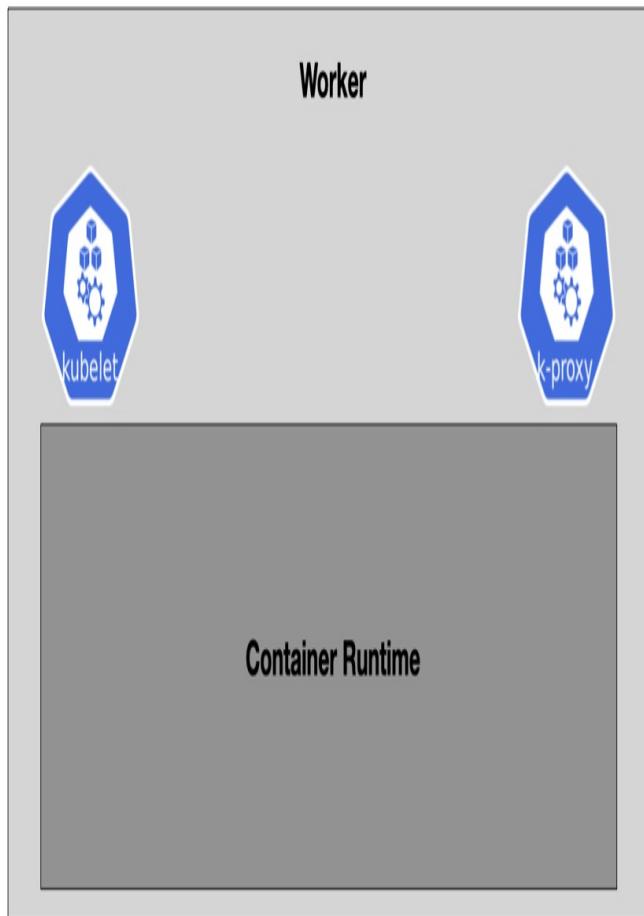
A worker node is the physical place where our workloads are deployed and run. Since workloads in Kubernetes are software containers, the container runtime is hosted inside each of the worker node.

Any worker node is composed by the following three elements:

- kubelet
It is an agent that ensures that containers are running in a Pod.
- proxy
A network proxy that implements part of the Kubernetes Service concept.
- container runtime
It is the responsible for running containers. At the time of writing this book, the following runtimes are supported: Docker, containerd, CRI-O, and any implementation of the Kubernetes CRI.

In [2.3](#) you can see all the elements that compose a worker node:

Figure 2.3. Elements of a Worker Node



Now that we've got a good understanding of the Kubernetes architecture let's start using Kubernetes from a developer or an operator point of view.

2.2 Deploying workloads in Kubernetes

So far, you've seen the architecture of a Kubernetes cluster, but as developers what we want to do is to deploy a web service, a database or a message broker, or any other element required by our application into Kubernetes. Also, we want to be able to configure them externally from the platform or be able to access them using a network protocol.

In the following sections, we are going to explore how a developer can interact with Kubernetes by deploying a simple application that returns a welcome message.

2.2.1 Deploying a workload

One of the most important Kubernetes resources to deploy a workload to the cluster is a *Pod*.

Pods are the smallest deployable unit in Kubernetes. A Pod is composed of a group of one or more containers where each of them shares the *IP*, shared storage, shared resources, and shared lifecycle.

Pods are the units where business workloads are running; for example, a service API, a database, a mailing server. One analogy that is helpful to understand what is a Pod is to think that it's a sort of a virtual machine (of course it isn't), that runs processes, where each process shares the resources, network, and lifecycle of the Virtual Machine. In a Pod it's the same concept but instead of running "processes", it runs containers.

There are many ways to deploy a Pod in a Kubernetes cluster, but one of the most used way in our opinion is by describing in a *YAML* file and apply it using `kubectl` CLI tool.

To create a *YAML* file containing the Pod definition, open a new terminal

window, and in a new directory create a file called `greeting-pod.yaml` defining the container image that belongs to the Pod as shown in [2.1](#):

Listing 2.1. Creating a Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: greeting-demo
spec:
  containers:
    - name: greeting-demo
      image: quay.io/lordofthejars/greetings-jvm:1.0.0
```

Once we created the file, we can apply it to the cluster by using `kubectl apply` subcommand:

```
kubectl apply -f greeting-pod.yaml
```

We need to wait until the Pod is allocated into a node and it is ready to be accessed. To do that, use `kubectl wait` subcommand in the terminal:

```
kubectl wait --for=condition=Ready pod/greeting-demo
```

Now validate that the Pod is allocated correctly by getting Pods status:

```
kubectl get pods
```

A Pod is allocated into a node and starts correctly as the final status is `Running`.

NAME	READY	STATUS	RESTARTS	AGE
greeting-demo	1/1	Running	0	18s #1

Let's do an experiment and delete the Pod we've created previously and see what's happening with its lifecycle:

```
kubectl delete pod greeting-demo
```

Wait a few seconds until the Pod is terminated and then get the Pod status:

```
kubectl get pods
```

The output (if you've waited enough time) will show you that greeting-demo Pod is no longer available.

```
No resources found in default namespace.
```

We can run the `kubectl get pods` as many times as we want, but trust us the Pod disappears forever.

So a Pod itself might not be useful in most of the cases as if the service dies for any reason, it will become unavailable until we redeploy again.

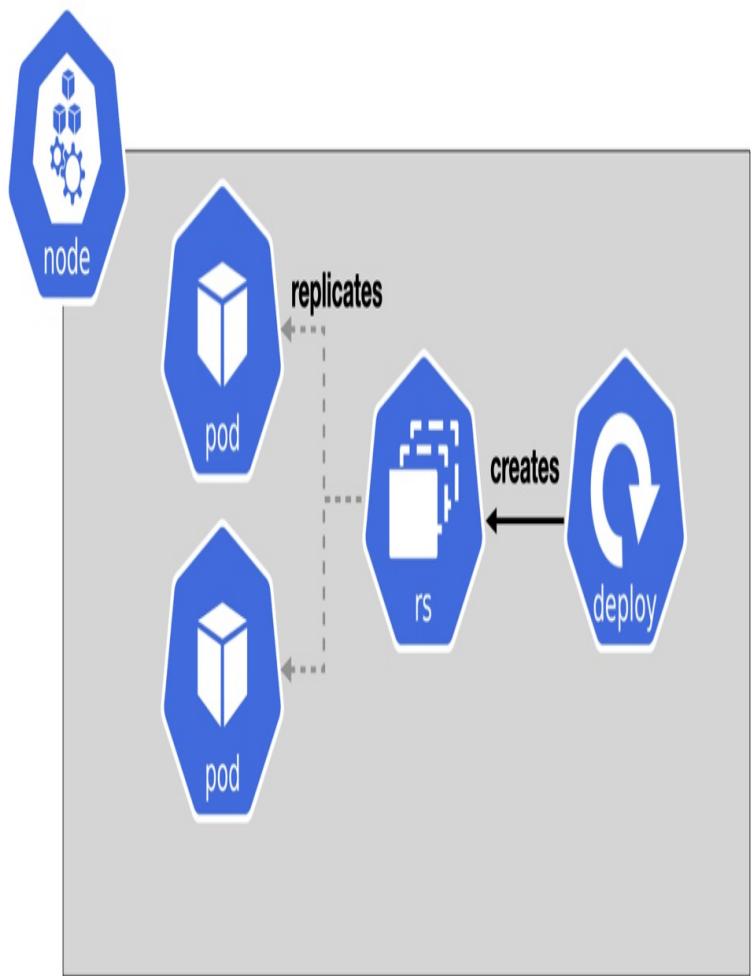
To restart automatically a Pod, we need to create a deployment.

2.2.2 Deployment object

Up until now when a Pod dies, it is not restarted automatically. This is because of the nature of a Pod. If you want some resiliency to Pod lifecycle and make it to be restarted automatically when there is an error, then you need to create a ReplicaSet.

Usually, a Replica Set is not created manually, but through a Deployment resource. A Deployment has always a Replica Set associated with it, so when a service is deployed using a Deployment resource, it explicitly has a Replica Set which monitors and restarts a Pod in case of an error as shown in [2.4](#).

Figure 2.4. Nature of a Deployment



To create a Deployment file, create a new file called `greeting-deployment.yaml` in the terminal window opened before.

This file is a Deployment file contains more elements than a Pod file.

You need to set the number of Pod replicas that are required at the start time. A typical value is one, but it could be any other number.

A deployment must also define the container image that belongs to the Pod and the listening port of the container.

The Deployment file should look like [2.2](#):

Listing 2.2. Creating a Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: greeting-demo-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: greeting-demo
  template:
    metadata:
      labels:
        app: greeting-demo
    spec:
      containers:
        - name: greeting-demo
          image: quay.io/lordofthejars/greetings-jvm:1.0.0
          imagePullPolicy: Always
        ports:
          - containerPort: 8080
```

Once we created the file, we can apply it to the cluster by using `kubectl apply -f greeting-deployment.yaml`

We need to wait until the Pod is allocated into a node and it is ready to be

accessed. To do that, use `kubectl wait` subcommand in the terminal:

```
kubectl wait --for=condition=Ready pod/greeting-demo
```

Validate that the Pod is started and allocated correctly by getting the Pod status:

```
kubectl get pods
```

And you'll get something similar like the following output:

NAME	READY	STATUS	RESTARTS	AGE
greeting-demo	1/1	Running	0	18s

But in this case, your workload will not be available until the deployment is ready. Run the following command to get the deployment status.

```
kubectl get deployment
```

The output of a Deployment is slightly different than a Pod. In this case, the `available` field is the important one as shows you the number of replicas that are up and running. Since in our example we set to one, then only one Pod is available through this Deployment.

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
greeting-demo-deployment	1/1	1	1	5m50s

Let's repeat the previous experiment and delete the Pod and see what's happening with its lifecycle now.

First of all, we need to find the Pod name created in the previous Deployment file by running the `kubectl get pods` command.

NAME	READY	STATUS	RES
greeting-demo-deployment-7884dd68c8-4nf6q	1/1	Running	0

Get the Pod name, in our case `greeting-demo-deployment-7884dd68c8-4nf6q` and delete it:

```
kubectl delete pod greeting-demo-deployment-7884dd68c8-4nf6q
```

Wait a few seconds until the Pod is terminated and get the Pod status:

```
kubectl get pods
```

Now the output is different from the previous section when we only created a Pod. Notice that now there is a new Pod running. We can see that the Pod is new by two fields, the name of the Pod is different than in the previous case, and the age is nearest our time.

NAME	READY	STATUS	RES
greeting-demo-deployment-7884dd68c8-qct8p	1/1	Running	0

The Pod has Deployment with a Replica Set associated; therefore it has been restarted automatically.

Now that we know how to deploy our workloads correctly, it is time to see how we can access them.

Service

So far, we've deployed an application to the Kubernetes cluster, however, each Pod that belongs to a deployment gets its IP address. Since Pods are by definition ephemeral, they are created and destroyed dynamically, and new IP addresses are assigned dynamically as well, reaching these pods by IP addresses, might not be the best choice as they might be invalid in the future.

A Kubernetes Service is the way to expose a set of Pods with a stable DNS name and IP address and it will load-balance across them.

Following the previous example with the `greeting-demo` deployment done, it is time to create a service so we can access it.

But before that, let us introduce what is a *Label* in Kubernetes.

A label is a key/value pairs associated with Kubernetes resources and, their main intend is to identify those objects from the user point of view. For example, we could annotate a Deployment with a custom label to identify it as a Deployment that belongs to the production environment.

If we look carefully at the previous Deployment, we'll see that there is a label defined with key `app` and value `greeting-demo` that is applied to all Pods created:

```
template:  
  metadata:  
    labels:  
      app: greeting-demo
```

The set of Pods targeted by a Service is usually determined by the labels registered in the Pods.

Let's expose the Pods created in the previous Deployment using a Service selecting the Pods with the `app: greeting-demo` label.

Create a new file called `greeting-service.yaml` in the working directory.

The Service definition should configure the port mapping between `containerPort` defined in Deployment (8080) and the exposed port we choose to be exposed by the Service. Moreover, we need to define the selector value setting the labels of `greeting-demo` Pods.

The Service file should look similar as shown in [2.3](#):

Listing 2.3. Creating a Service

```
apiVersion: v1  
kind: Service  
metadata:  
  name: the-service  
spec:  
  selector:  
    app: greeting-demo  
  ports:  
    - protocol: TCP  
      port: 80  
      targetPort: 8080  
  type: LoadBalancer
```

Once we created the file, we can apply it to the cluster by using `kubectl apply` ` subcommand:

```
kubectl apply -f greeting-service.yaml
```

Since we are using `minikube`, there will be no external IP to access to the service and `minikube` address must be used.

You can validate that there is no external IP associated with the created Service by running the following command in the terminal window:

```
kubectl get services
```

You should see that the `external IP` of the `the-service` Service remains in Pending status.

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP
the-service	LoadBalancer	10.102.78.44	<pending>	80:3095

In the terminal window, run the following command to set as environment variables the connection values to access the service:

```
IP=$(minikube ip)
PORT=$(kubectl get service/the-service -o jsonpath=".spec.ports[0].port")
```

And then we can query the service by using `curl` tool:

```
curl $IP:$PORT/hello
```

The greeting application returns a `Hello World` message as response to the request.



Important

In the case of running Kubernetes in a public cloud, an external IP will become a real IP in a few seconds.

You can get the external IP value by getting Service configuration:

```
kubectl get services
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
------	------	------------	-------------	---------

```
myapp    LoadBalancer    172.30.103.41    34.71.122.153    8080:31974
```

Clean Up

Before we can jump to the following concept, it is time to delete the application we've deployed in the previous section. To do that, we can use `kubectl delete` command:

```
kubectl delete -f greeting-deployment.yaml
```

Service isn't deleted deliberately as we have a use for it later on.

2.2.3 Volume

A Kubernetes volume is a directory containing some data which is accessible to the containers running inside pods. The physical storage of the volume is determined by the volume type used, for example, `hostPath` type uses the worker node filesystem to store the data or `nfs` uses the *NFS* (Network File System) to store the data.

Kubernetes volumes is a big topic as it is related to persistence topics, and this topic is out of the scope of this book. You are going to use volumes in this book, but only as a way for mounting ConfigMaps and Secrets, if you are not familiar with them, no worries, we'll cover them in the following sections.

2.3 Managing application configuration

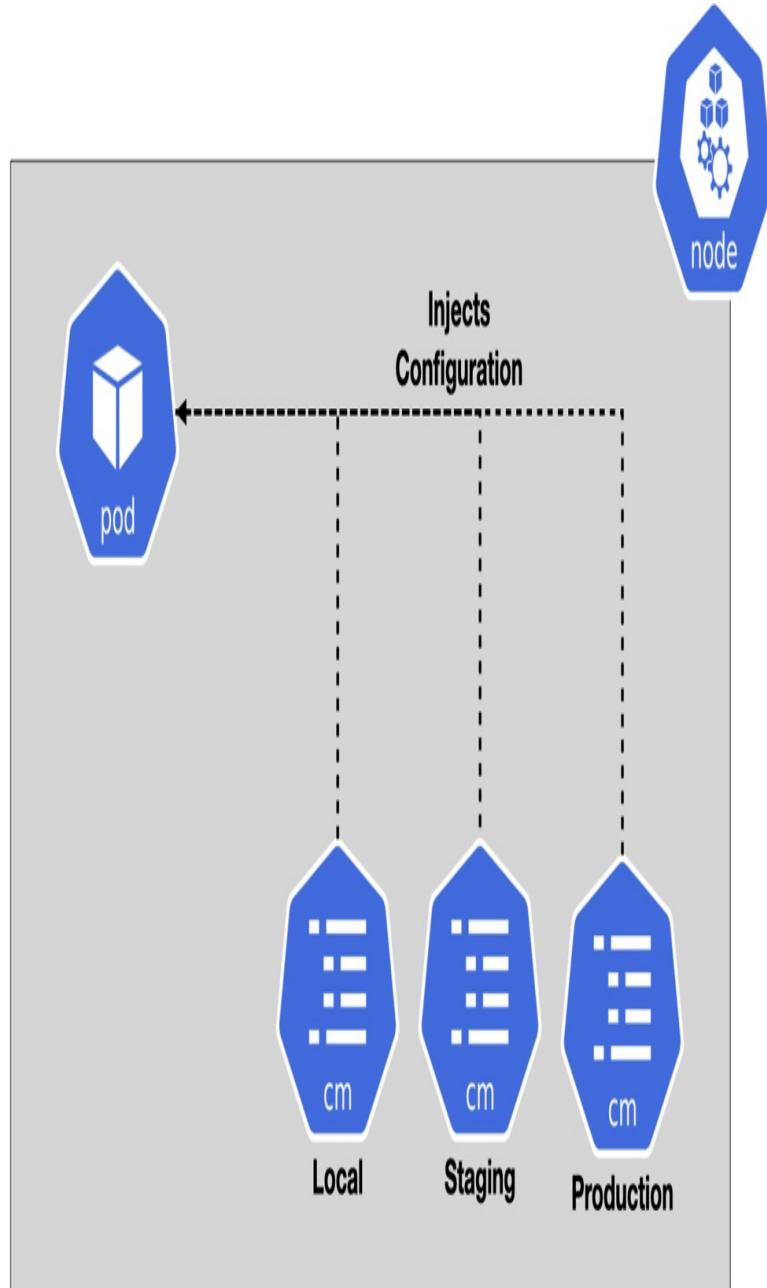
In the previous sections, we've seen how to deploy an application to a Kubernetes cluster with a welcome message *hardcoded* into the application. In this section, we'll set the welcome message externally using a configuration parameter.

2.3.1 ConfigMap

A ConfigMap is a Kubernetes object used to store **non-confidential data** in a map form. One of the big advantages of a ConfigMap is that lets you

externalize environment configuration data from the application code setting specific values depending on the cluster as seen at [2.5](#).

Figure 2.5. Injection of ConfigMap



The configuration map can be injected into a Pod to be consumed as a command-line argument, environment variable, or as a volume. In the book, we are going to cover the last two.

The application that has been deployed in the previous sections of this chapter, it returns a default welcome message (`Hello World`), but this welcome message is decoupled from the code so you can externally set it.

The following [2.4](#) shows the logic in the service that loads the welcome message to get back to the caller.

First of all, the code checks if there is a `GREETING_MESSAGE` environment variable set. If not set, then it tries to load a properties file located at `/etc/config/conf.properties` with the `greeting.message` key defined inside the file. Otherwise, the default message is returned to the caller.

Listing 2.4. Greeting Service

```
final String envGreeting = System.getenv("GREETING_MESSAGE");

if (envGreeting != null) {
    return envGreeting;
}

java.nio.file.Path confFile = Paths.get("/etc/config/conf.property");
if (Files.exists(confFile)) {

    final Properties confProperties = new Properties();
    confProperties.load(Files.newInputStream(confFile));

    if (confProperties.containsKey("greeting.message")) {
        return confProperties.getProperty("greeting.message");
    }
}

return "Hello World";
```

Environment Variables

The simplest way to get key/value from a config map is to inject them as environment variables into your pods. Then, we can get the environment variables in your applications using whatever methods the programming language provides you.

The important part of a ConfigMap resource is the data section. There is the field where you define the configuration items in form of key/values.

Let's create the ConfigMap resource named `greeting-config.yaml` with `greeting.message` as configuration key and `Hello Ada` as configuration value as shown in [2.5](#):

Listing 2.5. Creating a ConfigMap

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: greeting-config
data:
  greeting.message: "Hello Ada" #1
```

This ConfigMap creates a configuration item with a new welcome message.

You apply it as any other Kubernetes resource to the cluster by using `kubectl apply`:

```
kubectl apply -f greeting-config.yaml
```

The ConfigMap is created but it is just a configuration element, now we need to change the previous deployment file so it gets the configuration from the ConfigMap and inject it inside the container as an environment variable.

Create a new file called `greeting-deployment-configuration.yaml` in the working directory.

This Deployment file should be similar to the one we created previously, but containing an `env` section where we set the environment variable (`GREETING_MESSAGE`) to be created inside the Pod, and the value we want to assign which in this case it is taken from `greeting-config` Config Map.

The file should look as shown in [2.6](#):

Listing 2.6. Creating a Deployment with ConfigMap

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: greeting-demo-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: greeting-demo
  template:
    metadata:
      labels:
        app: greeting-demo
  spec:
    containers:
      - name: greeting-demo
        image: quay.io/lordofthejars/greetings-jvm:1.0.0
        imagePullPolicy: Always
        ports:
          - containerPort: 8080
        env:
          - name: GREETING_MESSAGE #1
            valueFrom:
              configMapKeyRef:
                name: greeting-config #2
                key: greeting.message #3
```

You apply it to the cluster by running `kubectl apply` command:

```
kubectl apply -f greeting-deployment-configuration.yaml
```

`GREETING_MESSAGE` environment variable is created inside the `greeting-demo` container with `Hello Ada` as value. Since our application is aware of this variable, the message that is sent back is the one configured in the `ConfigMap` under the `greeting.message` key.

Let's check it: With `IP` and `PORT` environment variables already set as explained in [the section called “Service”](#) section, we can query the service and see that the message has been updated to the configured one:

```
curl $IP:$PORT/hello
```

Hello Ada is returned as response as it is the message configured in the Config Map.

Now that we've seen how to configure our application using a ConfigMap and injecting the values as environment variables, let's move to inject this configuration value but as a file.

Volume

So far, we've seen that a ConfigMap can be injected as environment variables, and this is a perfect choice when you are moving legacy workloads to Kubernetes, but we can also mount a ConfigMap as a file using volumes.

Since the application can be configured using a properties file, we'll write a new properties file inside the container using a ConfigMap and volumes.

Let's create the ConfigMap resource named greeting-config-properties.yaml in the working directory.

To define the properties file, in the data section, set the filename required by our application (conf.properties) as key and the content of the properties file embedded as value.

The new Deployment file should look like [2.7](#):

Listing 2.7. Creating a ConfigMap

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: greeting-config
data:
  conf.properties: | #1 #2
    greeting.message=Hello Alexandra
```

You apply it to the cluster by running kubectl apply command:

```
kubectl apply -f greeting-config-properties.yaml
```

Now we need to materialize the `config.properties` file from the `ConfigMap` to the container. And for such a task, we need to define a Kubernetes volume in the container definition and store the content placed in the `ConfigMap` inside it.

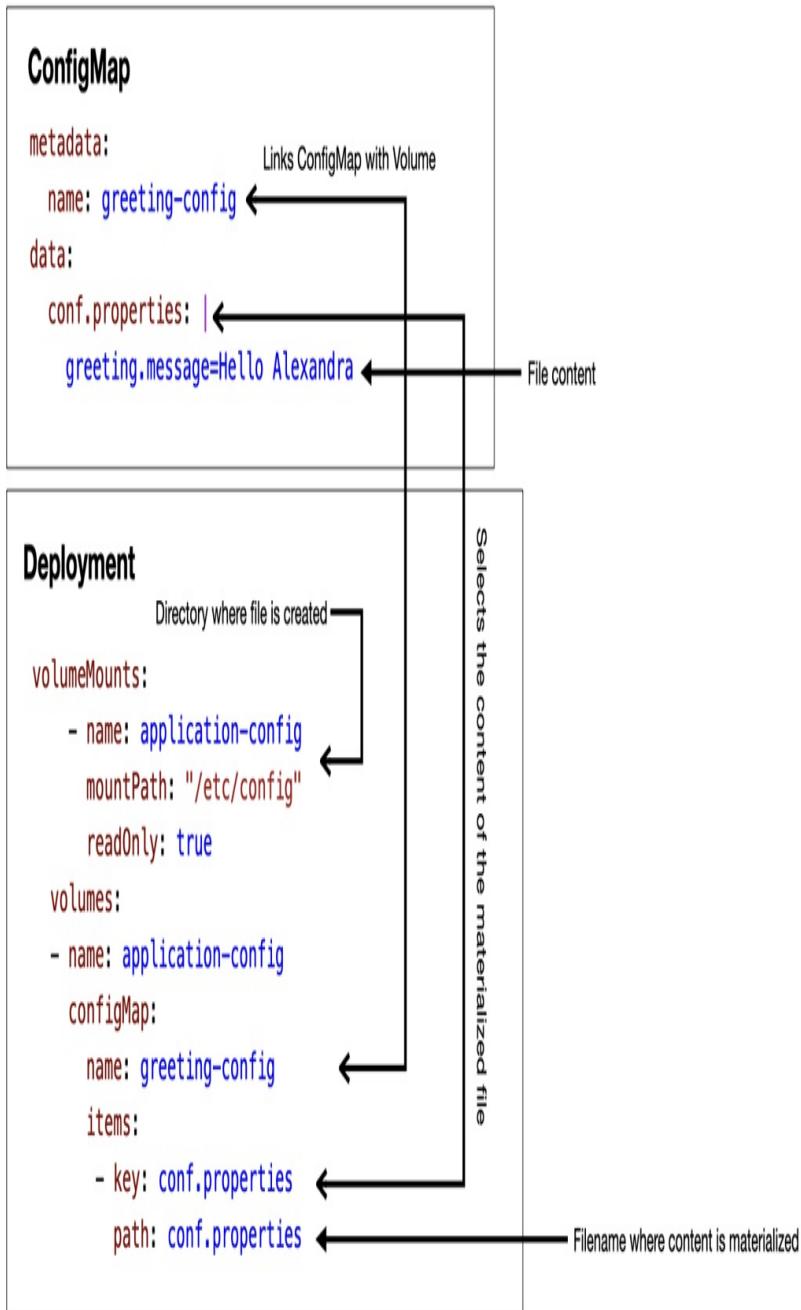
Create a new file named `greeting-deployment-properties.yaml` in the working directory. In the Deployment file, we need to define two big things the volume configuration and the Config Map where content is retrieved.

In `volumeMounts` section, set a name the name of the volume (`application-config`) and the directory where the volume is mounted, our application reads configuration properties file from `/etc/config`.

The second thing we need to do is linking the Config Map to the Volume, so the configuration file is created inside the defined volume with a specific name (`conf.properties`) and the content that was defined in the Config Map.

In the following [2.6](#) we can see the relationship between both elements:

Figure 2.6. Injection of ConfigMap



The Deployment file injecting configuration in a volume should look like [2.8](#):

Listing 2.8. Creating a Deployment with ConfigMap

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: greeting-demo-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: greeting-demo
  template:
    metadata:
      labels:
        app: greeting-demo
    spec:
      containers:
        - name: greeting-demo
          image: quay.io/lordofthejars/greetings-jvm:1.0.0
          imagePullPolicy: Always
          ports:
            - containerPort: 8080
          volumeMounts:
            - name: application-config #1
              mountPath: "/etc/config" #2
              readOnly: true
        volumes:
          - name: application-config #3
            configMap:
              name: greeting-config #4
              items:
                - key: conf.properties #5
                  path: conf.properties #6
```

You apply it to the cluster by running `kubectl apply` command:

```
kubectl apply -f greeting-deployment-properties.yaml
```

The container that is started contains a new file at `/etc/config/conf.properties` directory and the file content is the one embedded in the ConfigMap.

Let's check that the configured value in properties file is used by the service. With `IP` and `PORT` environment variables already set as explained in [the section called “Service”](#) section, we can query the service and see that the message has been updated to the configured one:

```
curl $IP:$PORT/hello
```

Hello Alexandra message is returned as it is the value configured in the Config Map.

We can use both approaches to inject all configuration values from a ConfigMap into a container, but what is the best approach? Let's explore some use cases and which might be the best approach.

Difference between Environment Variables and Volumes

It's now time to explore when to use environment variables and when volumes.

Environment Variables approach is usually used in the case of legacy applications that can be configured using environment variables and you *can't/don't want to* update the source code.

Volumes approach can be used in case of *green field applications* or applications that are configured using a file. The configuration process is easier with the volumes approach if the application requires to set multiple configuration properties as you configure all of them at once in the file. Moreover, the content of a volume is refreshed by Kubernetes if the ConfigMap gets updated. Of course, your application needs to handle this use case and provide a re-load configuration capability too. Keep in mind that this sync process is not immediately there is a delay between the change and the kubelet syncs the change as well as the `TTL` of ConfigMap cache.



Tip

If refreshing the configuration values is a key feature for your application, there is a more deterministic approach by using the [Reloader](#) project.

Reloader is a Kubernetes controller to watch changes in ConfigMaps and Secrets and do rolling upgrades on Pods with their associated Deployment, StatefulSet, DaemonSet and DeploymentConfig.

It has the advantage that it works in both environment variables and volumes approach and you do not need to update the service source code to handle this use case as the application is restarted during the rolling update. However, there is the drawback of having to install **Reloader** controller inside the cluster.

So far, we've seen the first of the two types of Kubernetes objects that can inject configuration data into a container.

But ConfigMaps content is no secret nor encrypted, as they are in plain text. This means that there is no confidentiality on the data. If we are trying to configure a database URL, a port, or database configuration like timeouts, retries, packet size, ... we can use ConfigMaps without much concerns regarding the security issues, but what's happening with parameters such as database username, database password, API key, ... where confidentiality is very important?

Then we need to use the second type of Kubernetes object that can inject configuration data into a container.

2.4 Kubernetes Secrets to store sensitive information

A Secret is a Kubernetes object used to store **sensitive/confidencial** data like passwords, API keys, SSH keys, ...

Secrets are similar to ConfigMaps as both approaches are used to inject configuration properties inside a container, the first one is secure, the latter not. Also, both are created similarly (with the difference of kind field to specify the type of object), and they are exposed inside a container in a similar way (as environment variables or mounting as a volume).

But obviously there are some differences that we are going to explore now:

2.4.1 Secrets are encoded in Base64

One of the big differences between Secrets and ConfigMaps is how data is stored inside *etcd*. Secrets store data in **Base 64** format meanwhile ConfigMaps store data in a plain text.

Let's go deep on **Base 64** format.

Base 64 is an encoding schema that represents binary and text data in an *ASCII* string format converting it into a radix-64 representation.

For example, a Alex text data is converted to QWxleA== in the Base64 format.

It is very important to have in mind that Base 64 is **not** an encryption method and, so any text encoded in Base 64 is a masked plain-text.

Let's create a Secret containing a secret greetings message.

The Secret object is similar to the ConfigMap object but contains two possible ways of setting data: `data` and `stringData`. The `data` is used when you want to encode configuration values to Base 64 manually. The `stringData` on the other hand lets you set configuration values as unencoded strings and they are automatically encoded.

Let's create a Secret named `greeting-secret-config.yaml` in the working directory as shown in [2.9](#) containing the secret message in `stringData` field:

Listing 2.9. Creating a Secret

```
apiVersion: v1
kind: Secret
metadata:
  name: greeting-secret
type: Opaque
stringData:
  greeting.message: Hello Anna #1
```

You apply it to the cluster by running `kubectl apply` command:

```
kubectl apply -f greeting-secret-config.yaml
```

Now that the secret has been created, we need to change the deployment file so it gets the message value from the Secret created in previous step and inject it inside the container. In the same way, we can inject configuration properties from ConfigMaps as environment variables or as volumes.

We'll write a deployment file that injects the secret as an environment variable in a similar way we did in the Config Map section, but in this case instead of using `configMapKeyRef` key we need to use `secretKeyRef`.

Create a new file called `greeting-deployment-secret-env.yaml` in the working directory as shown in [2.10](#):

Listing 2.10. Creating a Deployment with a Secret

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: greeting-demo-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: greeting-demo
  template:
    metadata:
      labels:
        app: greeting-demo
    spec:
      containers:
        - name: greeting-demo
          image: quay.io/lordofthejars/greetings-jvm:1.0.0
          imagePullPolicy: Always
          ports:
            - containerPort: 8080
          env:
            - name: GREETING_MESSAGE #1
              valueFrom:
                secretKeyRef: #2
                  name: greeting-secret #3
                  key: greeting.message #4
```

You apply it to the cluster by running `kubectl apply` command:

```
kubectl apply -f greeting-deployment-secret-env.yaml
```

Let's check that the secret value is used by the service when a request is sent. With `IP` and `PORT` environment variables already set as explained in [the section called “Service”](#) section, we can query the service and see that the message has been updated to the configured one:

```
curl $IP:$PORT/hello
```

`Hello Anna` is returned as it is the value of the secret we configured for this application.

Of course, don't do this in production, a secret is something you should never expose in the public API, but for this exercise, we thought it was a good way to show how secrets work.

We can also inject a secret as a volume, in a similar way as we did with `ConfigMaps`.

In this case, the `items` field is not specified, hence all the keys defined in the `Secret` resource are mounted automatically. Since the key of the mounted directory is `greeting.message`, a file named `greeting.message` is created at the configured volume with `Hello Alexandra` as data content.

```
volumeMounts:
  - name: greeting-sec
    mountPath: "/etc/config" #1
    readOnly: true
  imagePullPolicy: "IfNotPresent"
  name: "greeting"
  ports:
    - containerPort: 8080
      name: "http"
      protocol: "TCP"
volumes:
  - name: greeting-sec #2
    secret: #3
    secretName: greeting-secret #4
```

Now we know the basics of secrets, the difference between `ConfigMap` and `Secret`, and how to inject them into a container. But this is the basic stuff, there are still a lot of things we need to control until you we can say that our application is managing the secrets correctly.

So probably as a reader, you might be wondering, "Why a Secret is named *secret* if it is not really secret at all as it is not encrypted, it is just encoded in Base 64?". That is a fair question, but keep reading until the end of this chapter to fully understand the reasoning behind it.

2.4.2 Secrets are mounted in tmp filesystem

A secret is only sent to a node if there is a Pod that requires it. But what it is important is that a secret even though is mounted as a volume, is never written to disk but in-memory using `tmpfs` filesystem.

`tmpfs` stands for temporal filesystem and as its name suggests it is a filesystem but data is stored in volatile memory instead of persistent storage. When the Pod containing the secret is deleted, the kubelet is responsible to delete it as well from memory.

2.4.3 Secrets can be encrypted at rest*

Data at rest is the term that is known as the data persisted which is infrequently accessed. In this category falls the configuration properties (secrets are configuration properties too), as they are usually stored in files and they are usually accessed one time at boot-up time to be loaded into the application.

Encryption is the process of converting plain text data into ciphertext. After the text is ciphered, only authorized parties can decipher the data back to plain text.

Encryption data at rest then is the process of encrypting sensitive data at rest.

Among other things, all data from ConfigMaps and Secrets are stored inside etcd unencrypted by default. Notice that all these elements are *data at rest* and some should be protected. Kubernetes supports encryption at rest by ciphering Secret objects in etcd adding a new level of protection against attackers.

We know that this has been a really quick introduction to this topic, but we

are going to explore it in another chapter deeply as this is an important concept when we speak about secrets and Kubernetes.

2.4.4 Risks

We can think that now we are managing our secrets correctly, but it is not. Let's enumerate all the possible security breaches that a hypothetical attacker could exploit to steal our secrets.

Is it more secure to inject secrets as environment variables or as volumes?

Maybe you are wondering what is the best way to inject a secret, as an environment variable, or as a volume? Intuitively, you might think that as a volume is safer than as an environment variable because in case of an attacker gets access to the Pod, listing an environment variable is easier than searching through the whole filesystem trying to find the secret files. And it is a fair point, but let us show you that in terms of security when you get unwanted access to the Pod, both offer a similar level of security.

Let's suppose that an attacker gets access into a running Pod and the secrets are injected as environment variables, he could list all environment variables by running `export` command in a shell:

```
export

declare -x GREETING_MESSAGE="Hello Anna" #1
declare -x HOME="/"
declare -x HOSTNAME="greeting-demo-deployment-5664ffb8c6-2pstn"
...
```

So an attacker can figure out easily the values of your secrets.

So the other option is using volumes. Since any arbitrary directory can be mounted, we can think that we are safe because an attacker should need to know where the volume is mounted. Well yes, that's true, sadly there is a way to know that easily.

Now the attacker gets access to a running Pod with secrets mounted as a

volume, he could list all mounted filesystems by running the `mount` command in a shell:

```
mount | grep tmpfs

tmpfs on /dev type tmpfs (rw,nosuid,size=65536k,mode=755)
tmpfs on /sys/fs/cgroup type tmpfs (ro,nosuid,nodev,noexec,relati
tmpfs on /etc/sec type tmpfs (ro,relatime) /#1
...
ls /etc/sec #2
greeting.message
cat /etc/sec/greeting.message #3
Hello Anna
```

So there is no perfect solution, one of the advantages of using volumes instead of environment variables is that some applications might log the current environment variables at boot-up time, which at the same time can be sent to a central logging system so you are spreading all secrets across your infrastructure, making that any security breach in any of that part could expose them.

Is it totally lost? Of course not. First of all, you need to calibrate the chances of an attacker get access to your Pods/Nodes/Infrastructure. Second, there are some actions that you can apply to limit the access to a Pod, for example remove rights for executing `kubectl exec`.

Secrets are stored in etcd

As we read in Master Node section, `etcd` is a key-value database where all Kubernetes objects are stored, and of course, `ConfigMaps` and `Secrets` are not an exception.

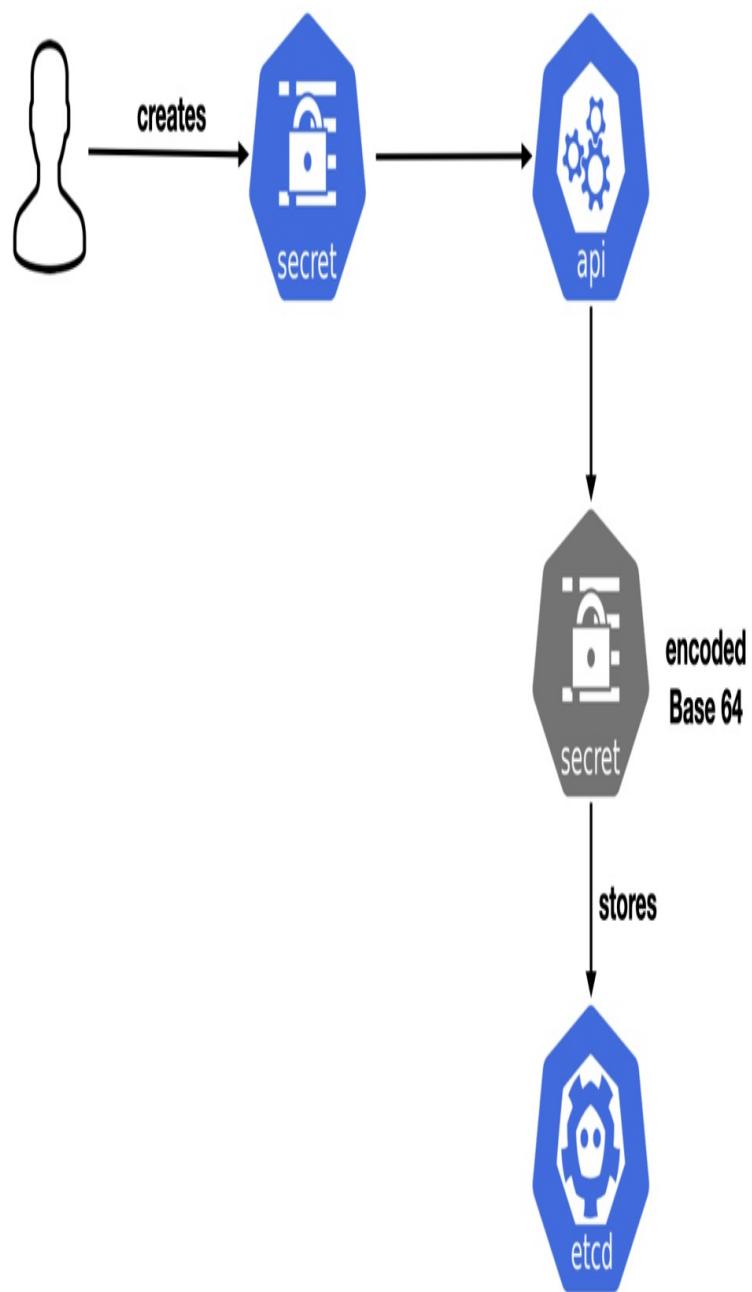
The process that happens when a secret is created is the following one:

1. A developer/operator creates a new secret resource and communicates with Kubernetes API service to apply it (`kubectl apply -f ...`).

2. The Kubernetes API service processes the resource and inserts the secret inside etcd under /registry/secrets/<namespace>/<secret-name> key.

The process is shown in the following illustration [2.7](#).

Figure 2.7. Process of storing a secret



etcd is after all a database, therefore we need to take into consideration some aspects:

- The access to etcd must be restricted to admin users, if not etcd could be queried by anyone to get the secrets.
- etcd database is persisted to disk. Since secrets are not encrypted by default, anyone with access to the disk can read the etcd content.
- Doing disk backups is a normal operation but again be aware of what's happening with backups as they contain sensitive information not encrypted by default.
- etcd is an external service that is accessed by the Kubernetes API server using the network. Make sure to use SSL/TLS for peer-to-peer communication with etcd (*data in transit*).

Base 64 is not encryption

Base 64 is not an encryption method but just an encoding method. It is important to remember that secrets are not encrypted by default and we need to enable *encryption data at rest* feature to store our secrets encrypted into etcd.

Acessing Pod

As we've seen before, if an attacker gets access to the Pod, then it is pretty easy to steal secrets even if we are using environment variables or volumes.

Related to this attack there is another one that we should be worried about. Anyone who can create a Pod can inject and read secrets.

Secrets at Source Code repositories

A secret can be created by using the kubectl CLI tool.

```
kubectl create secret generic greeting-secret \
--from-literal=greeting.message=Hello
```

But most of the time, we configure the secret through a file (either *JSON* or *YAML*) which might have the secret data encoded in Base 64 and not ciphered. And there are some risks associated with this approach as secrets can be compromised in the next situations:

- Sharing the file in an insecure way (ie email).
- Commit the file into the source repository.
- Losing the file.
- File backed up without any security.

Root permissions

At the time of writing this book, anyone with root permission on any node can read any secret from the API server, by impersonating the kubelet.

We've finished implementing a basic strategy for dealing with secrets and Kubernetes, but our work with secrets isn't complete, because we've yet to tackle most of the risks that we've identified in this section. Let's start from the ground, our next chapter shows how to manage secrets from the beginning and this is how to create and manage the Kubernetes resource file correctly and following the best security principles.

Stick around because now things are getting interesting.

2.5 Summary

- A Kubernetes cluster is composed of master and optionally worker nodes.
- Any Kubernetes resource and current status of the cluster is stored at the etcd instance.
- Config Maps are injected in a Pod as an environment variable or as a file.
- Secrets are not so different from Config Maps in terms of construction and usage.
- Secrets are **encoded** in Base 64, hence NOT secure by default.

3 Securely Storing Secrets

This chapter covers

- Capturing Kubernetes manifests to store in Version Control Systems
- Enabling secure secret storage at rest
- Using Kubernetes Operators to manage Kubernetes resources including secrets
- Incorporating security considerations to Kubernetes package managers
- Implementing Key Rotation to increase security posture

Chapter 2 provided an overview of the key architectural components of a Kubernetes environment as well as the way workloads are deployed and the methods for injecting configurations through the use of ConfigMaps and Secrets.

Once resources have been added to a Kubernetes cluster, how are they managed? What happens if they were inadvertently removed? It becomes increasingly important that they be captured and stored for potential later use. However, when working with resources that may contain sensitive information, careful thought and considerations must be taken into account.

This chapter introduces tools and approaches that can be used to store Kubernetes secrets securely at rest and illustrates the benefits of declaratively defining Kubernetes resources.

3.1 Storing Kubernetes Manifests at Rest

One of the benefits of cloud native technologies is that resources can be built, deployed and configured on-demand. With only a few clicks of a mouse or keyboard, entire architectures can be constructed with minimal effort. For those getting started with Kubernetes, excitement blossoms as one realizes just how easy it is to build complex applications. They might even show their parent, friend or co-worker. But, as one demonstrates their work, they may be

asked how it can be replicated. It is at that point that it becomes ever important that each Kubernetes resource, including those that contain sensitive values, be properly managed and stored for later use.

In Chapter 2, we covered the two primary methods for creating resources in a Kubernetes environment.

1. Using the Kubernetes CLI (`kubectl`) translate inputs provided via command line arguments
2. Explicitly state the configuration of resources using a YAML or JSON formatted file

The former, where the Kubernetes CLI provides the translation for us, is known as the Imperative method. For example, when we used the `kubectl create secret` subcommand to create the secret for our deployment in the prior chapter, the Kubernetes CLI determined how to interpret the input that we provided, and send a request to the Kubernetes API to create the secret.

While this approach does simplify the initial setup and configuration of resources, it does pose challenges into their long term supportability. The `kubectl create secret` command is not idempotent, and rerunning a second time will result in an error. This can be seen by executing the following which will attempt to create a secret called `greeting-secret` and result in an error as a secret with the same name in the `default` namespace already exists as it was created in Chapter 2

```
kubectl create secret generic greeting-secret -n default \
--from-literal=greeting.message=Hello

Error from server (AlreadyExists): secrets "greeting-secret" alre
```



Note

If no secret was already present in the `default` namespace, running the command above a second time will result in a similar error.

Now, instead of using the imperative approach, resources can be represented explicitly in either YAML or JSON format and applied to the cluster using

the `kubectl` tool. This approach is known as declarative configuration and has benefits to support the longterm lifecycle of resources within a Kubernetes environment.

The use of declarative configuration is one of the key traits of a concept that has gained popularity over the course of the last few years, Infrastructure as Code (IaC). Instead of manually configuring resources or using random scripts, the configurations applied to infrastructure or applications are explicitly defined which result in the following benefits:

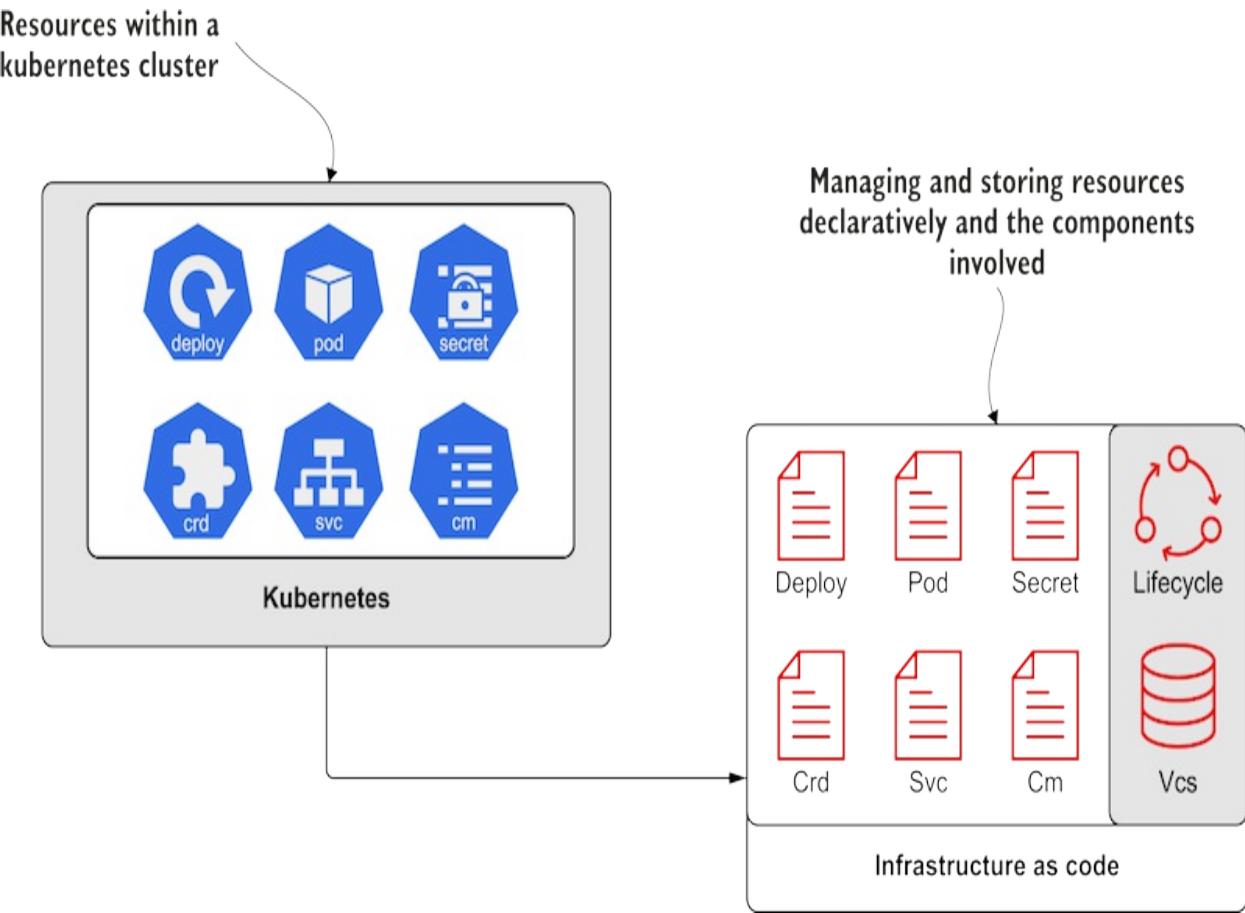
- Reduction of errors
- Repeatability
- Auditing and Tracking

The ability to audit and track these configurations are made possible especially when storing manifests in a Version Control System (VCS), such as Git. Once resources have been captured, their lifespan including what and who changed them can be appropriately tracked. So, if a disaster occurred, instead of having to determine what the state of the Kubernetes cluster was at that point, the manifests that been previously captured and stored can be reapplied, resulting in minimal downtime and effort.

3.1.1 Capturing resources for declarative configuration

When adopting a declarative based configuration or transitioning from a primarily imperative based approach, there are multiple strategies for capturing the manifests. Recall that the end result will be a series of YAML or JSON files. While both file types can be applied to a Kubernetes cluster using the `kubectl` command line tool, YAML based files are preferred to due their readability.

Figure 3.1. The capturing of Kubernetes resources as Infrastructure as Code (IaC)



Two common scenarios can be used to capture manifests for storage in a declarative fashion:

1. Capture the output from an invocation of `kubectl imperative commands`
2. Capture the output of resources already present within a Kubernetes environment.

Capturing `kubectl imperative commands`

To aid in the storage of manifests in a declarative fashion, The Kubernetes CLI provides two helpful flags that can be added when invoking imperative commands, such as `kubectl create secret`.

- `--dry-run` - Simulate how resources would be applied to the Kubernetes environment. Kubernetes versions older than 1.18 did not require the use of a parameter, such as `client`, as APIServer dry-run was refactored in

newer versions.

- -o - Output the result of the command in a number of formats, including YAML

Adding these flags to the imperative secret creation command as described above will output the representation of what would be sent to the Kubernetes cluster without any changes being made to actual the state of the cluster.

```
kubectl create secret generic greeting-secret -n default \
--from-literal=greeting.message=Hello --dry-run=client -o yaml
```

Capturing deployed resources

There is still tremendous value with the imperative capabilities of the Kubernetes CLI. The combination of imperative invocations and manual configurations are common in the development process as the details of each resource is tested and validated. Once satisfied with the configuration in place, it is recommended that these assets are then captured so that they can be stored in a Version Control System to align with the practices of Infrastructure as Code (IaC) principles.

The current state of resources can be queried using the `kubectl get` subcommand. In a similar fashion as shown in [Capturing kubectl imperative commands](#), the output of the command can be output in a variety of formats.

Execute the following command to display the contents of the `greeting-secret` that was created earlier in Chapter 2:

Listing 3.1. greeting-secret Secret

```
kubectl get secrets greeting-secret -o yaml
```

```
apiVersion: v1
data:
  greeting.message: SGVsbG8= #1
kind: Secret
metadata:
  creationTimestamp: "2020-12-25T00:22:44Z"
  managedFields:
    - apiVersion: v1
```

```
fieldsType: FieldsV1
fieldsV1:
  f:data:
    .: {}
    f:greeting.message: {}
  f:type: {}
manager: kubectl-create
operation: Update
time: "2020-12-25T00:22:44Z"
name: greeting-secret
namespace: default
resourceVersion: "27935"
selfLink: /api/v1/namespaces/k8s-secrets/secrets/greeting-secre
uid: b6e87686-f4e6-454d-b391-aef37a99076e
type: Opaque
```

As you may notice in the output, there are fields including status and several fields within metadata (uid, resourceVersion, and creationTimestamp just to name a few) with runtime details from current cluster. These properties are not suitable for storage and should be removed. They can be removed manually or using a tool, such as `yq`, a lightweight YAML processor. An example of removing runtime properties using `yq` can be seen below

Listing 3.2. Outputting the content of the secret to a file

```
kubectl get secrets greeting-secret -o yaml | \
yq e 'del(.metadata.namespace)' - | \
yq e 'del(.metadata.selfLink)' - | \
yq e 'del(.metadata.uid)' - | \
yq e 'del(.metadata.resourceVersion)' - | \
yq e 'del(.metadata.generation)' - | \
yq e 'del(.metadata.creationTimestamp)' - | \
yq e 'del(.deletionTimestamp)' - | \
yq e 'del(.metadata.deletionGracePeriodSeconds)' - | \
yq e 'del(.metadata.ownerReferences)' - | \
yq e 'del(.metadata.finalizers)' - | \
yq e 'del(.metadata.clusterName)' - | \
yq e 'del(.metadata.managedFields)' - | \
yq e 'del(.status)' -
```

An overview of how to install the `yq` tool on your machine can be found in Appendix B.

The output from the prior command can be redirected to a file to enable the

storage for versioning within a Version Control System as seen below:

```
kubectl get secrets greeting-secret -o yaml | \
yq e 'del(.metadata.namespace)' - | \
yq e 'del(.metadata.selfLink)' - | \
yq e 'del(.metadata.uid)' - | \
yq e 'del(.metadata.resourceVersion)' - | \
yq e 'del(.metadata.generation)' - | \
yq e 'del(.metadata.creationTimestamp)' - | \
yq e 'del(.deletionTimestamp)' - | \
yq e 'del(.metadata.deletionGracePeriodSeconds)' - | \
yq e 'del(.metadata.ownerReferences)' - | \
yq e 'del(.metadata.finalizers)' - | \
yq e 'del(.metadata.clusterName)' - | \
yq e 'del(.metadata.managedFields)' - | \
yq e 'del(.status)' - \
> greeting-secret.yaml
```

With resources now being described in a declarative manner, and eligible to be tracked and visible in Version Control Systems, it becomes even more important that protections be made to ensure the values cannot be easily determined. Since Secrets are merely Base64 encoded, it is crucial that additional mechanisms be employed to obstruct the ability to ascertain their values.

The remainder of this chapter will introduce and demonstrate tools that can be used to secure Secrets at rest.

3.2 Tools for Securely Storing Kubernetes Resources

While the Kubernetes CLI does not offer any additional native capabilities to secure manifest for storage, the popularity of Kubernetes has afforded integration with other cloud native tools to help solve this challenge. These tools include those that already have the functionality to secure manifests at rest as well as solutions that have been specifically developed for this purpose in a Kubernetes context. As a consumer of Kubernetes, you may be tasked with managing resources that are either focused on the underlying infrastructure or the applications that are deployed within the platform; or maybe both. The tool that you will ultimately look to use depends on the use

case. Understanding which tools are available and how they can be used will help you make an informed decision to select the appropriate tool for your specific task.

3.2.1 Ansible Vault

A typical deployment of Kubernetes (not including Minikube) will need to have considerations made for both infrastructure and application components. These include the physical and virtual resources to support the control plane and worker nodes as well as the configuration of Kubernetes manifests. Configuration management tools are well positioned for this space as they not only manage the sometimes complex configurations associated with Kubernetes environments, but they help illustrate and implement Infrastructure as Code concepts.

Ansible is a popular configuration management tool and can be used to manage various aspects of the Kubernetes ecosystem. One of the key benefits of Ansible versus other comparable tools is that it is well suited for cloud environments as it is agentless (does not require a central management server) and communicates via SSH (Secure Shell), a common communication protocol. All that you need is the tool on your local machine and you can get right to work!

Installing Ansible

Ansible is a Python based tool, and as such, is the only prerequisite and can be installed on a variety of Operating Systems. Given that the instructions do vary depending on the target Operating System, refer to the official documentation

(https://docs.ansible.com/ansible/latest/installation_guide/intro_installation.html) on how to install Ansible for your machine.

Ansible 101

Ansible organizes automation directives in a series of YAML files. Directives describing the configurations to be applied to targets are organized into

Playbooks which declare the hosts that configurations should be applied to along with a series of tasks that define the desired state of each target machine. For example, a simple playbook could enforce that all Linux machines have a "Message of the Day" (motd) that is presented to all users when they login.

An example playbook is shown below:

Listing 3.3. Example Ansible Playbook

```
- hosts: linux #1
  tasks:
    - name: Set motd
      copy:
        content: "Linux Hosting Environment." #3
        dest: /etc/motd #4
```

Target instances are organized into *groups* and declared within *Inventory* files and define how Ansible facilitates the connection along with any *Variables* that are used during playbook invocation.

Playbooks are then invoked using the `ansible-playbook` command which will perform the execution of the automation.

```
ansible-playbook <playbook_file>
```

Ansible and Kubernetes

Ansible's "bread and butter" is the management and configuration of infrastructure. As the popularity of Kubernetes continues to grow, it is becoming a key component in the infrastructure of many organizations, and as such, integrations between Ansible and Kubernetes are available.

One of the available capabilities through this integration is the management of Kubernetes resource, and achieved using the `k8s` module. Modules are reusable scripts that can be included in playbooks. In the MOTD example above, the `copy` module was used to copy content from the local machine to the remote target.

Let's create an Ansible playbook to manage the configuration of our Kubernetes cluster by using the greeting-secret that we already have available on the local machine. Before we begin, let's prepare our working environment. First, make a copy of the greeting-secret.yaml file on our machine and create a new file called greeting-secret_ansible.yaml.

```
cp greeting-secret.yaml greeting-secret_ansible.yaml
```

Next, create a namespace called kubernetes-secrets-ansible for us to use for this scenario.

```
kubectl create namespace kubernetes-secrets-ansible
```

Next, change the namespace preference for our *kubectl* client to target the newly created namespace.

```
kubectl config set-context --current --namespace=kubernetes-secre
```



Note

When setting the namespace preference, all subsequent commands will query against the targeted namespace

Now, within the same directory where this file greeting-secret_ansible.yaml is located, create a new file called k8s-secret-mgmt.yaml with the following content to contain the playbook:

Listing 3.4. k8s-secret-mgmt.yaml

```
- hosts: localhost #1
gather_facts: no #2
connection: local #3
tasks:
  - name: Create Secret on Cluster
    k8s: #4
      definition: > #5
        "{{lookup('file', playbook_dir +
          '/greeting-secret_ansible.yaml') }}"
      state: present
      namespace: kubernetes-secrets-ansible #6
```

```
no_log: True #7
```

By default, the `k8s` module uses the `kubeconfig` file from the local machine to determine the method for communicating with the Kubernetes cluster. Since we were already authenticated to the `minikube` instance, this is taken care of for us.

Before executing the playbook, the `openshift` python module must be installed in order for Ansible to be able to communicate with the Kubernetes cluster. OpenShift is a distribution of Kubernetes and the `k8s` Ansible module requires the `openshift` module as a requirement prior to execution. This can be accomplished using pip, which depending on your operating system, may have been how Ansible itself was installed. If pip is not currently installed, Instructions on how to do so can be found in Appendix C.

Add the `openshift` Python module by executing the following command.

```
pip install openshift
```

With the necessary dependencies installed, run the playbook:

```
ansible-playbook k8s-secret-mgmt.yaml
```

```
[WARNING]: No inventory was parsed, only implicit localhost is av
[WARNING]: provided hosts list is empty, only localhost is availa
PLAY [localhost] ****
*****
***** TASK [Create Secret on Cluster] ****
*****
***** changed: [localhost]
PLAY RECAP ****
*****
***** localhost : ok=1    changed=1    unreachable=0
      skipped=0    rescued=0    ignored=0
```



Note

You can safely ignore the warnings as they do not affect the execution of the playbook

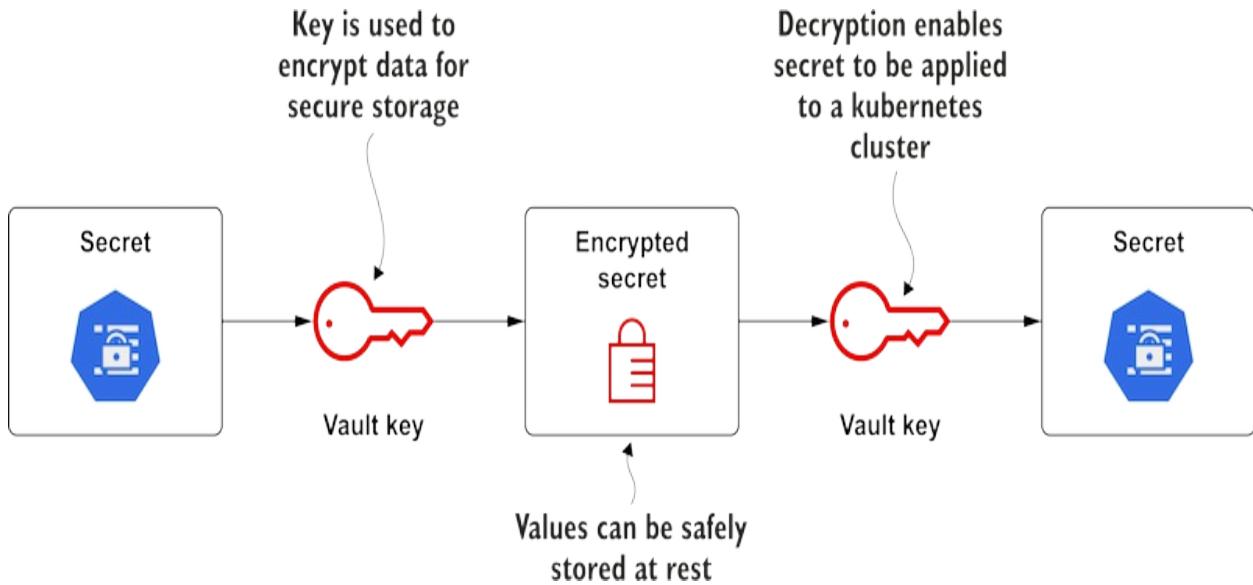
As emphasized in the output, the playbook ran successfully with the secret defined locally is now present on the Kubernetes cluster in the *kubernetes-secrets-ansible* namespace. This can be confirmed by running `kubectl get secrets greeting-secret`

Ansible Vault

As you might imagine, configuration management tools, such as Ansible, face the same challenges with managing sensitive values. When creating the Ansible playbook in the prior section, the Secret being sourced from the `greeting-secret_ansible.yaml` file contains the Base64 encoded value and the values are ultimately susceptible to being easily decoded. Fortunately, Ansible provides the capabilities aid in this situation by enabling the encryption of files that can be decoded at runtime through the use of Ansible Vault.

Ansible vault allows for variables and files to be protected so that they can be safely stored. Unlike Secrets in Kubernetes, Ansible vault uses encryption and not encoding to avoid being easily reverse engineered as in the case with Secrets.

Figure 3.2. The encryption and decryption process using Ansible Vault



As of Ansible 2.10, Ansible Vault only supports AES256 as the cipher algorithm that is used to encrypt the sensitive material.

To encrypt the content of the `greeting-secret_ansible.yaml` containing the secret with sensitive values, use the `ansible-vault encrypt` command. You will be prompted to provide a password that can be used to encrypt and decrypt contents of the encrypted password:

```
ansible-vault encrypt greeting-secret_ansible.yaml
```

```
New Vault password:  
Confirm New Vault password:  
Encryption successful
```

Once the file has been encrypted by Ansible Vault, the resulting file takes the following form:

Listing 3.5. `greeting-secret_ansible.yaml`

```
$ANSIBLE_VAULT;1.1;AES256  
<ENCRYPTED_CONTENT>
```

The result is a UTF-8 encoded file that contains a new line terminated header followed by the encrypted contents.

The header contains up to four (4) elements:

1. The format ID (only currently supports \$ANSIBLE_VAULT)
 2. The vault format version
 3. The cypher algorithm
 4. Vault ID label (Not used in this example)

The payload of the file is a concatenation of the ciphertext and a SHA256 digest as a result of the `hexlify()` method of the Python `binascii` module. The specific details will not be described here, but is explained thoroughly within the Ansible Vault documentation (https://docs.ansible.com/ansible/latest/user_guide/vault.html).

Once a file has been encrypted using Ansible Vault, the `--ask-vault-password` or `--vault-password-file` must be provided when calling the `ansible-playbook` command. To make use of the `--vault-password-file` flag, the password must be provided as the content within the referenced file.



Tip

Instead of providing a flag to the `ansible-playbook` command, the location of the Vault password file can be provided by the `ANSIBLE_VAULT_PASSWORD_FILE` environment variable.

Run the playbook and add the `--ask-vault-pass` flag which will prompt for the Vault password to be provided. When prompted, enter the password and press *Enter*. If the appropriate password was provided, the playbook will execute successfully.

```
PLAY RECAP ****
*****
localhost : ok=1    changed=0    unreachable=0
              skipped=0   rescued=0   ignored=0
```

If an incorrect value was provided, the following message will appear similar to the following:

```
TASK [Create Secret on Cluster] ****
*****
fatal: [localhost]: FAILED! => {"censored": "the output has been
```

Since we specified `no_log` in the task, a more descriptive error will not be provided. To investigate further, you may temporarily comment out `no_log` to ascertain the ultimate cause of the failure.

By using Ansible vault to encrypt the contents of the Kubernetes Secret in the `greeting-secret_ansible.yaml` file, the playbook and encrypted file can be safely stored in a Version Control System.

Ansible vault illustrates how one can manage encrypting and decrypting Kubernetes resources from a client side perspective. The next section will introduce how to transition the responsibilities to components running within the Kubernetes cluster instead.

3.3 Kubernetes Operators

While Ansible Vault satisfied the need to securely store sensitive Kubernetes assets, there are several areas that could be improved upon:

1. Those executing the Ansible automation are given the password to decrypt sensitive assets
2. Decryption occurs on the client side. Any sensitive assets are transmitted either in their cleartext value or associated with a Kubernetes Secret, and thus are Base64 encoded.

An alternate strategy is to leverage a model where the decryption process

occurs completely within the Kubernetes cluster, abstracting the end user or automation process from managing sensitive material as resources are applied.

This is the approach that is implemented by the Sealed Secrets project, where decryption occurs by a controller running within the cluster.

Recall back to chapter 2 that a key component of master nodes is that they contain controllers, aptly named as they implement a non-terminating, control loop, to manage and monitor the desired state of at least one resource within the cluster. When changes to the targeted resource occurs, the controller will enforce the desired state of the cluster matches the desired state.

One of the most common controllers that most end users would be familiar is a ReplicaSet Controller. Deployments are a common method for registering workloads into Kubernetes and a ReplicaSet is generated automatically whenever a Deployment is created. The ReplicaSet controller will monitor pods associated with the ReplicaSet and enforce the number of active pods matches the desired state as defined within the ReplicaSet.

3.3.1 Custom Resource Definitions

Historically, Kubernetes had a fairly small number of resources, such as Pods and Secrets. As popularity for the platform grew, so did the desire for the creation of new resource types, both from core maintainers as well as from users. As any developer can attest to, changes to core API's are typically a challenging and drawn out process.

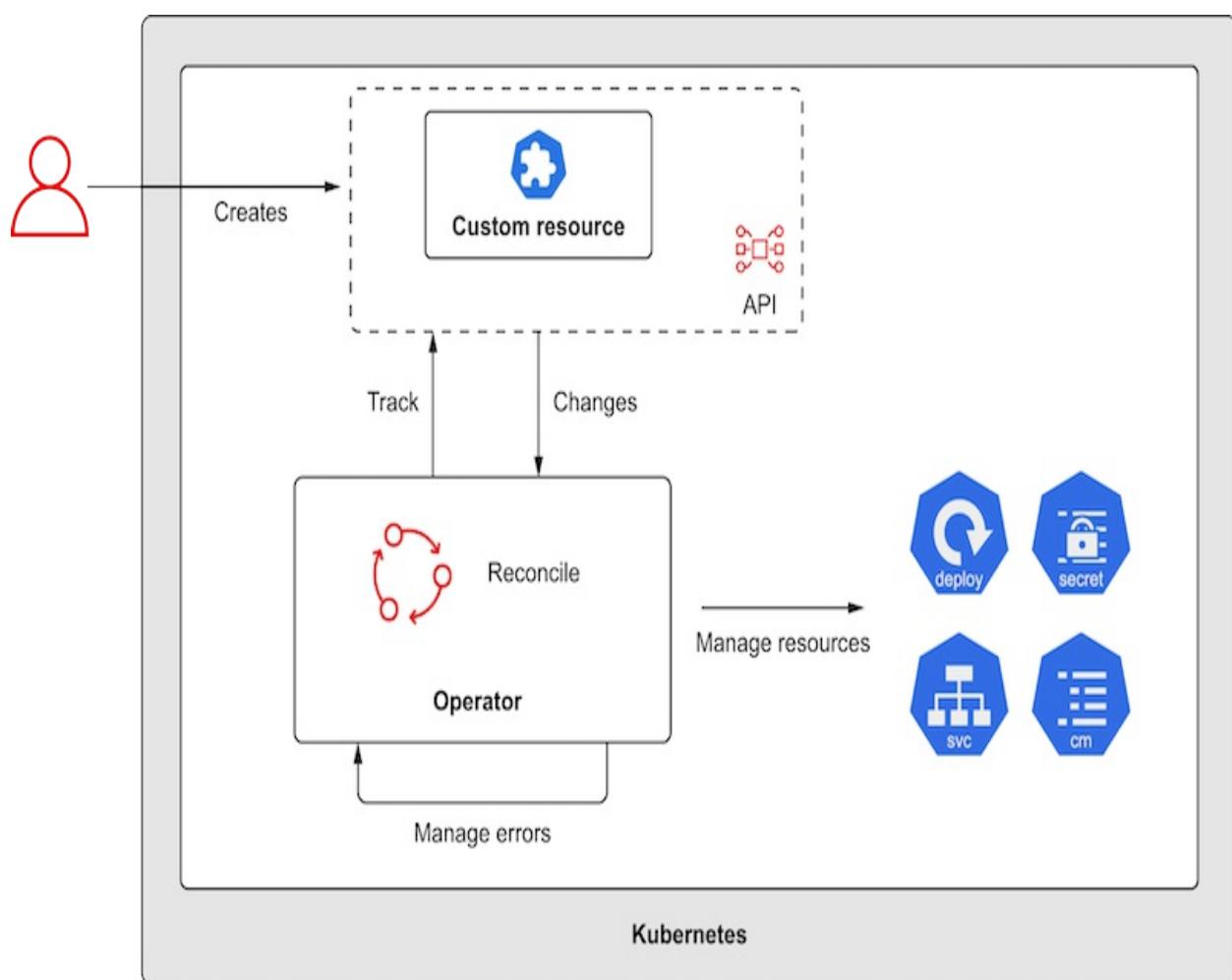
Custom Resource Definitions (CRD's), a new resource type, were a solution to this issue as it provided developers the opportunity to register their own API's and properties associated with these resources while being able to take advantage of the functionality within the API server without interfering with the core set of API's.

For example, a new resource called CronTabs could be defined with the goal of executing tasks at a particular scheduled point in time. An application could be developed to query the API that Kubernetes has allocated for CronTabs resources and execute any of the desired business logic. However,

instead of routinely querying the API, what if we were able to perform many of the same capabilities of the included set of controllers, such as immediately being able to react to state changes as in the creation of modification of a resource. Fortunately, client libraries, and in particular, "client-go" for the GoLang programming language provide these capabilities.

This concept of developing an application to monitor a custom resource and take action against it is known as an Operator and this pattern has been widely adopted within the Kubernetes community and is implemented by the Sealed Secrets project. The process for developing operators and custom controllers was once a large feat as developers needed to have intimate knowledge of Kubernetes internals. Fortunately, tools, such as kubebuilder and the Operator Framework have simplified this process.

Figure 3.3. An overview of how Operators manage resources in Kubernetes



3.3.2 Sealed Secrets

Given that the majority of the Sealed Secrets solution are offloaded to the controller/operator, what actions does it perform? In retrospect, Sealed Secrets contains three distinct components:

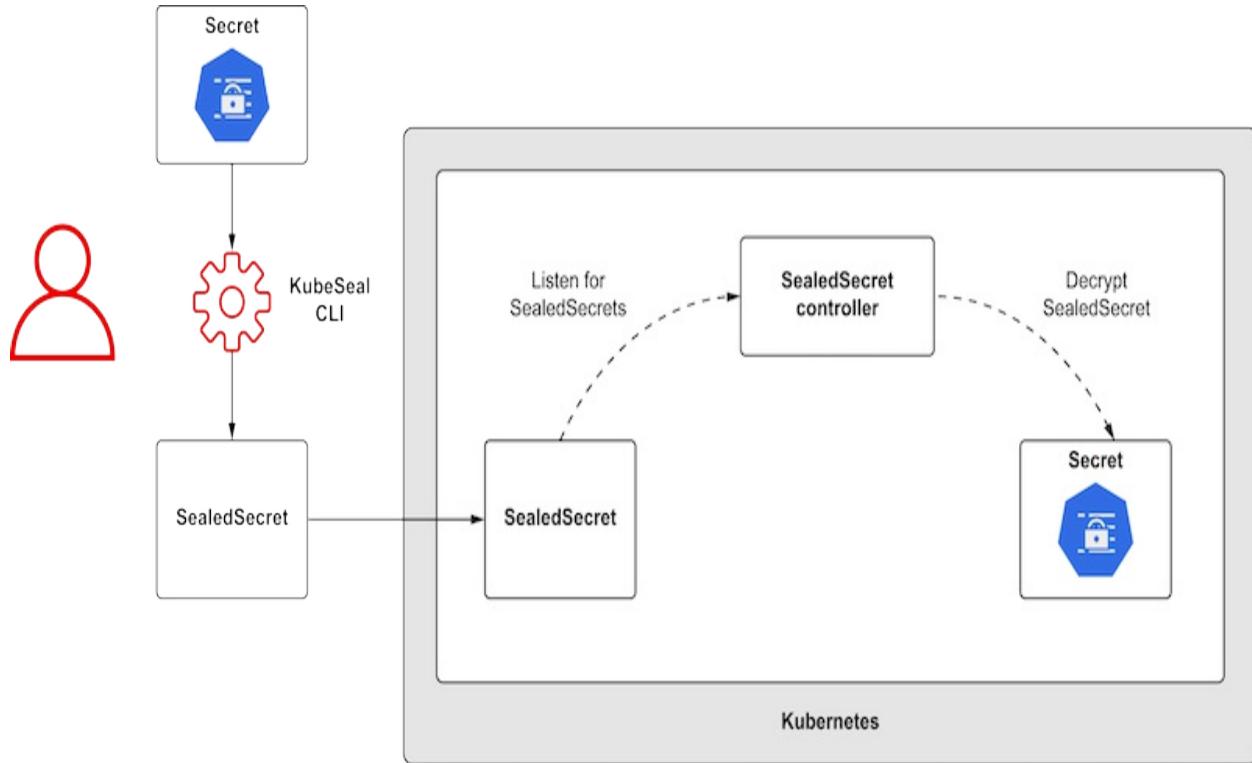
- The Operator/Controller
- A CLI tool called *kubeseal* which is used by the end user to encrypt Kubernetes secrets
- A Custom Resource Definition (CRD) called `SealedSecret`

After the CRD is added to the cluster and the controller is deployed to a namespace, the controller will create a new 4096-bit RSA public/private key pair if one does not exist which is saved as a secret within the same namespace the controller is deployed within.

End users use the *kubeseal* tool to convert a standard Kubernetes Secret into a *SealedSecret* resource. The encryption process takes each value within the Kubernetes Secret and performs the following actions:

1. Value is symmetrically encrypted with a randomly-generated one time use 32 bit session key.
2. The session key is then asymmetrically encrypted with the public key created by the controllers' previously generated public certificate using Optimal Asymmetric Encryption Padding (RSA-OAEP)
3. Result is stored within the *SealedSecret* resource.

Figure 3.4. The processes and components involved to encrypt and decrypt a secret using the Sealed Secrets project



Note

A more detailed overview of the encryption and decryption in use can be found within the Sealed Secret project homepage: <https://github.com/bitnami-labs/sealed-secrets>

Installing Sealed Secrets

The first step to installing Sealed Secrets is to deploy the controller to the cluster. While multiple methods are supported, installation via the raw Kubernetes manifests will be used.

For the purpose of this book, version v0.13.1 will be used and steps related to the installation of the controller and associated `kubeseal` command line tool can be found on the release page within the project repository (<https://github.com/bitnami-labs/sealed-secrets/releases/tag/v0.13.1>)

Similar to the process that was implemented in the prior section covering Ansible Vault, create a new namespace called `kubernetes-secrets-`

sealedsecrets and set the namespace preference to target this namespace.

```
kubectl create namespace kubernetes-secrets-sealedsecrets  
kubectl config set-context --current \  
--namespace=kubernetes-secrets-sealedsecrets
```

Add the controller to the Kubernetes cluster by executing the following command:

```
kubectl apply -f https://github.com/bitnami-labs/sealed-secrets/r
```

By default, the controller is installed into the `kube-system` namespace. Confirm the controller has been deployed successfully by listing the running pods within the `kube-system` namespace.

```
kubectl get pods -n kube-system -l=name=sealed-secrets-controller  
NAME                               READY   STATUS    RESTARTS  
sealed-secrets-controller-5b54cbfb5f-4gz9j  1/1     Running   0
```

Next, install the *kubeseal* command line tool. The binary releases along with steps on how it can be installed can be found on the release page previously referenced earlier in this section. Be sure to follow the steps detailed for your operating system.

Once *kubeseal* has been installed, confirm that the CLI has been installed properly and can obtain the public certificate that was generated by the controller, enabling us to encrypt secrets:

```
kubeseal --fetch-cert #1
```

```
-----BEGIN CERTIFICATE-----  
MIIErjCCApagAwIBAgIRAI0gwJnDRCIcZon5GumMT8UwDQYJKoZIhvcNAQELBQA  
wADAEFw0yMDEyMjgxNjI3MDhaFw0zMDEyMjYxNjI3MDhaMAAwggIIwGAQCSqGSIB3  
DQEBAQUAA4ICDwAwggIKAoICAQDOSr3qLBJ4YQRiKvQgkQgMN+sCp2mQo8vJbj8z  
r0aINXdkD6isHqq80uJ0uJ6ZigFpDmoyOUV1Hbkpr1ngu6d41fBpEW0caREZrcd9  
2s8yT2/8yJQ2Q1pZawG10XjHOFMNETdk3bveplWGWCY7QUKJJwHpW5vGVs9xLU34  
nnbPK0/dY106bnhIfVRgYvom0+IIIfSDx3t70Gg/hEm2jp7rkNBIdW0qnH7GwTNVx  
6FdD+DGztSgqTMdtla7IwRZjfXSf3HAIK0ZY8cq7hsd3+JewSsWwctNCHbeW4Y5  
QNjKXcBr9UeReZ6+B0w8p8xSSBYE0DPNLbqccjcjYT/1D/r7Ja2Pb1W4X/tt8Dwc  
EccnjGW+3zYdAQulxLN+EZos+hlgFcNAeBHkPwbC9oDAamfsJAihGIWMa/CyBZAm  
2eF2aFtU0djDEhrVIuzrw4JKSdatqD0Bu0Q0LQ108PM/GnAzDGzz9jswfdRmj0PS  
t20XyRG+9irB4SIv47KjWXulc7h9hYrQWxD1Ny/R6TeqirA/h0iBn4ZgaY3xx3+/
```

```
tDFJ5YkR+rzEcf+w/5I3Sb0zKQ9XvERGVUJUFJbjXoes8JY0qxFZosUyaiwi+xWT
F8R/1k0+0wtH2u1e4pq265I1HBJGQc0puKpf1U/q1uACncRsi2s+EHA323T7Jkc7
3srk9wIDAQABoyMwITA0BgNVHQ8BAf8EBAMCAAEwDwYDVR0TAQH/BAUwAwEB/zAN
BgkqhkiG9w0BAQsFAAOCAgEAeX4Mf+65e8r2JMTqNKP1XnEYEw/jnq7BpjwxjNxw
Av0F2YdZifi/0U9Xr5SA+uCwYYgRB5wFpZ8trckTaLUiszTeLtlwl1Jouf2VICbY
N6RF1uHbBEYEyZl7daoF3Sd1stj/oZBPmjEP120Lus0WpYkGDdy+29fzUz271yA8
P1UE5Uq/7/0P/UIuU9pMQMbcuP0F970Dp/8i2iXYEiXPbe7s+h0GXlsrjyD65Fz
cwc9etAXuHrxCKPyCATyzW3CmU+WqE6nVCNgwh4j5r2SEeR3UZVw8Yub45IoZiE
PMcT1fa9e4hw4muKEmygdYCbiFQLsa0G/MtBv+IwpamPtoY6edjUY+00pgX70lI9
ymfnhGLyGqHLbwhZpc3gvJHWcj9mRkGr66KAHA1+H01Jw/aua0A3Fo2DBP2Rufts
g3NgE5G6zPnfcalapjt+C17Wu9TfzcIxVtTgM6g+LePgYP3tTRzAMv0DzKHSpBqW
v98pF1cG0vrVk15rLIca1CMYhP95el4qtfcXwQzKmnQBhw+emaCIudvyFRJdFM2o
f0pSiRYkpLDrqZ2fiqw+eqts80hUDOh9GvJzxtZb0ccxTbgaKxX9MtAQ1lw3vYJx
EHcp06JmUc09GtYCju2gJH29baHwldNDeP/3z9913RmnIWggh4b+G0FmPmB5X0hb
PR4=
```

-----END CERTIFICATE-----

Encrypting Secrets Using Sealed Secrets

Now that Sealed Secrets has been successfully installed, let's encrypt the contents of the `greeting-secret.yaml` that was created using `kubeseal` in listing [3.2](#) to create a new file called `greeting-secret_sealedsecrets.yaml` file.

Listing 3.6. Encrypting a Secret Using kubeseal

```
kubeseal --format yaml <greeting-secret.yaml \
>greeting-secret_sealedsecrets.yaml -n kubernetes-secrets-sealed
```

Review the contents of the newly generated `greeting-secret-sealedsecrets.yaml` file:

Listing 3.7. greeting-secret-sealedsecrets.yaml

```
apiVersion: bitnami.com/v1alpha1
kind: SealedSecret
metadata:
  creationTimestamp: null
  name: greeting-secret
  namespace: kubernetes-secrets-sealedsecrets
spec:
  encryptedData:
    greeting.message: AgAKGnqEn6MRRsDGoH2lhKTwJ0UVeUaN+Kq0Uyr13ZN
    template: #1
```

```
metadata:  
  creationTimestamp: null  
  name: greeting-secret  
  namespace: kubernetes-secrets-sealedsecrets
```

As an additional security measure, the namespace and name associated with the secret are added as part of the OAEP process so it is important that the `SealedSecret` resource be generated with this in mind.

Finally, lets verify that the newly created `SealedSecret` resource can be added to the Kubernetes cluster. Once added, the Sealed Secrets controller should decrypt the contents and create a new Secret within the same namespace.

```
kubectl apply -f greeting-secret_sealedsecrets.yaml
```

Confirm that a new secret called `greeting-secret` has been created in the `kubernetes-secrets-sealedsecrets` namespace.

```
kubectl get secrets -n default greeting-secret  
NAME          TYPE      DATA  AGE  
greeting-secret  Opaque    1     20s
```

The Sealed Secrets controller also emits Kubernetes events based on the actions that it performs and can be verified by querying for events in the default namespace.

```
kubectl get events -n kubernetes-secrets-sealedsecrets
```

LAST SEEN	TYPE	REASON	OBJECT	ME
3m38s	Normal	Unsealed	sealedsecret/greeting-secret	Se un su

The addition of events provides addition insight into the lifecycle of resources managed by Sealed Secrets.

Given the tight connection between the `SealedSecret` resource and the associated `Secret`, if the `SealedSecret` resource is removed, Kubernetes will also remove the referenced `Secret`. This is due to the fact that the `Secret` is owned by the `SealedSecret` resource. Kubernetes Garbage Collection will

cascade the deletion of resources to any resources that are owned. Additional details related to Kubernetes garbage collection can be found within the Kubernetes documentation

(<https://kubernetes.io/docs/concepts/workloads/controllers/garbage-collection/>).

By demonstrating how to make use of Sealed Secrets in our Kubernetes environment, we can feel confident that we can safely store the `SealedSecret` resource in our Version Control System without the worry of sensitive assets being easily discovered.

In the next section, we will introduce how to manage sensitive assets within Kubernetes package managers.

3.4 Managing Secrets within Kubernetes package managers

As we have seen thus far, Kubernetes provides the primitives for running complex applications: ConfigMaps and Secrets for storing configuration assets, Services for simplifying network access and Deployments for managing the desired state of container resources. However, one feature that Kubernetes does not natively provide is a mechanism for easily managing these disperse resources. This issue becomes increasingly clear as resources begin to accumulate within Infrastructure as Code repositories. Members of the Kubernetes community saw this as a challenge and there became a desire to better manage the lifecycle of Kubernetes applications in a similar fashion to any other off cluster application. Traditionally, these are features facilitated by a package manager, such as yum, apt-get or brew. The result of their efforts led to the creation and eventual popularity of Helm, a package manager for Kubernetes.

Helm simplifies the lives of application developers and consumers of Kubernetes applications by providing the following key features:

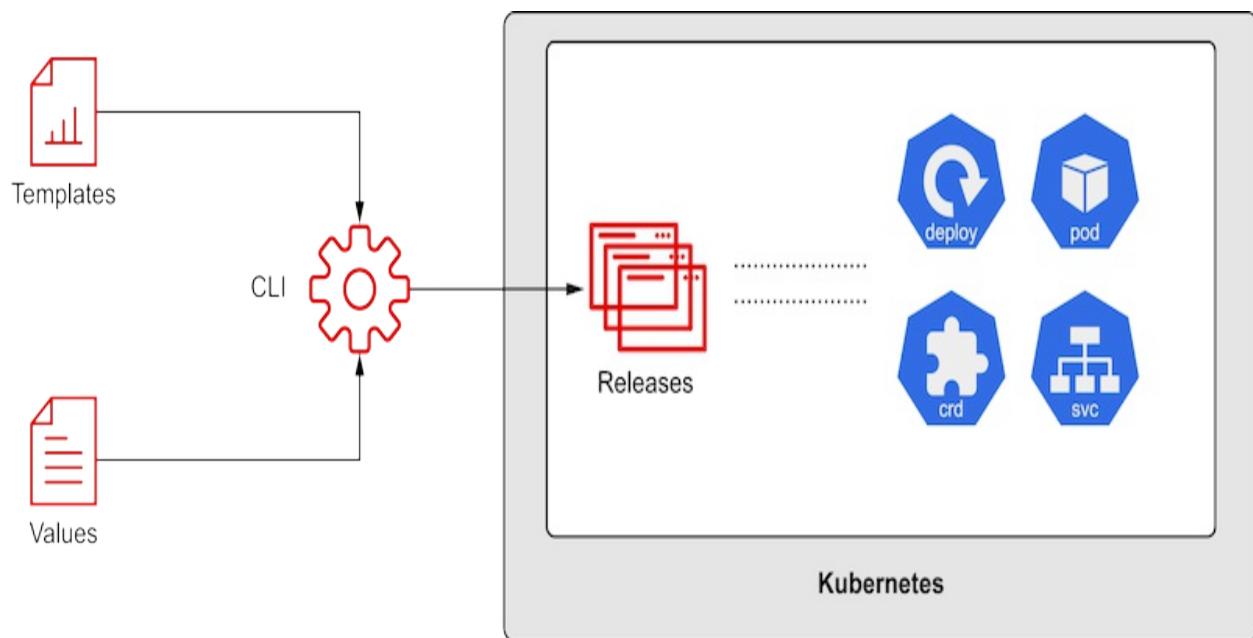
- Lifecycle management (install, upgrade, rollback)
- Manifest templating
- Dependency Management

Helm uses a packaging format known as **Charts** and contain all of the Kubernetes manifests that would be associated with an Application which are deployed as a single unit. Manifests are known as **Templates** within Helm and are processed through Helm's templating engine (golang template based with additional support from libraries such as Sprig) at deployment time.

Values are parameters that are injected into the templated resources. The combined set of rendered manifests once deployed to Kubernetes is known as a **Release**.

Finally, Helm is managed using a Command Line client tool that helps facilitate the entire lifecycle of a Chart. This will be the primary tool that will be used beginning in the next section.

Figure 3.5. A Helm Release combines Templates and Values to create Kubernetes resources



Tip

Additional information related to Helm can be found on the Helm website at <https://helm.sh/>

3.4.1 Deploying the Greeting Demo Helm Chart

The manifests associated with the Greeting Demo application that we have referred to in Chapter 2 have been packaged into a Helm Chart to demonstrate the benefits that Helm can provide. In this section, you will install the Helm CLI tool, review the Greeting Demo Helm Chart, and deploy it to your Kubernetes cluster.

First, download the Helm Command Line tool. Multiple installation options are available depending on your Operating System. The steps and instructions can be found on within the Helm project website (<https://helm.sh/docs/intro/install/>).

Once the Helm CLI has been installed, make sure that Git is available on your machine as it is needed to manage assets in version control within this chapter as well as subsequent chapters. See Appendix D for instructions on how Git can be installed and configured.

Now, clone the repository containing the Helm chart to your machine.

```
git clone https://github.com/lordofthejars/kubernetes-secrets-sou  
cd greeting-demo-helm #2
```

Once inside the chart directory, you will notice the following directory structure.

```
— Chart.yaml  
— templates  
    — NOTES.txt  
    — _helpers.tpl  
    — configmap.yaml  
    — deployment.yaml  
    — ingress.yaml  
    — secret.yaml  
    — service.yaml  
    — serviceaccount.yaml  
— values.yaml
```

The *Chart.yaml* is the manifest for the Helm chart and contains key metadata including the name of the chart as well as the version. The *templates* directory contains the Kubernetes resources that will be deployed to the cluster when the chart is installed. The key difference is they are now templated resources instead of the raw manifests that we have been working

with thus far.

Listing 3.8. secret.yaml

```
{ {- if not .Values.configMap.create -} } #1
apiVersion: v1
kind: Secret
metadata:
  name: {{ include "greeting-demo.fullname" . }}
  labels:
    {{- include "greeting-demo.labels" . | nindent 4 -}}
type: Opaque
stringData:
  greeting.message: {{ required "A valid greeting message is requ
{{- end -}}
```

The `include` function references named template that are present within the `_helpers.tpl` file. A full overview of templates and the Helm directory structure can be found within the Helm documentation.



Note

Additional Kubernetes resources, such as an Ingress and Service Account, are also included in this Chart. These are created as part of the typical boilerplate when the `helm create` command is used. By default, these resources are not deployed during typical usage of this chart, but can be if desired by way of setting the appropriate Values.

with the templates are located in the `Values.yaml` file. By browsing through the file, you will notice many key attributes including the number of replicas as well as the image location and tag. At the bottom of the file, you will notice a property called `greeting.message` with no value specified

```
greeting:
  message:
```

If you recall the snippet from listing 3.8, this property is injected by referencing the `$.Values.greeting.message` in the `secret.yaml` file in the templates directory. Also of note is the `required` function that enforces that a value be set before this chart can be installed. Values can be specified in a

number of ways including files or using the command line and Helm makes use precedences in order to determine which property is ultimately applied. Those defined in the `values.yaml` have the lowest priority.

Before we install this chart to the Kubernetes cluster, let's first start by creating a new namespace called `kubernetes-secrets-helm` and change the current content into the newly created namespace.

```
kubectl create namespace kubernetes-secrets-helm  
kubectl config set-context --current --namespace=kubernetes-secre
```

Next, since a value for `greeting.message` must be provided, create a new file called `values-kubernetes-secrets.yaml` containing the following contents:

Listing 3.9. `values-kubernetes-secrets.yaml`

```
greeting:  
  message: Hello from Helm!
```

Next, install the Helm chart by providing a name for the *Release*, the chart location, and a reference to the values file that contain the required property

```
helm upgrade -i greeting-demo . -f values-kubernetes-secrets.yaml
```



Note

The `helm upgrade` command was used with the `-i` flag as it provides an idempotent method for installing Helm Charts. If an existing chart is present, it will be upgraded. Otherwise a new installation will occur.

If the installation was successful, an overview of the release is provided along with the rendered contents of the `Notes.txt` file contained from the *templates*.

Curl the IP address and port exposed by the minikube service to confirm that the greeting that was set in the Helm value and stored in the `greeting-secret` secret is presented.

```
curl $(minikube ip):$(kubectl get svc --namespace kubernetes-secr
```

```
greeting-demo -o jsonpath=".spec.ports[*].nodePort")/hello
```

3.4.2 Using Helm Secrets

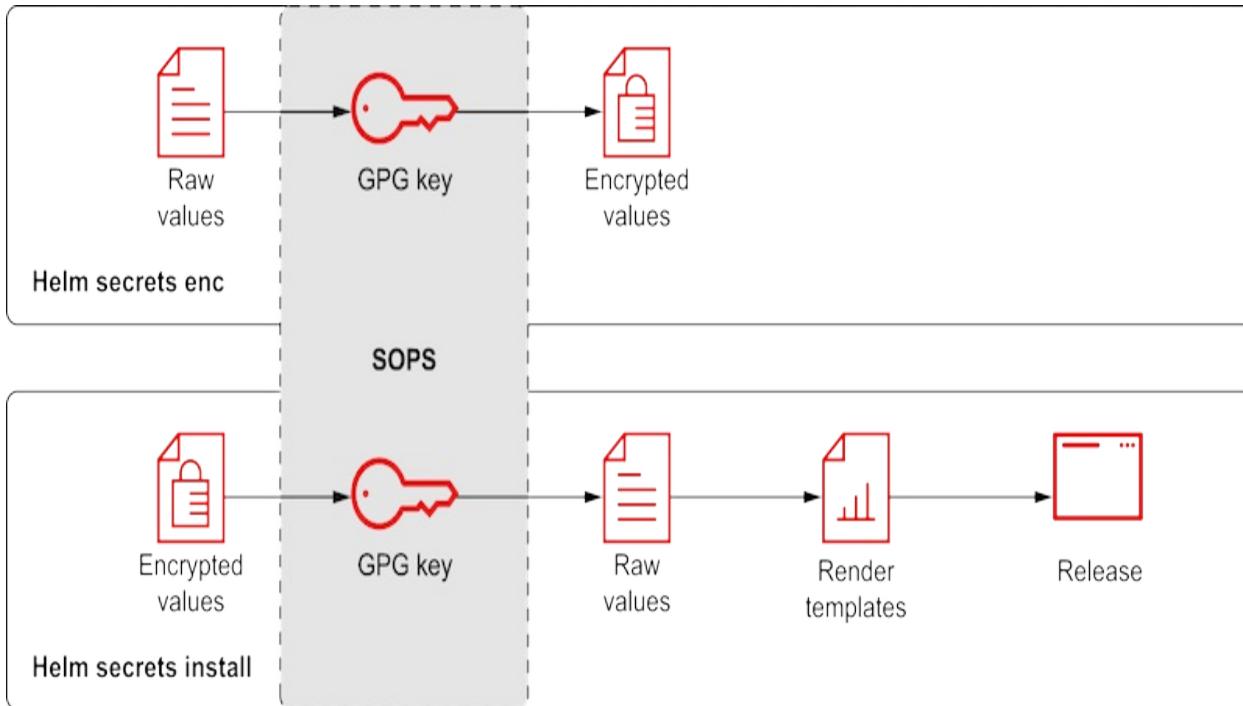
While we were successful in deploying the application as a Helm chart, we are once again faced with the challenge of how to manage sensitive content stored within the `values-kubernetes-secrets.yaml` file.

As you might expect, other members of the Helm and Kubernetes community sought to derive a solution to provide security to Helm values so that can be stored at rest. One of the most popular options available is Helm Secrets, a Helm plugin that provides integration with various secrets managements backend. Plugins in Helm are external tools that are not part of the Helm code base, but can be accessed from the Helm CLI.

Helm Secrets, by default uses SOPS (Secrets OPerationS), from Mozilla, as its default secrets management driver. SOPS is a tool that encrypts key:value file types, such as YAML, and integrates with cloud (AWS/GCP KMS) and non-cloud secrets management solutions.

For the integration of SOPS as the backend for Helm Secrets, the most straightforward and easiest to setup for our purpose is to leverage PGP (Pretty Good Privacy). PGP, in a similar fashion to Sealed Secrets, uses asymmetric public/private key encryption to secure the content of files and has been a popular method for transmitting emails securely. At encryption time, a random 256 bit data key is used and passed to PGP to encrypt the data key and then the properties within the file.

Figure 3.6. Encrypting secrets and using them as part of a Helm release using Helm Secrets



Let's prepare our machine with the software and configurations necessary to protect the sensitive properties for our helm chart.

PGP (GNU Privacy Guard) is an open standards implementation of the proprietary PGP, and the corresponding gpg tool will enable us to manage our keys appropriately. Check Appendix E for further instructions on how it can be installed.

Once the gpg CLI has been installed, create a new public/private key:

```
gpg --generate-key
```

Enter your name and email address. When prompted to enter a passphrase, omit entering any value.



Note

While it may seem counterintuitive to leave the GPG passphrase blank, it simplifies the integration with automation tools (some do not support passphrases), but also reduces the need to manage yet another sensitive asset.

After the public/private key were successfully created, you can confirm by listing the keys in the keyring.

Listing 3.10. Listing GPG Keys

```
gpg --list-keys

pub    rsa2048 2020-12-31 [SC] [expires: 2022-12-31]
      53696D1AB6954C043FCBA478A23998F0CBF2A552 #1
uid          [ultimate] John Doe <jdoe@example.com>
sub    rsa2048 2020-12-31 [E] [expires: 2022-12-31]
```

Make note of the value starting with 5369 as this is the fingerprint for the public key and will be used shortly when configuring SOPS.

Next, install the Helm Secrets plugin using the Helm CLI.

```
helm plugin install https://github.com/jkroepke/helm-secrets
```

SOPS will also be installed as part of the Helm Secrets installation.

Values files managed by the Helm Secrets plugin, by convention, are located in a directory called `helm_var`. Create this directory within the *greeting-demo* chart repository:

```
mkdir helm_vars
```

To complete the integration between Helm Secrets and SOPS, create a new file called in the `helm_vars` directory named `.sops.yaml` with the following content

```
---
creation_rules:
  - pgp: "53696D1AB6954C043FCBA478A23998F0CBF2A552"
---
```

Replace the content next to `pgp` with the fingerprint for your own public key discovered in listing [3.10](#).

Next, move the `values-kubernetes-secrets.yaml` to the `helm_vars` directory so that it is within the same directory as the `.sops.yaml` file so that

SOPS will decrypt the file appropriately.

```
---  
mv values-kubernetes-secrets.yaml helm_vars  
---
```

With the integration with SOPS complete and the desired values file in the proper location, use Helm Secrets to encrypt the file:

Listing 3.11. Encrypting Using Helm Secrets

```
helm secrets enc helm_vars/values-kubernetes-secrets.yaml
```

```
Encrypted values-kubernetes-secrets.yaml
```

Confirm that the contents of the *values-kubernetes-secrets.yaml* file are encrypted similar to the following:

Listing 3.12. values-kubernetes-secrets.yaml

```
greeting:  
  message: ENC[AES256_GCM,data:SYfMBpax8mT0qzPed3ksjA==,iv:OrN/  
sops:  
  kms: []  
  gcp_kms: []  
  azure_kv: []  
  hc_vault: []  
  lastmodified: '2020-12-31T04:21:40Z'  
  mac: ENC[AES256_GCM,data:h2fQPc9hzmGMaKIE73aYU2TxbwVYQQLRcYHW  
  pgp:  
    - created_at: '2020-12-31T04:21:39Z'  
      enc: |  
        -----BEGIN PGP MESSAGE-----  
  
        hQEMA0e7sMUYmEkyAQgAhrnbGtCkbRwEDky1TmWTHXeKhoEx+2bbD  
        iDwat80MKu21GKgnVj2RAMIxwyjaLdoGY+pDXYxUJ5StFojh1bJbk  
        uvkpeYNwLtAZWd6Shj11vAkVEDMsh3xFtv9ot2uwL/DuxmSvoIR50  
        UzsK0SMWQjyIT69oUK1JYd+Nwj0sv1oWJMAkra367EZxKzKKi1eKF  
        +P8ctjQqeWi8bC/wN6PdRGVYfZD8bF3CxgdtYUKHRseNvjX2H6rD6  
        bGf0u4n5SccgGftgnYI8nXL7vnAntuLREz6XDnLQDNJeAVzDt623n  
        HYfaEqBWI8bcPfBwHv3g9F1sAk86W86IR6pB0m0wD/twW9/J7InW9  
        oWVWwF8IZzNWVb6Sj16EXIaB+ssJX0tfWXyBD83w8Q==  
        =TH/P  
        -----END PGP MESSAGE-----
```

```
fp: 53696D1AB6954C043FCBA478A23998F0CBF2A552 #2
unencrypted_suffix: _unencrypted
version: 3.6.1
```

Note that the file has been separated into two primary sections.

1. Keys from the original file are retained. Values are now encrypted
2. SOPS related metadata including enough information to enable the encrypted values to be decrypted

The contents can be easily updated as necessary using the `helm secrets edit` command. To denote that we are now using Helm Secrets to manage the content of our values file in a secure manner, update the value of the `greeting.message` property to read "Hello from Helm Secrets!":

```
helm secrets edit helm_vars/values-kubernetes-secrets.yaml
```

Update the contents of the file:

```
greeting:
  message: Hello from Helm Secrets!
```

Now, upgrade the chart with the updated encrypted values file using Helm Secrets

```
helm secrets upgrade greeting-demo . -i -f helm_vars/values-kuber
```

Revision 2 should be displayed indicating a successful release. However, if you attempt to query the endpoint exposed by the application, it will not reflect the updated values since only the underlying Secret was modified and a solution similar to the **Reloader** introduced in Chapter 2 is not in use.

Delete the running pod which will allow the updated value to be injected into the newly created pod.

```
kubectl delete pod -l=app.kubernetes.io/instance=greeting-demo
```

Once the newly created pod is running, query the application endpoint to confirm the response displays the property contained within our encrypted values file.

```
curl $(minikube ip):$(kubectl get svc --namespace kubernetes-secrets greeting-demo -o jsonpath=".spec.ports[*].nodePort")/hello
```

One benefit to using Helm Secrets is that it provides an introduction to SOPS, a tool that can be used to protect sensitive key/value files for secure storage outside the Helm ecosystem.

The management of sensitive assets does not end once the values have been encrypted. In the next section, we will discuss how rotating secrets can be used to increase the overall security posture beyond day 1.

3.5 Rotating Secrets

Individuals and organizations go to great lengths in order to protect sensitive information that could potentially gain access to critical systems. However, regardless of the strength of any secure value, or the tools that are used to protect access to them, there is always the potential for compromise. The key is minimize or reduce the potential.

With that in mind, one of the methods that can be used to reduce the attack vector is to implement some form of secret rotation. Rotation can occur in two primary areas:

1. The actual value being secured
2. The keys or values used to generate the encrypted asset

Rotation of the sensitive asset is a concept that most of us should be familiar with, such as resetting passwords on a regular basis. This practice, however, has been known to fall short in the context of managed assets since systems make use of the asset instead of a human.

Fortunately, each of the tools that have been introduced thus far to secure sensitive assets at rest (Ansible Vault, Sealed Secrets, and Helm Secrets) support some form of rotation.

3.5.1 Ansible Vault Secret Key Rotation

Encrypted files generated by Ansible Vault can be rekeyed to allow a

different password to be used to secure and access the stored asset using the `rekey` subcommand of `ansible-vault`.

Using the `greeting-secret_ansible.yaml` file that was encrypted in listing [3.11](#), use the `ansible-vault rekey` subcommand to start the rekeying process, enter the existing password and then a new password when prompted:

```
ansible-vault rekey greeting-secret_ansible.yaml
```

```
Vault password:  
New Vault password:  
Confirm New Vault password:  
Rekey successful
```

The contents of the `greeting-secret_ansible.yaml` file has been updated.

3.5.2 Sealed Secrets Key Rotation

As described in [3.5.2](#), whenever the Sealed Secret controller starts, it checks if an existing public/private key is available (with the label `sealedsecrets.bitnami.com/sealed-secrets-key=active`). Otherwise, a new keypair will be generated.

Sealing keys themselves are renewed automatically (new secret created) every 30 days and the controller will consider any secret with the `sealedsecrets.bitnami.com/sealed-secrets-key=active` label as a potential key used for decryption.

However, rotation can be initiated at any time, whether it be due to a compromise or other reasons, by either setting the `--key-cutoff-time` flag or by using the `SEALED_SECRETS_KEY_CUTOFF_TIME` environment variable on the controller deployment. The value using either method must be in RFC1123 format.

Force the Sealed Secrets controller to regenerate a new keypair by executing the following command to add an environment variable on the `sealed-secrets-controller` deployment:

```
kubectl -n kube-system set env deployment/sealed-secrets-controller  
SEALED_SECRETS_KEY_CUTOFF_TIME="$(date -R)"
```

A new rollout of the *sealed-secrets-controller* will be initiated. Confirm the new keypair was generated:

```
kubectl -n kube-system get secrets \  
-l=sealedsecrets.bitnami.com/sealed-secrets-key=active
```

NAME	TYPE	DATA	AGE
sealed-secrets-key6kdnd	kubernetes.io/tls	2	25m26s
sealed-secrets-keyqdrb5	kubernetes.io/tls	2	47s

With the new private key available to the controller, Secrets associated to existing *SealedSecrets* can be reencrypted or new Secrets can be encrypted using the `kubeseal` CLI.



Note

Existing keys are not removed whenever a new key is added as it is added to the list of active keys. Old keys can be removed manually only after the new key has been created.

3.5.3 SOPS Secret Key Rotation

SOPS, the secrets management tool underneath Helm Secrets also supports key rotation. There are two mechanisms for which rotation can be implemented within SOPS:

1. The GPG key itself
2. The data key that is used at encryption time

The most straightforward option is to rotate the data key used to encrypt the file. To accomplish this task with our existing Helm values file located within the `helm_vars` folder, use the SOPS tool itself and passing the `-r` flag along with the location of the file to rotate:

```
sops -r --in-place helm_vars/values-kubernetes-secrets.yaml
```

You can confirm that SOPS has updated the file by verifying the `lastmodified` property underneath the `sops` section.

In addition to the data key being rotated, the master GPG key used can also be updated. To do so, you can create a new GPG key or reference an existing GPG and pass the associated fingerprint of the key by using the `--add-pgp` flag as shown below:

```
sops -r --in-place --add-pgp <FINGERPRINT> \
helm_vars/values-kubernetes-secrets.yaml
```

To remove the old key, execute the above command, but replace `--add-pgp` with `--rm-pgp` and use the fingerprint of the key you wish to remove. In any event, be sure to update the contents of the `helm_vars/.sops.yaml` file with the fingerprint of the key that you would like Helm Secrets to use to manage the secure assets.

Regardless of the secrets management tool being used, once rotation has been completed, it is important that any system or application that is dependant on the secure asset are updated appropriately in order to reduce the potential of downtime or error due to misconfiguration.

3.6 Summary

- Expressing Kubernetes resources declaratively allow them to be captured and stored in a Version Control Systems
- Tools, such as Ansible and Helm, have support for storing sensitive resources at rest securely
- Operators automate actions in Kubernetes environments and can be used to encrypt sensitive values from within the cluster
- SOPS is a general purpose tool for encrypting various file formats and includes integration with KMS providers
- Secret key rotation replaces existing encryption keys by generating new cryptographic keys and reduces the risk of a compromise

4 Encrypting Data at Rest

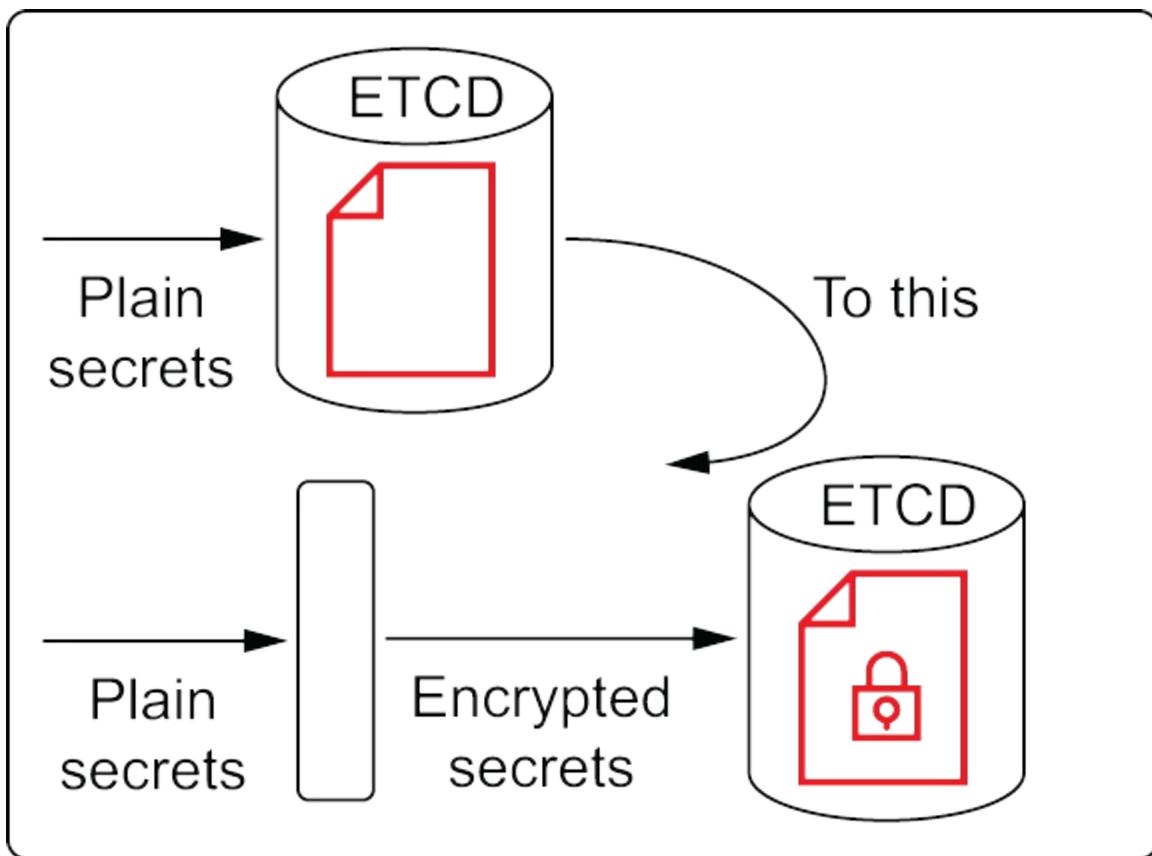
This chapter covers

- Data encryption at rest for Kubernetes cluster storage.
- Enabling the KMS provider for data encryption.

In chapter 3, we saw how to protect secrets when storing them in Git, but this is just one place where secrets are stored; another place is inside the Kubernetes cluster.

We'll demonstrate that secrets are not encrypted by default by querying directly the etcd database. Then we'll walk through the process of encryption data at rest, how it is enabled in Kubernetes to have secrets encrypted. The process is summarized in the following figure [4.1](#).

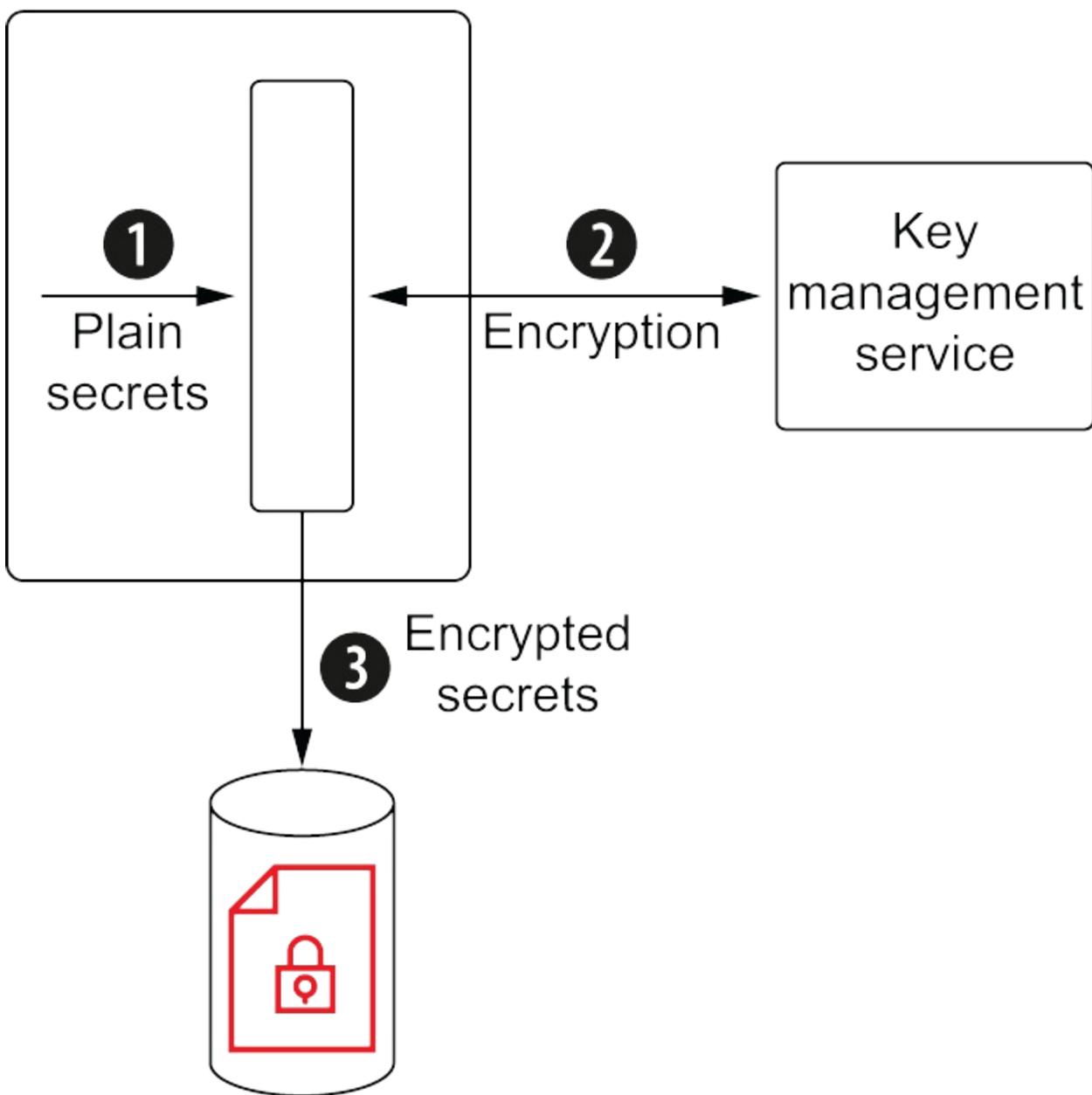
Figure 4.1. From plain text secrets to encrypted secrets



Finally, we will make the process secure by using a Key Management Service (KMS) to manage encryption keys as shown in figure [4.2](#).

Figure 4.2. Key Management Service (KMS) for managing keys

Kubernetes cluster



4.1 Encrypting secrets in Kubernetes

Imagine, we've got an application that needs to connect to a database server; obviously, a username and password are required to access it. These configuration values are secrets and they need to be stored correctly so if the system (or the cluster) is compromised, secrets are kept secret and the attacker is not be able to exploit them to access to any part of our application.

The solution is encrypting these secrets so if they are compromised, the attacker would get a chunk of bytes instead of the real values.

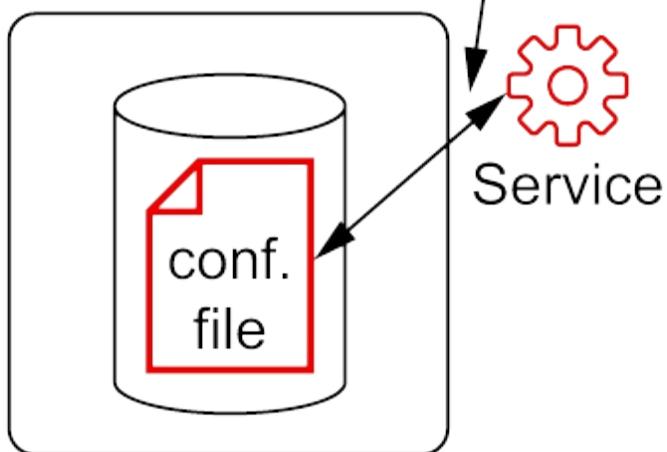
4.1.1 Data at rest vs Data in motion

Speaking in terms of data flow, we detailed the characteristics of *Data at rest* extensively in Chapter 3 as the term used for *persisted data* that is infrequently changed.

On the other hand, *Data in transit*, or *Data in motion*, is data that is moving from one location to another usually through the network. We can protect *Data in transit* by using *mutual TLS* protocol in the communication between parties, but this is out of the scope of the book, and we are going to focus on how to protect *Data at rest*.

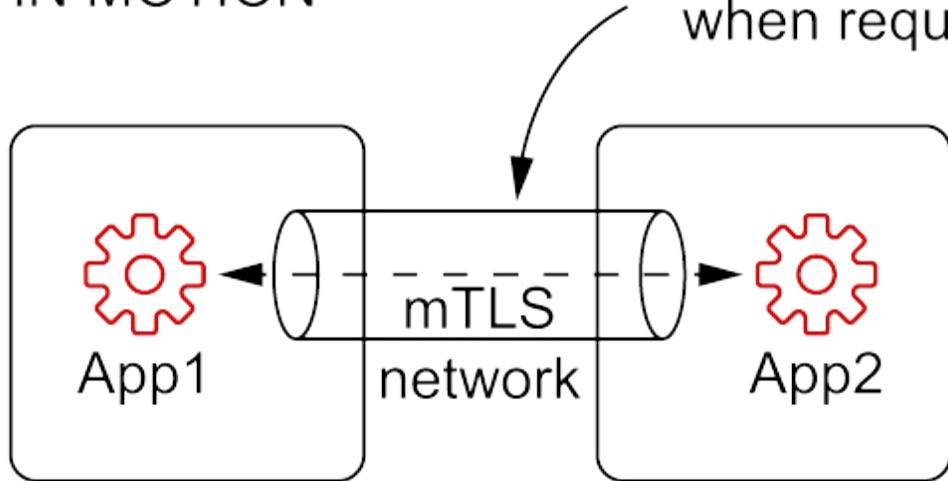
Figure 4.3. Data in transit needs to be encrypted at communication level

DATA AT REST



Loads configuration
at boot-up time

DATA IN MOTION



Data is transmitted
when required

Before we address the problem of not encrypting data at rest, let's create a plaintext secret and get it by querying directly etcd server as an attacker might do.

4.1.2 Plain Secrets

Let's create a secret using the kubectl tool and querying etcd server using the etcdctl CLI.

Creating the Secret

In a terminal window, move to default Kubernetes namespace and create a new secret with two key/values.

```
kubectl config set-context --current --namespace=default  
kubectl create secret generic db-secret --from-literal=username=d
```

We can list the created secret by using kubectl tool:

```
kubectl get secrets
```

NAME	TYPE	DATA
db-secret	Opaque	2

Installing etcdctl

etcdctl is a command-line client for interacting with etcd server and it is used for querying keys stored in the database among other operations. This tool might be really helpful for us to understand how data is stored in etcd database.

The installation process of the tool might differ depending on the OS you are using, for this reason, we leave here the link to the official installation guide (<https://github.com/etcd-io/etcd/releases/tag/v3.4.14>) of the etcd version we use in this book to query the database.

Accessing etcd

etcd server is running in kube-system namespace under 2379 port. As we are using minikube, we can use the port-forwarding feature to access the etcd server directly from our local machine.

In a terminal window, run the following command to expose etcd in the localhost host.

```
kubectl port-forward -n kube-system etcd-minikube 2379:2379
```



Important

If running the previous commands you get the following error Error from server (NotFound): pods "etcd-minikube" not found, run the following command to get the name for your environment.

```
kubectl get pods -n kube-system
```

NAME	READY	STATUS	RESTARTS
coredns-66bff467f8-mh55d	1/1	Running	1
etcd-vault-book	1/1	Running	1
kube-apiserver-vault-book	1/1	Running	0
kube-controller-manager-vault-book	1/1	Running	1
kube-proxy-lbhd6	1/1	Running	1
kube-scheduler-vault-book	1/1	Running	1
storage-provisioner	1/1	Running	1

In this case, the etcd Pod name is etcd-vault-book.

The second step to access etcd server is copying the etcd certificates from the running Pod to the local machine. Open a new terminal window and copy the certificates by using kubectl tool:

```
kubectl cp kube-system/etcd-minikube:/var/lib/minikube/certs/etcd  
kubectl cp kube-system/etcd-minikube:/var/lib/minikube/certs/etcd
```



Important

etcd-minikube is a directory that matches the etcd Pod name. Modify accordantly to your environment.

Finally, we can configure etcdctl to connect to the etcd server and query for the secret created in the previous step. etcd organizes its content in key/value format; for the specific case of secret objects, the content is stored in keys following the following format:

/registry/secrets/<namespace>/<secret_name>.

Let's execute the following commands in a terminal window as shown in listing [4.1](#):

Listing 4.1. Configures etcdctl

```
export \ #1
ETCDCTL_API=3 \
ETCDCTL_INSECURE_SKIP_TLS_VERIFY=true \
ETCDCTL_CERT=/tmp/peer.crt \
ETCDCTL_KEY=/tmp/peer.key

etcdctl get /registry/secrets/default/db-secret #2
```

The output should be similar to the one shown in listing [4.2](#). Notice that although the output is not perfectly clear, it is not hard at all to see the secret content.

Listing 4.2. etcdctl output

```
/registry/secrets/default/db-secret
k8s
```

```
v1Secret
N
      db-secretdefault"*$df9e87f7-4eed-4f5b-985a-7888919198472
password
      devpassword #1
usernamedevuserOpaque"
```

We can stop the port-forwarding as we do not need anymore for now by aborting the process (push `ctrl+C` on the first terminal)

You understand now that if the etcd server is compromised, nothing blocks an attacker to get all secrets in plain text. In the following section of this chapter, we explore the first solution to store secrets encrypted.

4.1.3 Encrypting Secrets

To use encryption data at rest, we'll need to introduce a new Kubernetes object named `EncryptionConfiguration`. In this object, you specify what Kubernetes object you want to encrypt, it can be secret objects but it's actually possible to encrypt any other Kubernetes objects. Also, you need to specify the secrets provider which is a pluggable system where we specify

the encryption algorithm and the encryption keys to be used.

At the time of writing the book the following providers are supported:

- **identity**

No encryption enabled, the resources are written as-is.

- **aescbc**

AES-CBC with PKCS#7 padding algorithm. It is the best option for encryption at rest.

- **secretbox**

XSalsa20 and Poly1305 algorithm. It is a new standard but it might not be suitable in environments with high levels of review.

- **aesgcm**

AES-GCM with a random nonce algorithm. It is only recommended if you implement an automatic key rotation.

- **kms**

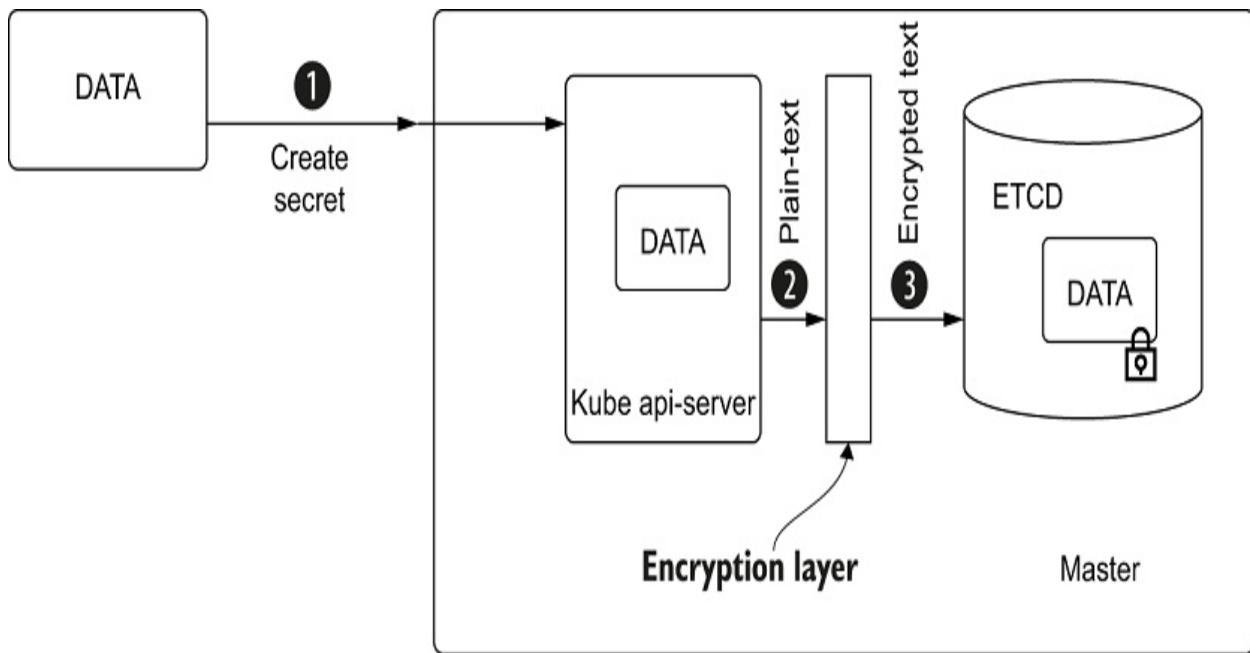
Uses an envelop encryption schema. Key encryption keys are managed by the configured Key Management Service (KMS). This is the most secure way and we are going to explore it later on in this chapter

To examine how encryption data at rest works in Kubernetes, we'll repeat the same exercise done in the previous section but at this time having Kubernetes configured to encrypt secrets.

Enabling Encryption Data at Rest

It is important to understand that encryption data at rest happens on the `kube-apiserver` that is running in a master node. If it is enabled, every time that a Kubernetes object is sent to the Kubernetes cluster, the `kube-apiserver` delegates to the encryption configuration part to encrypt the objects before they are sent to the `etcd` database to be stored. Obviously, when a secret needs to be consumed, it is decrypted automatically, so from the point of view of a developer, nothing special is required special; they work as usually do. Figure [4.4](#) illustrates what was just described.

Figure 4.4. Encryption layer encrypts secrets automatically before sending them to etcd



Let's generate an `EncryptionConfiguration` object to set that any Kubernetes Secret is encrypted using `aescbc` algorithm and with a random encryption key as shown in listing 4.3.

Listing 4.3. `EncryptionConfiguration`

```
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  - resources:
    - secrets #1
providers:
  - aescbc:
    keys:
      - name: key1
        secret: b6sjdRWAPhtacXo8m01cfgVYWXzwuls3T3NQ0o4TBhk= #2
  - identity: {}
```



Tip

To generate a random key in Base64, we can use a tool such as `openssl` or `head`:

```
openssl rand -base64 32
```

```
head -c 32 /dev/urandom | base64 -i -
```

As you might remember, the encryption process occurs in the kube-apiserver process, and this implies that we need to materialize the EncryptionConfiguration file into master nodes where it is running.

The process of accessing into master node might differ depending on the Kubernetes platform, in minikube it is done by running the `minikube ssh` command to obtain an SSH session on the master node. When inside the master node, run the `sudo -i` command to execute the following commands as a superuser.

Listing 4.4. SSH'd minikube

```
minikube ssh  
sudo -i
```



Tip

If you get an error like *Error getting config: then you need to specify the minikube profile* with the `-p` flag.

You can list current active profile with `minikube profile list`:

```
minikube profile list
```

Profile	VM Driver	Runtime	IP	Port	Version
istio	virtualbox	docker	192.168.99.116	8443	v1.18.6
kube	virtualbox	docker	192.168.99.117	8443	v1.18.6

```
minikube ssh -p kube.
```

And then inside the SSH'd instance, run the `sudo` command.

Listing 4.5. Update to superuser

```
sudo -i
```

At this point, we are inside the `minikube` virtual machine where `kube-apiserver` is running. Let's create a new file at `/var/lib/minikube/certs/encryptionconfig.yaml` with the content shown in listing 4.6:

Listing 4.6. `encryptionconfig.yaml`

```
echo "
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  - resources:
    - secrets
  providers:
    - aescbc: #1
      keys:
        - name: key1
          secret: b6sjdRWAPhtacXo8m01cfgVYWXzwuls3T3NQ0o4TBhk=
    - identity: {}
"
" | tee /var/lib/minikube/certs/encryptionconfig.yaml #2
```

The file is created in the master node, now we can quit the SSH terminal by typing `exit` twice.

```
exit
```

```
exit
```

We are back now on our computer, but we still have to do the last step before secrets are encrypted. That is configuring `kube-apiserver` to pick up the `EncryptionConfiguration` file created in the previous step.

To configure the `kube-apiserver` process, we need to set the argument `--encryption-provider-config` argument value to the `EncryptionConfiguration` path.

The easiest way in `minikube` is by stopping the instance and start it again using the argument `--extra-config`.



Tip

If you are not using `minikube`, the following link <https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver/> gives information about setting this configuration property in a Kubernetes `kube-apiserver`.

```
minikube stop  
minikube start --vm-driver=virtualbox --extra-config=apiserver.en
```

4.1.4 Creating the Secret

In a terminal window, move to the default Kubernetes namespace and create a new secret named `db-secret-encrypted` with two key/values.

```
kubectl config set-context --current --namespace=default  
kubectl create secret generic db-secret-encrypted --from-literal=
```

At this point, the secret is created in the same way as before, but let's explore how the data is stored inside etcd.

Accessing etcd

Let's repeat exactly the same process as we did previously in the [Accessing etcd](#) section to get the content of the secret `db-secret-encrypted` and validate that now it is stored encrypted instead of in the plain text.

In one terminal window, expose the etcd server in localhost:

```
kubectl port-forward -n kube-system etcd-minikube 2379:2379
```

In another terminal, let's repeat the process of copying the etcd certificates and configure the `etcdctl` using environment variables:

```
kubectl cp kube-system/etcd-minikube:/var/lib/minikube/certs/etcd  
kubectl cp kube-system/etcd-minikube:/var/lib/minikube/certs/etcd  
export \
```

```
ETCDCTL_API=3 \
ETCDCTL_INSECURE_SKIP_TLS_VERIFY=true \
ETCDCTL_CERT=/tmp/peer.crt \
ETCDCTL_KEY=/tmp/peer.key
```

We can now query etcd to get the value of the db-secret-encrypted key to validate that it is encrypted and impossible to decipher its values.

```
etcdctl get /registry/secrets/default/db-secret-encrypted
```

The output should be similar to the followig one:

```
/registry/secrets/default/db-secret-encrypted
cm?9>?*?-??c?????Ø?~?6I?????=@@????e?????.??
8Y
t?p ?b? ?V?????w??6????l????v????Ey?q.?
```

Unlike the previous section, the secrets are encrypted in kube-apiserver and then sent to be stored in the etcd server.

We can stop port-forwarding as we do not need anymore for now by aborting the process (push `ctrl+c` on the first terminal).

You can now see that an attacker could get access to the etcd server or access to a backup of etcd (please protect against this), but secrets are encrypted using the keys configured in the `EncryptionConfiguration` object. But is it enough?

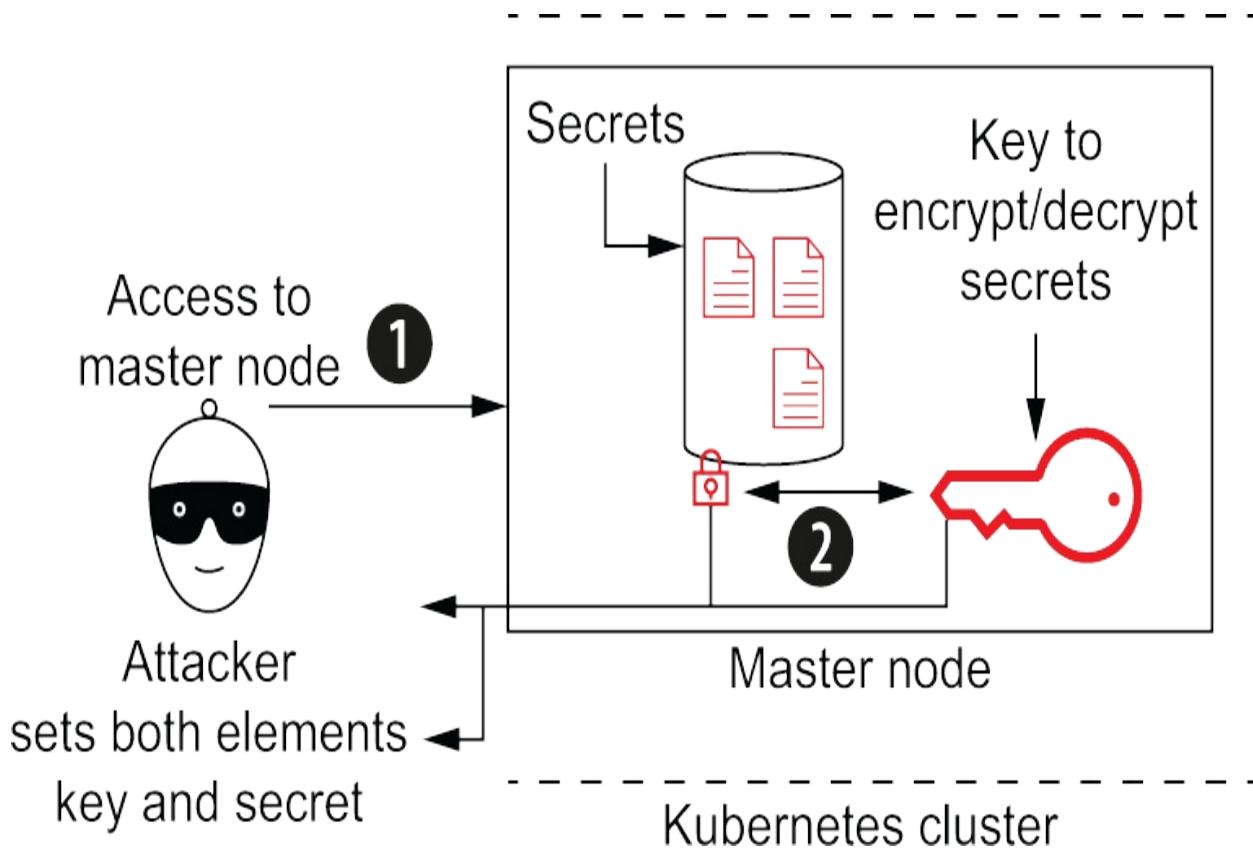
Security of the keys

Using `EncryptionConfiguration`, we've increased the hurdles attackers have to overcome to access the plain text secrets, but still some weakness around.

The reason for that is that the encryption keys are stored in plaintext on the file system in the master node. If the attacker gets access to the master machine, remember that is where the `EncryptionConfiguration` file is stored, they can grab the keys, query the encrypted secrets, and decrypt them with the keys stolen from the master node.

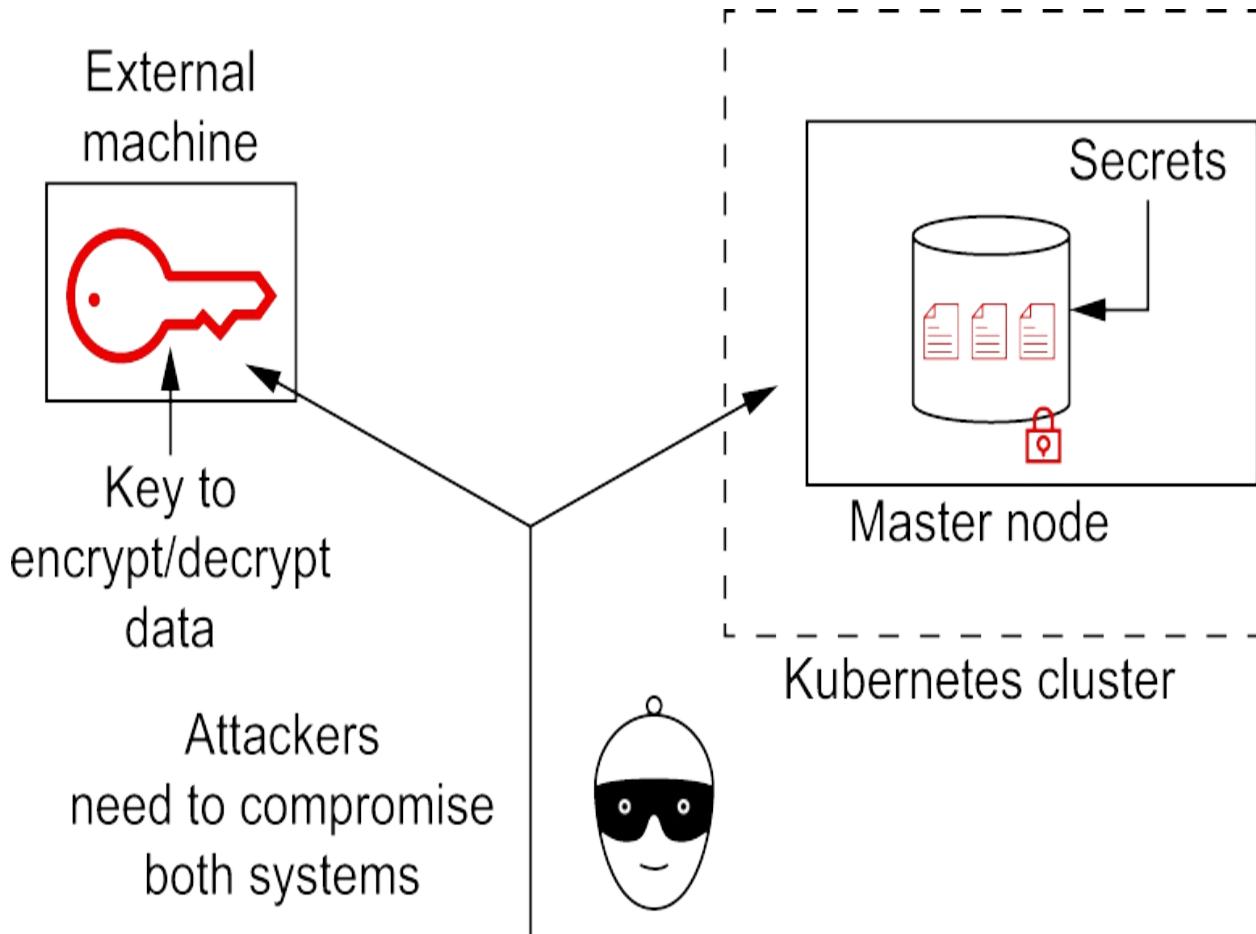
One of the big issues here is that the keys that encrypt data live alongside the data itself. Any attacker that has access to our master node can potentially get secrets and the keys to decrypt them. We've not improved the security a lot in case that a master node is compromised.

Figure 4.5. A compromised cluster implies that both data and keys to decrypt data are exposed



To avoid this vector attack, encryption keys and data should be stored in separate machines. In this way an attacker needs to compromise multiple systems to get access to the secrets as shown in the figure [4.6](#).

Figure 4.6. Splitting data and keys into different machines, makes the system safer



In addition to this big issue, there are other drawbacks to take into consideration when using the previous approach:

- The keys need to be generated manually using external tooling.
- The key management process is done manually.
- The key rotation is a manual process that requires an update to the `EncryptionConfiguration` file with the implication of a restart on the `kube-apiserver` process.

Clearly, we've improved our security model by encrypting secrets, and it might be enough depending on your use cases and the level of security you expect, but there is still room for improvement. In the following section, we'll dig into how to use a Key Management Service (KMS) in Kubernetes to store in different machines encryption keys and encrypted data.

4.2 Key Management Server

The previous application secrets were encrypted but keys used to encrypt them were not protected. Any unwanted accesss would result of a lost of them and give the possibility to the attacker to decrypt application secrets. Let's improve the previous application to protect these keys when using Kubernetes.

To increase the security of keys used to encrypt data we need that a Key Management Service (KMS) is deployed outside the Kubernetes cluster. In this way, keys are managed outside the cluster meanwhile secrets are stored inside the cluster (etcd).

This new approach makes it difficult for a possible attacker to get our secrets as two systems must be compromised; first of all, he needs to get access to etcd or a disk backup to take the secrets, and assuming he got them, they are just a bunch of encrypted bytes.

The second thing is getting the keys to decrypt them, but the big difference with the previous section is that now keys are not in the same machine nor stored in plaintext.

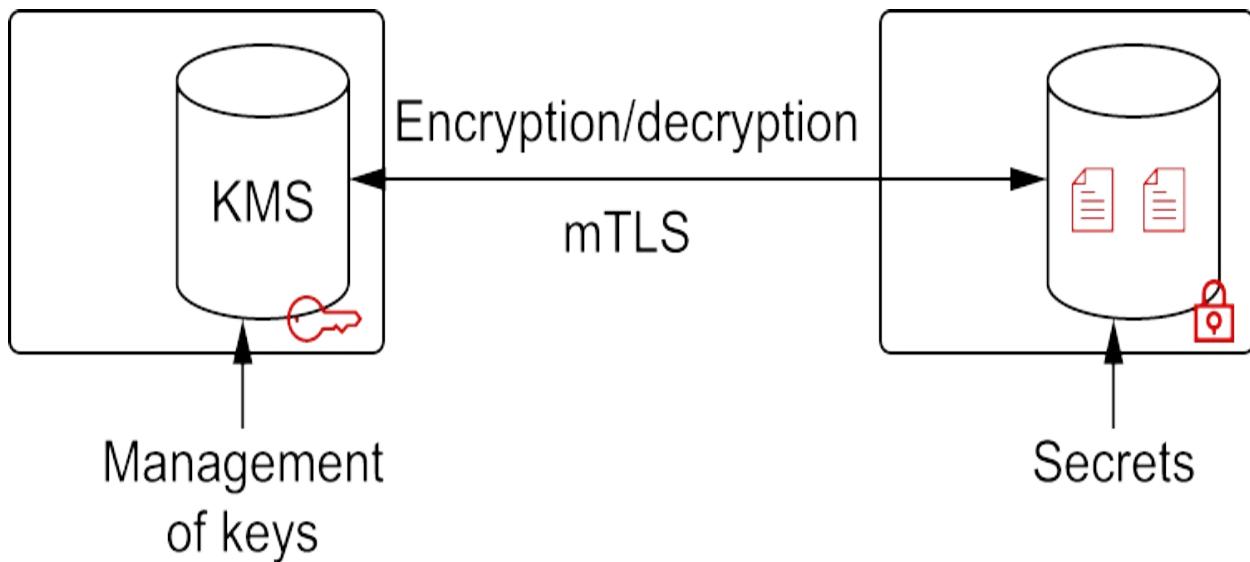
A second system needs to be compromised as the attacker needs to get access to the Key Management Service and get the keys used at encryption time to decrypt the secrets.

Of course, it is still possible, but we've added another layer of protection that needs to be broken.

As we want to keep our secrets more protected and resilient to a possible attack, we need to move from storing keys and encrypted data in the same machine to have a clear differentiation between the location where keys and encrypted data are stored.

A Key Management Service is a server that centralizes the management of encryption keys and provides some capabilities for handling cryptographic operations on data in-transit. This makes the perfect tool to have keys and data storage completely separated.

Figure 4.7. Keys used to encrypt/decrypt are managed in the Key Management Service (KMS)



4.2.1 Kubernetes and KMS provider

We've seen in [Enabling Encryption Data at Rest](#) section that Kubernetes can use encryption at rest to encrypt secrets and different kinds of providers are supported for encrypting data. One of the providers that are integrated is the KMS provider. This provider is the recommended one when using an external Key Management Server.

One of the aspects of the KMS encryption provider is that it uses *an envelope encryption* schema to encrypt all data. It is important to understand exactly how this schema works and why it is adopted to store data in etcd.

Envelope Encryption

To use envelope encryption schema, we need three pieces of data: the data to encrypt (the secret), a data encryption key (*DEK*), and a key encryption key (*KEK*).

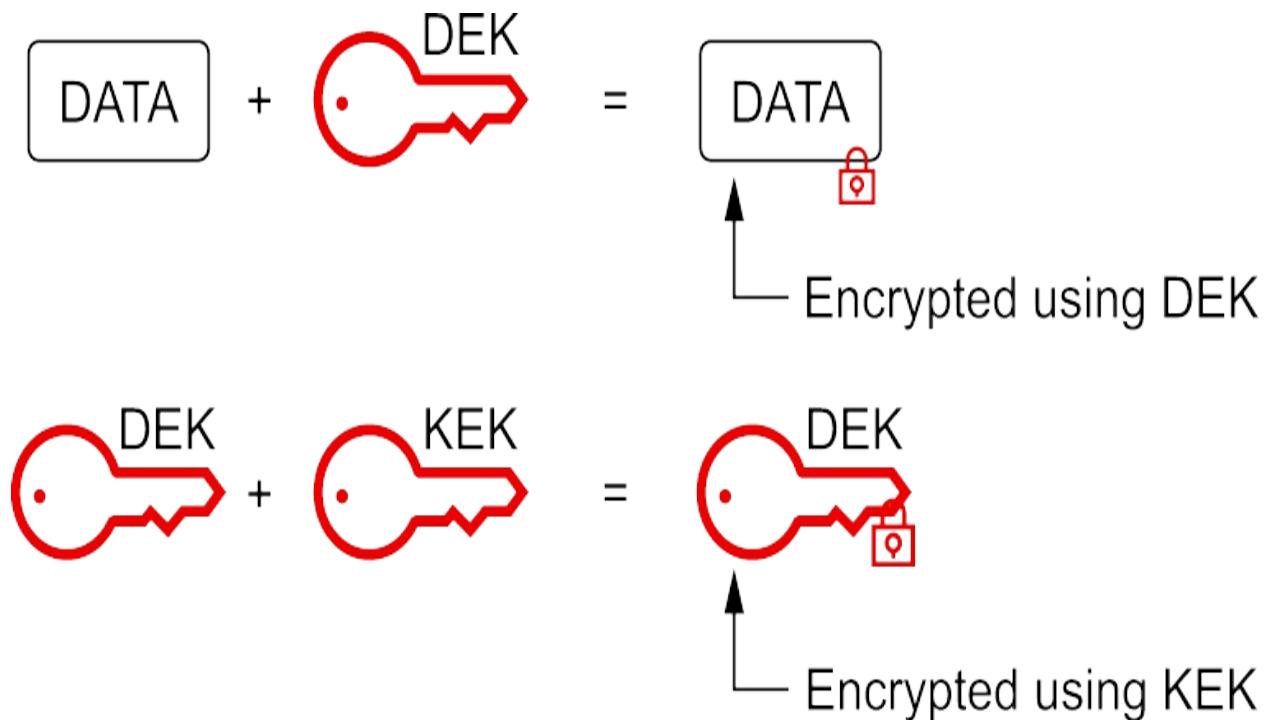
Every time that new data needs to be encrypted, a **new** data encryption key is generated and it is used to encrypt the data. As you can see, each piece of data (or secret) is encrypted by a new encryption key (*DEK*), and are created on the fly.

In addition to the data encryption key, the envelope encryption schema also

has a key encryption key. This key is used to encrypt the data encryption key (*DEK*). In contrast, the *KEK* is just generated once and it is stored in a third-party system like a Key Management Service.

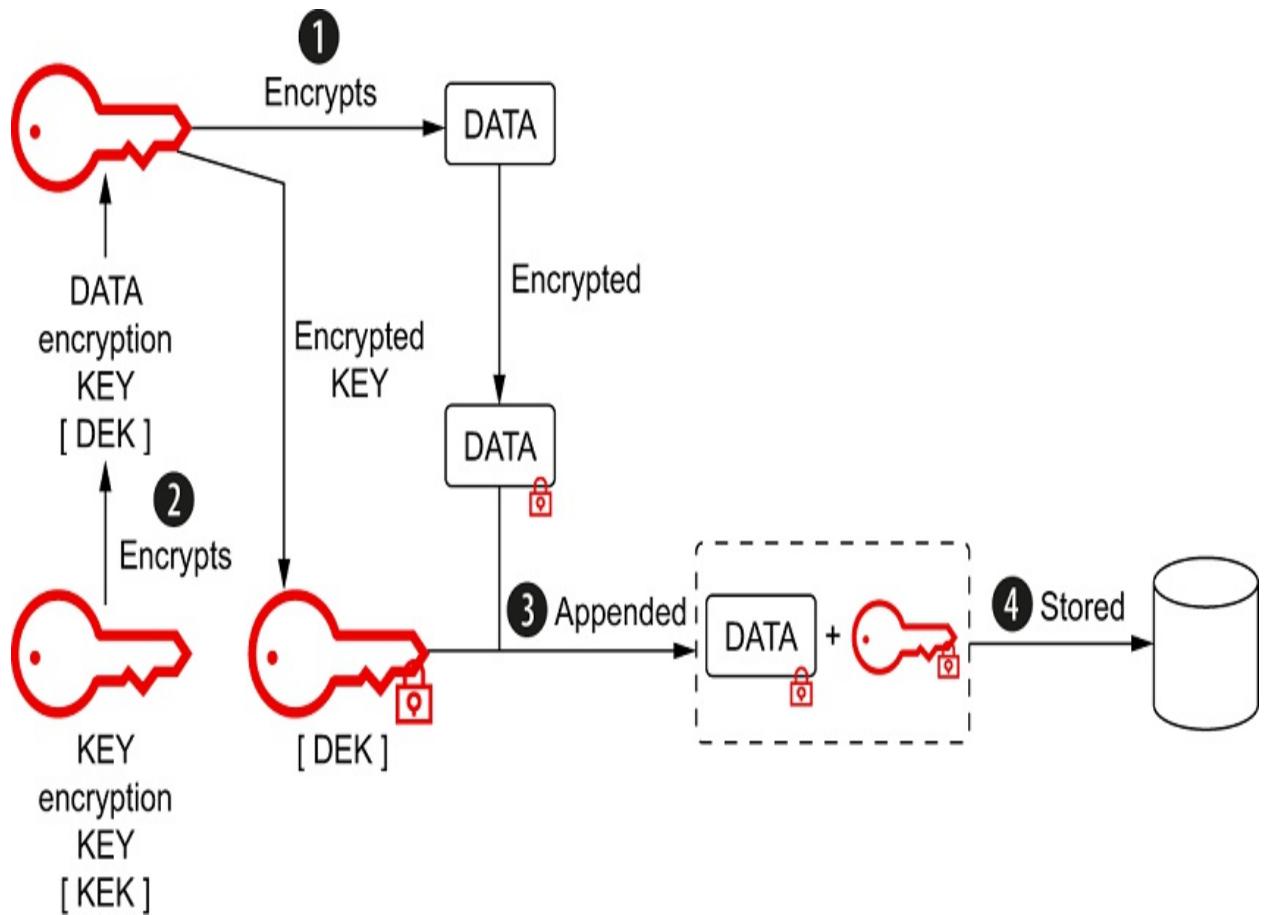
At this point, there are two chunks of encrypted bytes, data encrypted with *DEK* and *DEK* encrypted with *KEK*. Figure 4.8 shows both chunks and how they are encrypted.

Figure 4.8. Data is encrypted with DEK (Data Encryption Key) and DEK is encrypted with KEK (Key Encryption Key)



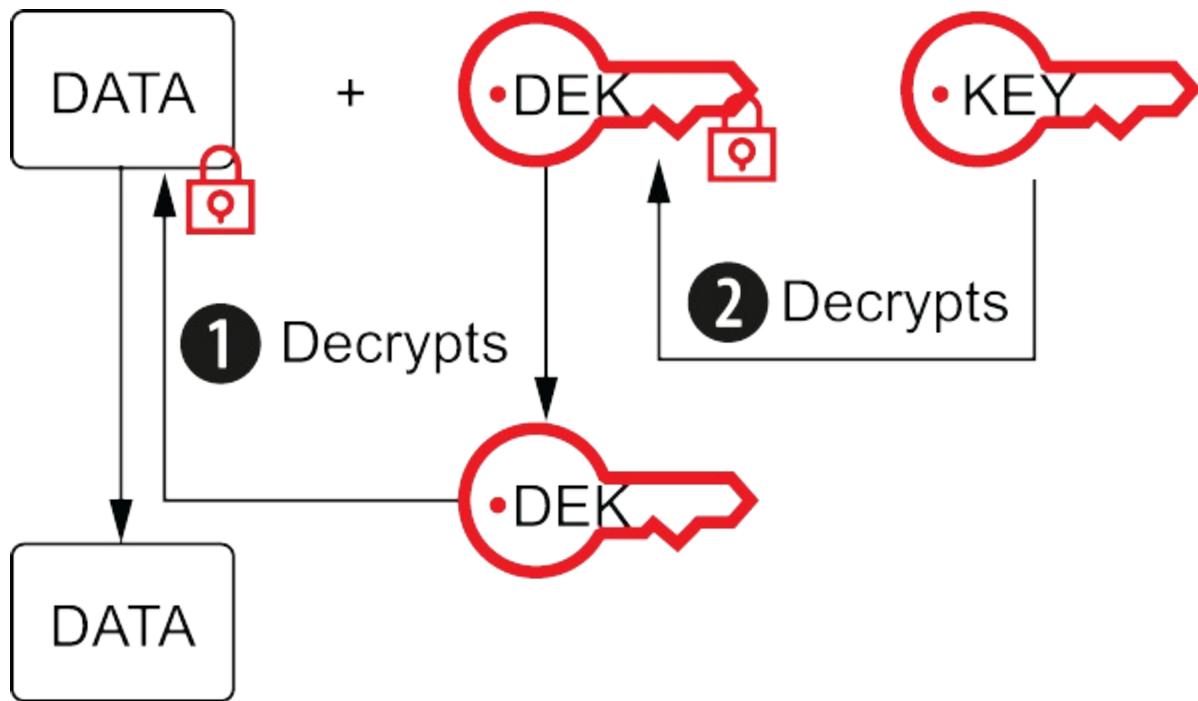
These two parts are appended together and stored as a single piece of data side by side. One of the big advantages of this approach is that each data has its own encryption key, if the data encryption key is compromised, for example by using brute force, the attacker would only able to decrypt that secret but not the rest of the secrets. Figure 4.9 shows the whole process.

Figure 4.9. Envelope encryption schema



To decrypt a secret, we need to do the reverse process. First of all, we split the data again with two chunks of data (encrypted secret and encrypted *DEK*): *DEK* is decrypted using *KEK*, and finally, the secret is decrypted using the decrypted *DEK* as shown in the following figure [4.10](#).

Figure 4.10. Decryping envelope encryption schema

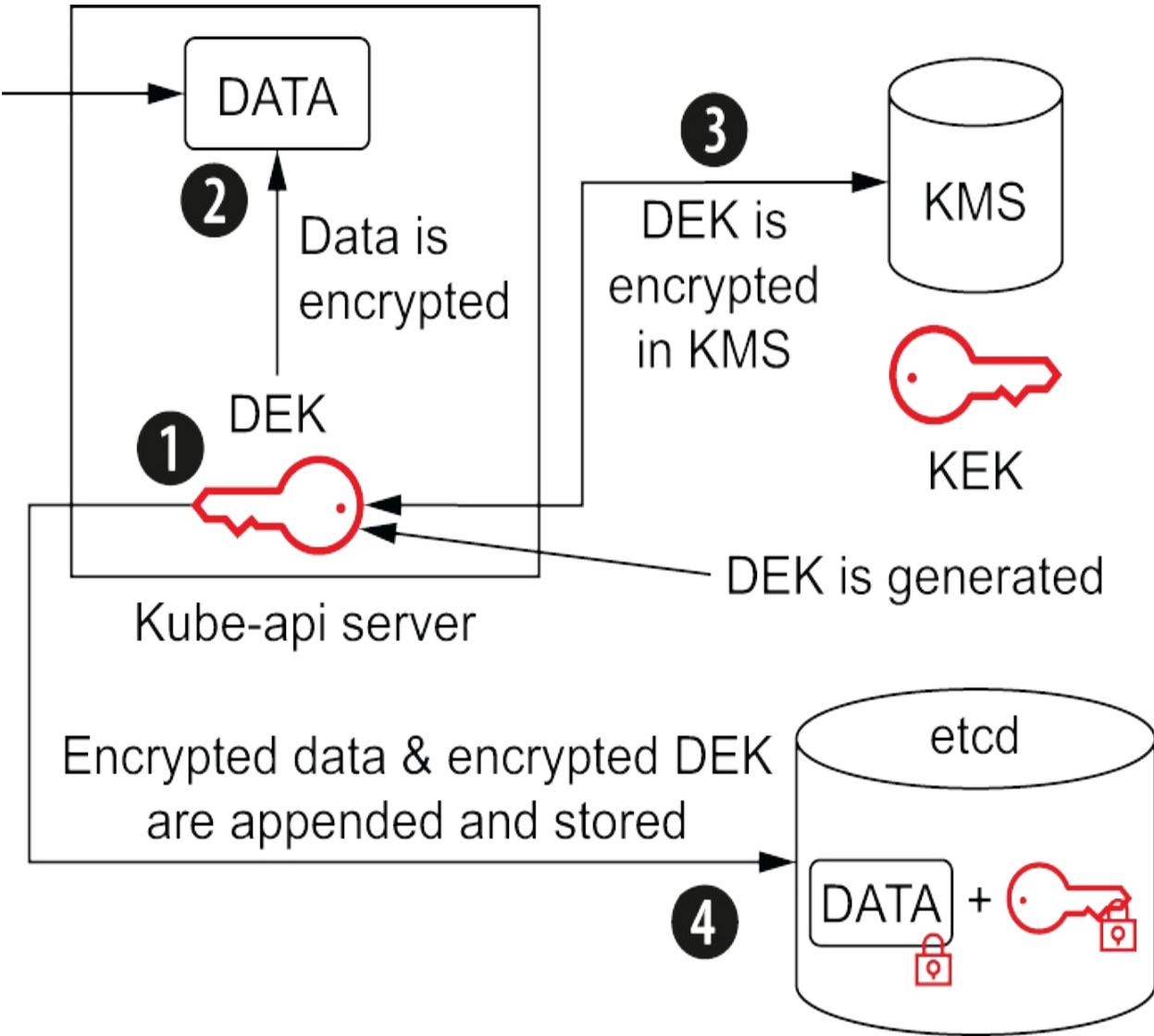


Kubernetes and Envelope Encryption

The Kubernetes KMS encryption provider uses envelope encryption in the following way:

Every time data needs to be encrypted, a new data encryption key (*DEK*) is generated using the *AES-CBC with PKCS7# padding* algorithm. Then the *DEK* is encrypted using a key encryption key (*KEK*) that is managed by the remote Key Management Server. Figure 4.11 shows this process.

Figure 4.11. Kubernetes and envelope encryption schema



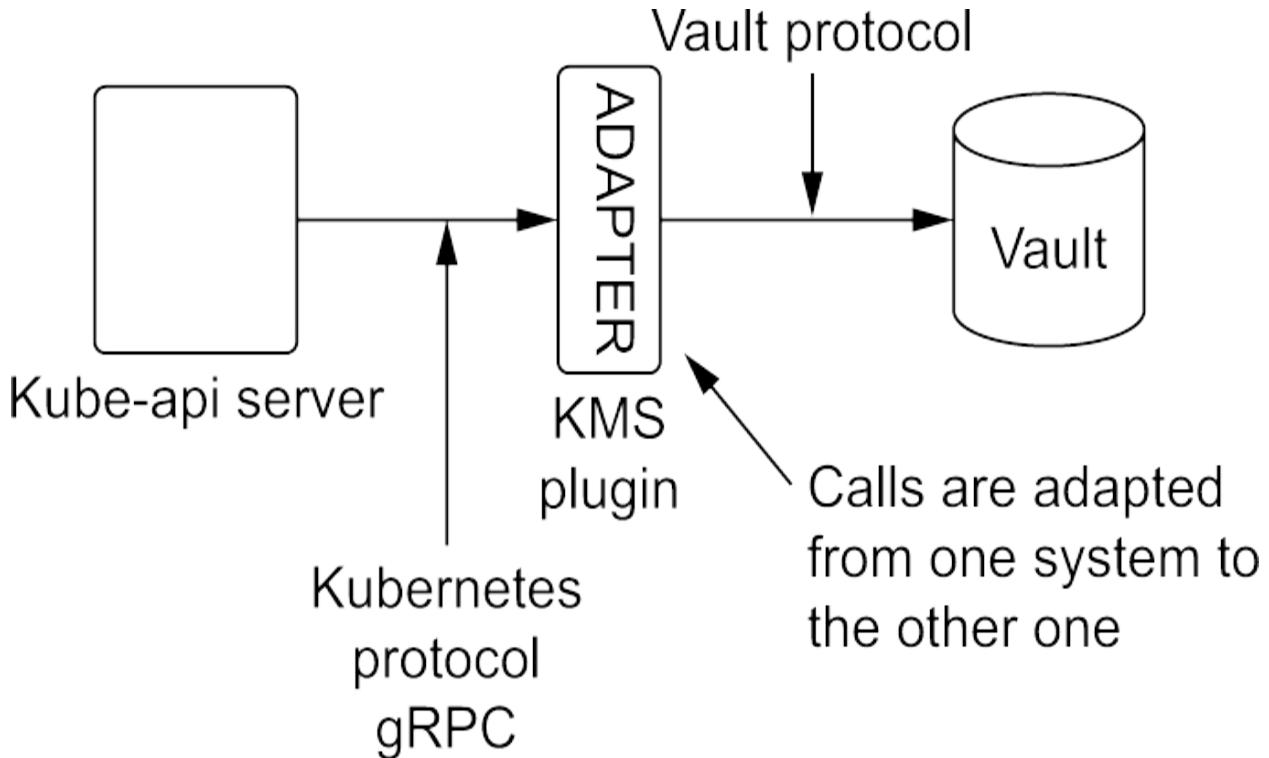
HashiCorp Vault as KMS

Using the KMS provider is the most secure way to encrypt/decrypt secrets, but we need a Key Management Server implementation that manages the key encryption keys and encrypts the data encryption keys.

But how the Kubernetes cluster communicates with the remote Key Management Server? In order to communicate with a remote server, the KMS provider uses *gRPC* protocol to communicate with a Kubernetes KMS plugin deployed in the Kubernetes master nodes. This plugin acts as a bridge between the kube-apiserver and the Key Management Service, adapting the encryption and decryption kube-apiserver flow to the protocol required by

the remote KMS.

Figure 4.12. KMS provider/plugin system



There are a lot of Kubernetes KMS plugins already supported out-of-the-box, to cite a few: IBM Key Protect, SmartKey, AWS KMS, Azure Key Vault, Google Cloud KMS, or HashiCorp Vault.

Since the book is written to be cloud provider agnostic, we are going to use HashiCorp Vault as remote KMS, but keep in mind that the process would be similar to the one explained here for any other KMS implementation.

Don't worry for now about what is HashiCorp Vault as we are going to explain deeply in the following chapter as it offers a lot of key features regarding secrets. But for this specific chapter, think about HashiCorp Vault as a deployable service that offers an endpoint to encrypt/decrypt data in-transit without storing it. All the key management occurs internally in Vault service. From the point of view of a user, the data is sent to the service and it is returned back ciphered or deciphered depending on the use case.



Tip

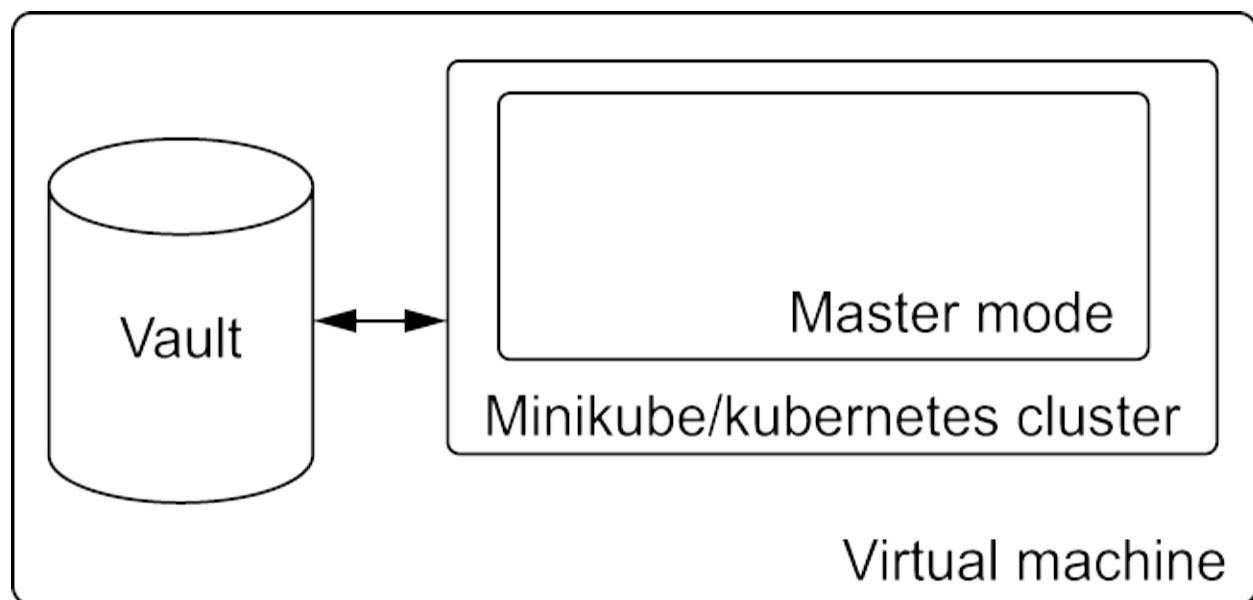
The KMS plugin system is designed to be extensible, so you could implement a new plugin for a specific KMS implementation. Usually, this should not be required as most of the KMS providers offers integration to Kubernetes, but keep in mind that nothing blocks you to implement a Kubernetes KMS plugin by yourself.

Let's start moving the keys from the master node to an external KMS.

Installing HashiCorp Vault

We have repeated over and over again that secrets and encryption keys should be deployed in different machines, this means that Key Management Server must run separately from Kubernetes master nodes. This is the way you should always proceed in a real scenario, but for this academic use case where we are using `minikube` running inside a Virtual Machine, and for the sake of simplicity, we are going to install HashiCorp Vault inside the Virtual Machine, but outside the Kubernetes cluster. Figure [4.13](#) illustrates the configuration.

Figure 4.13. HashiCorp Vault as KMS





Important

At the end of this section, we are providing you a single command that executes a script that automates all the steps explained in the following sections. Although the process is automated, we are going to explain to you the whole process so you can repeat it in any other environment.

To install HashiCorp Vault, we need to download and install it into the Virtual Machine, and register it as a service in systemd so it is started automatically every time we start the virtual machine. The steps are shown in listing [4.7](#).

Listing 4.7. Installation script

```
#1
curl -sfLo vault.zip https://releases.hashicorp.com/vault/1.1.2/vault.zip
unzip vault.zip
sudo mv vault /usr/bin/
sudo chmod +x /usr/bin/vault

#2
cat <<EOF | sudo tee /etc/profile.d/vault.sh
export VAULT_ADDR=http://127.0.0.1:8200
EOF
source /etc/profile.d/vault.sh

#3
sudo addgroup vault
sudo adduser -G vault -S -s /bin/false -D vault

#4
sudo mkdir -p /etc/vault/{config,data}

cat <<EOF | sudo tee /etc/vault/config/config.hcl
disable_mlock = "true"
backend "file" {
    path = "/etc/vault/data"
}
listener "tcp" {
    address      = "0.0.0.0:8200"
    tls_disable = "true"
}
EOF
```

```
sudo chown -R vault:vault /etc/vault

#5
cat <<"EOF" | sudo tee /etc/systemd/system/vault.service
[Unit]
Description="HashiCorp Vault - A tool for managing secrets"
Documentation=https://www.vaultproject.io/docs/
Requires=network-online.target
After=network-online.target
[Service]
User=vault
Group=vault
ExecStart=/usr/bin/vault server -config=/etc/vault/config
ExecReload=/bin/kill --signal HUP $MAINPID
ExecStartPost=-/bin/sh -c "/bin/sleep 5 && /bin/vault operator un
KillMode=process
KillSignal=SIGHUP
Restart=on-failure
RestartSec=5
TimeoutStopSec=30
StartLimitBurst=3

[Install]
WantedBy=multi-user.target
EOF

sudo systemctl start vault
```



Warning

For this example, listening address has been set to `0.0.0.0` so any host can access the Vault server. This is ok for non-production environments or educational purposes, but in real environments, configure it appropriately.

Configuring the Transit Secret Engine

Vault needs to be unsealed so it can be accessed externally and also enabling the transit secret engine so Vault can be used to encrypt/decrypt data in-transit. No worries if you still don't understand why these steps are required as we are going to be explained in more detail in the following chapter.

For this specific example, we are configuring Vault to be accessed by a user providing the `vault-kms-k8s-plugin-token` value as a token and creating an encryption key named `my-key`.

```
#1
vault operator init -format=json -key-shares=1 -key-threshold=1 |
vault operator unseal "$(cat /etc/vault/init.json | jq -r .unseal"

#2
vault login "$(cat /etc/vault/init.json | jq -r .root_token)"

#3
vault token create -id=vault-kms-k8s-plugin-token

#4
vault secrets enable transit

#5
vault write -f transit/keys/my-key
```

Installing Vault KMS provider

After having Vault up and running, we need to install and setup the Vault KMS provider/plugin. There are four important things to configure for the KMS provider:

1. The encryption key name (`my-key`).
2. The address where the Vault server is running (`127.0.0.1`).
3. The token that is required to access to Vault (`vault-kms-k8s-plugin-token`).
4. Setup the socket file for the Vault KMS provider (`/var/lib/minikube/certs/vault-k8s-kms-plugin.sock`).

Remember that the KMS provider is a *gRPC* server that acts as a bridge between `kube-apiserver` and the KMS. The steps are shown in listing [4.8](#).

Listing 4.8. Install kms vault script

```
#1
curl -sfLo vault-k8s-kms-plugin https://github.com/lordofthejars/
unzip vault-k8s-kms-plugin.zip
```

```

sudo mv vault-k8s-kms-plugin /bin/vault-k8s-kms-plugin
sudo chmod +x /bin/vault-k8s-kms-plugin

sudo mkdir -p /etc/vault-k8s-kms-plugin

#2
cat <<EOF | sudo tee /etc/vault-k8s-kms-plugin/config.yaml
keyNames:
- my-key
transitPath: /transit
addr: http://127.0.0.1:8200
token: vault-kms-k8s-plugin-token
EOF

sudo chown -R vault:vault /etc/vault-k8s-kms-plugin

#3
cat <<EOF | sudo tee /etc/systemd/system/vault-k8s-kms-plugin.service
[Unit]
Description="KMS transit plugin"
Requires=vault.service
After=vault.service
[Service]
User=root
Group=root
#4
ExecStart=/usr/bin/vault-k8s-kms-plugin -socketFile=/var/lib/minikube/kubelet.sock
ExecReload=/bin/kill --signal HUP $MAINPID
KillMode=process
KillSignal=SIGINT
Restart=on-failure
RestartSec=5
TimeoutStopSec=30
StartLimitBurst=3
[Install]
WantedBy=multi-user.target
EOF

sudo systemctl start vault-k8s-kms-plugin

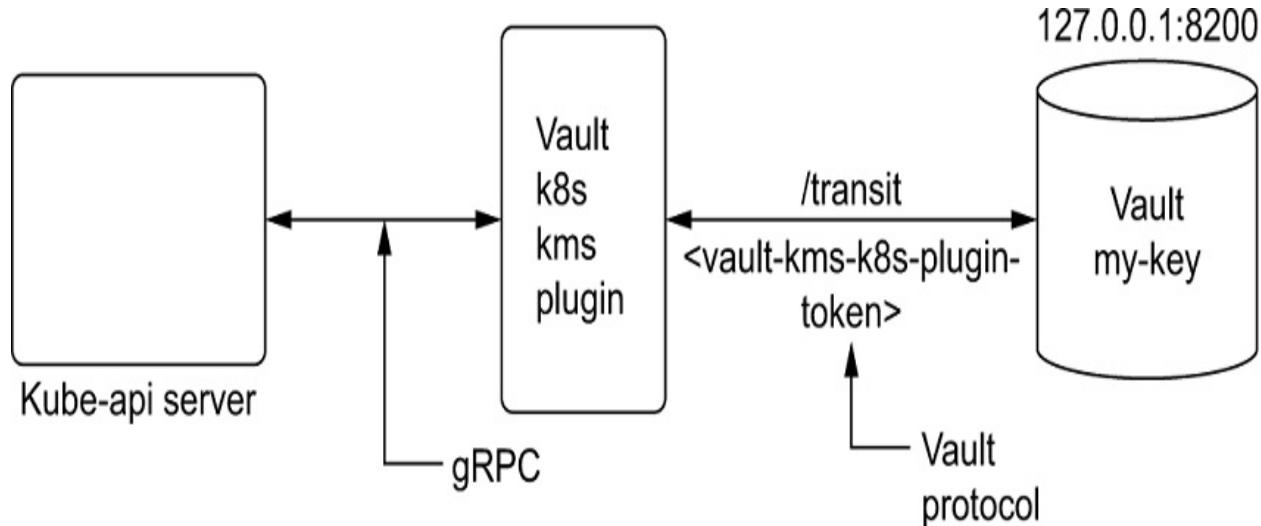
```

Configuring Kubernetes KMS Provider

If you remember in our previous example [Enabling Encryption Data at Rest](#), we created an `EncryptionConfiguration` file to enable encryption data at rest in the Kubernetes cluster. We now need to create an

EncryptionConfiguration file to configure the Vault KMS provider instead of aescbc provider. Figure 4.14 shows how Kubernetes-api server interacts with Kubernetes KMS plugin.

Figure 4.14. HashiCorp Vault as KMS

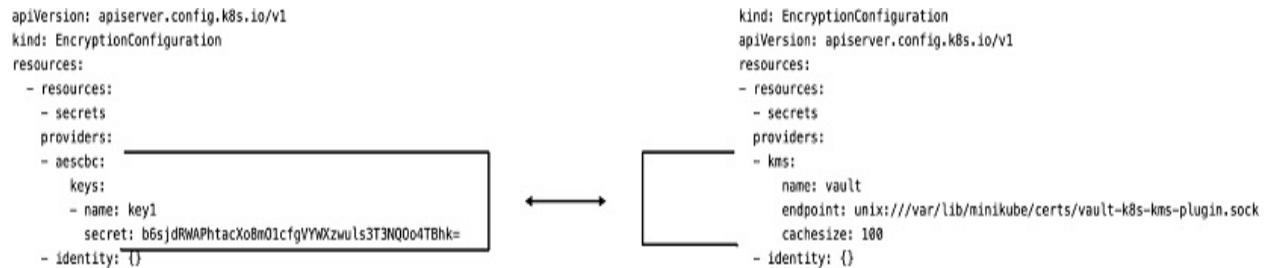


The important parameter to set is the endpoint because that's the location where the provider/plugin is listening. In this case, it was configured in the previous step with the socketFile parameter to `/var/lib/minikube/certs/vault-k8s-kms-plugin.sock`.

```
cat <<EOF | sudo tee /var/lib/minikube/certs/encryption-config.yaml
kind: EncryptionConfiguration
apiVersion: apiserver.config.k8s.io/v1
resources:
- resources:
  - secrets
providers:
- kms: #1
  name: vault
  endpoint: unix:///var/lib/minikube/certs/vault-k8s-kms-plugin.sock
  cachesize: 100
- identity: {}
EOF
```

Figure 4.15 shows the difference between EncryptionConfiguration file when KMS is used and when not.

Figure 4.15. EncryptionConfiguration vs KMS EncryptionConfiguration



Restart kube-apiserver

The last step is to restart the `kube-apiserver` so the new configuration takes effect and envelope encryption happens using Vault as a remote KMS. In the past, we restarted the whole `minikube` instance, but in this case, we are going to use a different approach by just restarting the `kubelet` process.

```
#1
sudo sed -i '/- kube-apiserver/ a \\ \\ \\ - --encryption-provider

#2
sudo systemctl daemon-reload
sudo systemctl stop kubelet
docker stop $(docker ps -aq)
sudo systemctl start kubelet
```

Putting everything together

As we noted previously, a script is provided to you to execute all the previous steps automatically.

Moreover, we suggest you now to use a new `minikube` instance, so you've got a clean `minikube` instance with Vault installed inside the virtual machine.

Let's create a `minikube` instance under the `vault` profile, SSH into the virtual machine where it is running the Kubernetes cluster, and run the script that executes all steps explained above. Execute the following commands:

```
#1
minikube stop
```

```
#2  
minikube start -p vault --memory=8192 --vm-driver=virtualbox --ku  
  
#3  
minikube ssh "$(curl https://raw.githubusercontent.com/lordofthej
```

4.2.2 Creating the Secret

In a terminal window, create a new secret named `kms-db-secret-encrypted` with two key/values.

```
kubectl create secret generic kms-db-secret-encrypted --from-literal
```

At this point, the secret is created in the same way as before, but the secret is encrypted using the envelope encryption schema. Let's explore how the data is stored inside etcd.

Accessing etcd

Let's repeat exactly the same process as we did in [Accessing etcd](#) to get the content of the secret `kms-db-secret-encrypted` and validate that it is stored encrypted instead of in the plain text.

In one terminal window, expose the etcd server in localhost:

```
kubectl port-forward -n kube-system etcd-vault 2379:2379
```

In another terminal, let's repeat the process of copying the etcd certificates and configure the `etcdctl` using environment variables:

```
kubectl cp kube-system/etcd-vault:/var/lib/minikube/certs/etcd/pe  
kubectl cp kube-system/etcd-vault:/var/lib/minikube/certs/etcd/pe  
  
export \  
    ETCDCTL_API=3 \  
    ETCDCTL_INSECURE_SKIP_TLS_VERIFY=true \  
    ETCDCTL_CERT=/tmp/peer.crt \  
    ETCDCTL_KEY=/tmp/peer.key
```

We can now query etcd to get the value of the `kms-db-secret-encrypted` key to validate that it is encrypted and impossible to decipher its values.

```
etcdctl get /registry/secrets/default/kms-db-secret-encrypted
```

The output should be similar as the one shown in listing [4.9](#).

Listing 4.9. Encrypted kms secret

```
/registry/secrets/default/kms-db-secret-encrypted  
cm?9>?*?-??c?0?~?6I?=?@?e??.??.??.??.?  
8Y  
t?p ?b? ?V?w?6?l?v?Ey?q.?
```

Unlike in the previous section, the secrets are encrypted in kube-apiserver using envelope encryption schema and then send to be stored in etcd server.

We can stop port-forwarding as we do not need anymore for now by aborting the process (push `ctrl+c` on the first terminal).

Also, we can stop the current minikube instance and start the default one that only contains a Kubernetes instance running:

```
minikube stop -p vault
```

```
minikube start
```

4.3 Summary

- Secrets are not encrypted in etcd by default hence we need to find a way to encrypt them to avoid any attacker with access to etcd can read them.
- `EncryptionConfiguration` Kubernetes object is the way to configure Kubernetes to encrypt resources (secrets), but if a remote *KMS* is not used both encrypted data and encryption keys are stored in the same machine.
- To have data and keys stored in different machines, Kubernetes supports the usage of a remote *KMS*.

5 HashiCorp Vault and Kubernetes

This chapter covers

- Enabling HashiCorp Vault for use by end user applications deployed to Kubernetes
- Integrating Kubernetes authentication to simplify access to Vault resources
- Accessing secrets stored in HashiCorp Vault by applications deployed to Kubernetes

Chapter 4 introduced HashiCorp Vault as a KMS that could be used to provide encryption for secrets and other resources stored in etcd, the key/value datastore for Kubernetes, so that these values could not be readily accessed as they are stored at rest.

This chapter focuses on the importance of using a secrets management tool, like HashiCorp Vault, to securely store and manage sensitive assets for applications deployed to Kubernetes as well as demonstrating how both applications and Vault can be configured to provide seamless integration with one another. By using a tool like Vault, application teams can offload some of the responsibilities involved when managing sensitive resources to a purpose built tool, while still being able to integrate with their applications.

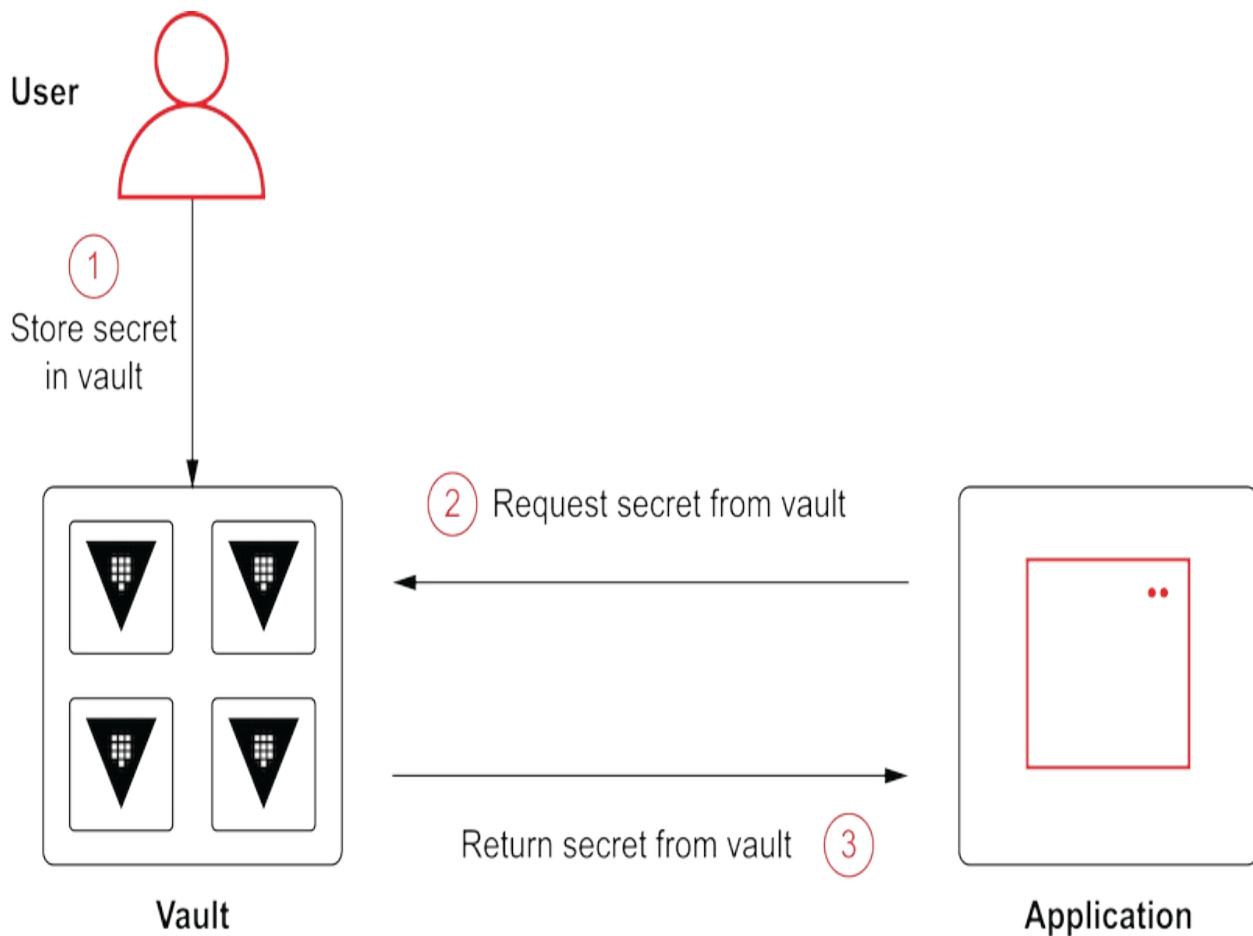
5.1 Managing Application Secrets Using HashiCorp Vault

As we have seen thus far, sensitive assets can be used either by the core infrastructure components of Kubernetes or by applications deployed to the platform. Chapter 4 focused primarily on the infrastructure portion, and how to properly secure the primary database of Kubernetes, etcd, by encrypting the values using HashiCorp Vault. While Vault can aid in keeping the platform secure, it is more commonly used to store and protect properties for use in applications.

When securing the values for storage within etcd as described in Chapter 4, Vault was used as an intermediary to perform the cryptographic functions needed to encrypt and decrypt *Data in transit* using the *transit Secrets Engine* of Vault. For applications deployed to Kubernetes, they themselves are not designed to act as a secrets store and would look to another tool that is better designed for this purpose. This is where Vault can be used as a solution.

Let's imagine that we wanted to use Vault to store the secret locations of where field agents are deployed around the world. While a database could be used in this scenario, we can demonstrate how to store arbitrary data in a secure fashion using a Vault instance deployed to Kubernetes and access the values within an application that is deployed to Kubernetes.

Figure 5.1. Secrets storage within Vault and an application requesting access to retrieve the stored values



5.1.1 Deploying Vault to Kubernetes

In chapter 4, HashiCorp Vault was installed within the *minikube* Virtual Machine to provide a clear separation between the Kubernetes infrastructure components (particularly *etcd*) and the KMS. Since *etcd* is so crucial to the functionality of the Kubernetes cluster, you would want to ensure that there were no dependencies upon one another for proper operation (also known as the "Chicken or the egg" dilemma).

Since there is less of a hard dependency between applications and the KMS (and vice versa), and to be able to take advantage of many of the benefits of Kubernetes itself (such as scheduling and application health monitoring), Vault will be deployed to Kubernetes for the purpose of acting as a KMS for applications.

There are several methods that Vault can be deployed to Kubernetes, but the most straightforward is to use *Helm*. A chart is available from Hashicorp (<https://github.com/hashicorp/vault-helm>) and supports the majority of deployment options that would be needed for either a development instance or to support a production ready cluster.

To get started, use the terminal from any directory to first add the Hashicorp repository to Helm containing the Vault chart:

```
helm repo add hashicorp https://helm.releases.hashicorp.com  
"hashicorp" has been added to your
```

Then, retrieve the latest updates from the remote repositories which will pull down the content to your machine:

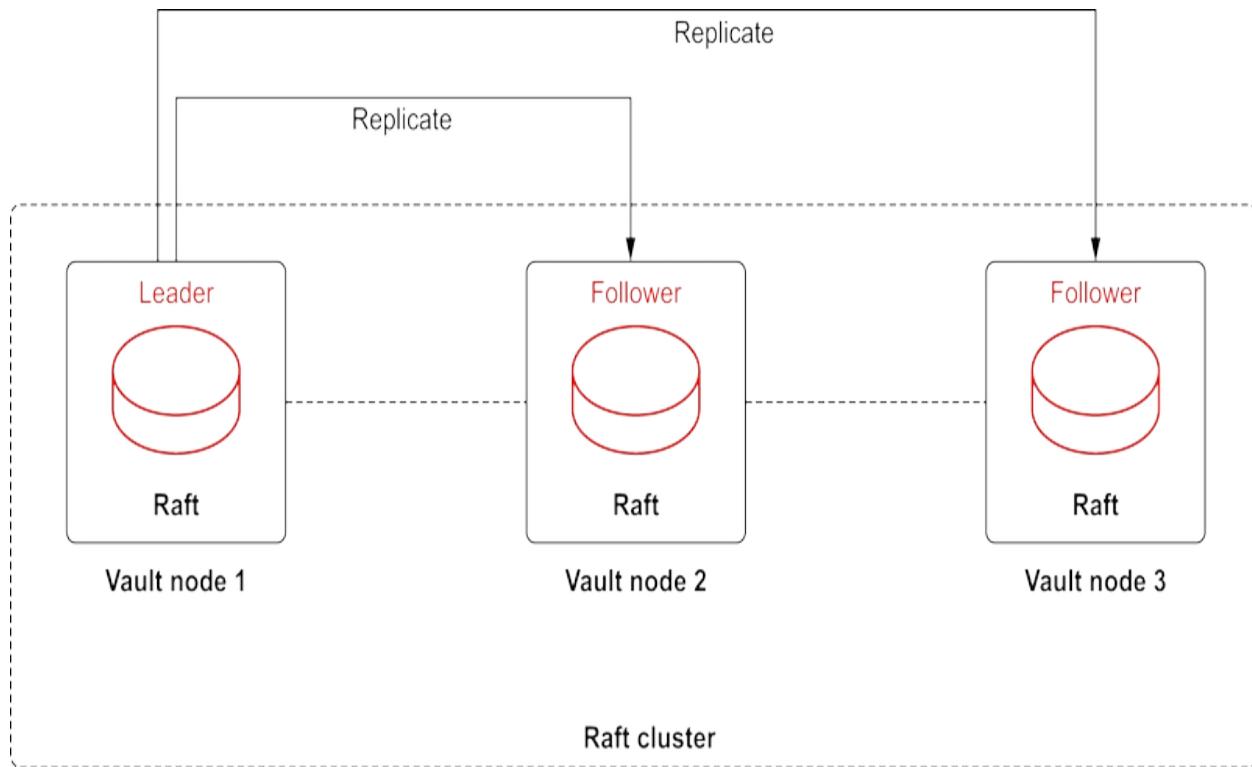
```
helm repo update  
Hang tight while we grab the latest from your chart repositories.  
...Successfully got an update from the "hashicorp" chart reposito  
Update Complete. ✨ Happy Helming! ✨
```

Vault HA Storage Using Integrated Raft

The next consideration that must be considered whenever Vault is deployed is how we will handle the storage of its metadata. When Vault was deployed outside of Kubernetes within the *minikube* virtual machine, the *filesystem* type of storage was used. This configuration will persist to the local filesystem and is an easy way to get Vault running. However, when thinking about a production style deployment, the *filesystem* type of storage would not be desirable as it only supports a single instance.

Another benefit of deploying Vault on Kubernetes aside from the aforementioned reasons and to align with the desire for a more production style deployment is the ability to achieve High Availability (HA) easily. However, going beyond a single instance requires the use of a different storage type other than *filesystem*. While Vault does support a number external storage backends ranging from relational and NoSQL databases to cloud object storage, a simple option that is highly available and does not have any external dependency is integrated storage using Raft.

Figure 5.2. Leader election and the replication of data using Raft storage



- 1. Cluster members select a leader via a leader election
- 2. Each vault instance communicates to each raft instance
- 3. Storage occurs on leader instance
- 4. Data replicated from leader instance to follower instances

Raft is a distributed *Consensus Algorithm* where multiple members form a cluster after an election occurs. A leader is determined based out the outcome of the election where they have the responsibility of determining the shared state of the cluster and replicate the state to each of the followers. The minimum number in any cluster is 3 ($N/2+1$ where N is the number of total nodes) as to ensure there is a minimum number of nodes if a failure occurs. Raft is a common protocol and is used by other solutions in the cloud native space including etcd when it operates in a highly available configuration.

With an understanding of the storage requirement's needed to support a highly available deployment of Vault, install the Vault Helm chart by setting the `server.ha.enabled=true` and `server.ha.raft.enabled=true` values which will enable HA along with enabling Raft. In addition, since we are attempting to deploy a highly available deployment to a single *minikube* node, set the `server.affinity` value to an "" which will skip the default pod affinity configurations which would attempt to schedule each vault pod to a

separate node and thus result in a scheduling failure.

Execute the following command to install the chart:

Listing 5.1. Deploying vault using Helm

```
helm upgrade -i vault hashicorp/vault \
  --set='server.ha.enabled=true' \
  --set='server.ha.raft.enabled=true' \
  --set='server.affinity=""' -n vault \
  --create-namespace
```

Confirm that three (3) instances Vault deployed to the vault namespace:

```
kubectl get pods -n vault
```

NAME	READY	STATUS	RESTART
vault-0	0/1	Running	0
vault-1	0/1	Running	0
vault-2	0/1	Running	0
vault-agent-injector-76d54d8b45-dvzww	1/1	Running	0

In addition to the the expected three vault instances, another pod prefixed with vault-agent-injector is also deployed and serves the purpose of dynamically inject secrets into pods. The Vault Agent Injector will be covered in detail later in the chapter.

If any of the Vault or Vault Injector pods are not currently in a *Running* state or none of the pods appear, the kubectl get events -n vault command to investigate the cause of such issues.

With vault deployed, the first step is to initialize vault. Since there are multiple instances of vault, one of them should be designated as the initial leader; vault-0 in this instance.

First, check the status of vault which should denote that it has yet to be initialized

```
kubectl -n vault exec -it vault-0 -- vault operator init -status
Vault is not initialized
```

Now, initialize the Vault so that you can begin interacting with it:

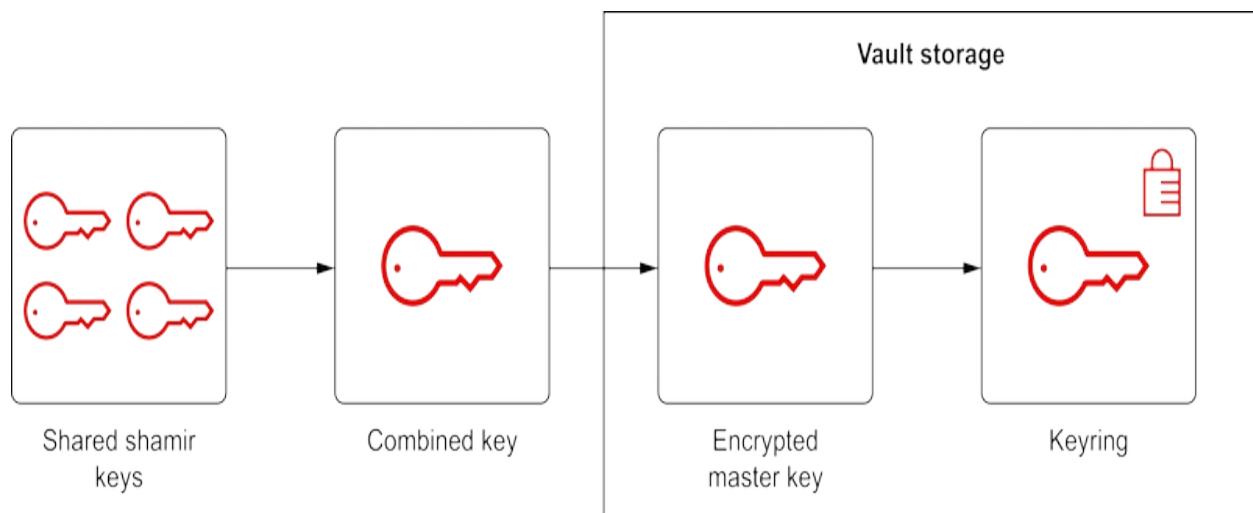
Listing 5.2. Initializing the vault

```
kubectl -n vault exec -it vault-0 -- vault operator init #1
```

When Vault is initialized, it will be put into *sealed* mode, meaning that it knows how to access the storage layer, but cannot decrypt any of the content. When Vault is in a sealed state, it is akin to a bank vault where the assets are secure, but no actions can take place. To be able to interact with Vault, it must be unsealed.

Unsealing is the process of obtaining access to the *master key*. However, this key is only one portion of how data in Vault is encrypted. When data is stored in Vault, it is encrypted using an *encryption key*. This key is stored along with the data in Vault and encrypted using another key, known as the *master key*. However, it does not stop there. The *master key* is also stored with the data in Vault and encrypted one more time by the *unseal key*. The *unseal key* is distributed into multiple shards using an algorithm known as Shamir's Secret Sharing. To reconstruct the *seal key*, a certain number of shards must be provided individually one at a time, which will enable access to the combined key, and ultimately enable access to the data stored within Vault.

Figure 5.3. The steps involved to unseal Vault



When vault was initialized, five (5) unsealed keys were provided representing the shards that will be used to construct the combined key. By default, three (3) keys are needed to reconstruct the combined key. Let's begin the process of unsealing the Vault for the vault-0 instance.

Execute the following command to begin the unsealing process. When prompted, enter the value of the key next to *Unseal Key 1* at the prompt from the vault operator `init` command executed in listing [5.2](#). :

```
kubectl -n vault exec -it vault-0 -- vault operator unseal
```

Unseal Key (will be hidden):

Listing 5.3. Progression of unsealing the vault

Key	Value
---	-----
Seal Type	shamir
Initialized	true
Sealed	true
Total Shares	5
Threshold	3
Unseal Progress	1/3 #1
Unseal Nonce	fbb3714f-27cb-e362-ba40-03db093aea23
Version	1.7.2
Storage Type	raft
HA Enabled	true

The unsealing process is underway as the *Unseal Progress* row as shown in listing [5.3](#) indicates that one of the keys has been entered. Execute the the same `kubectl -n vault exec -it vault-0--vault operator unseal` two more times and providing the values next to *Unseal Key 2* and *Unseal Key 3* and so on when prompted.

Keep providing keys until you see an output similar to listing [5.4](#):

Listing 5.4. An unsealed Vault instance

Key	Value
---	-----
Seal Type	shamir
Initialized	true

Sealed	false #1
Total Shares	5
Threshold	3
Version	1.7.2
Storage Type	raft
Cluster Name	vault-cluster-c5e28066 #2
Cluster ID	0a3170f4-b486-7358-cba7-381349056f3e
HA Enabled	true #3
HA Cluster	n/a
HA Mode	standby
Active Node Address	<none>
Raft Committed Index	24

The `vault-0` instance is now unsealed. However, since Vault is running in HA mode, the other two members must be joined to the newly created cluster and undergo the unsealing process.

First, join the `vault-1` instance to the Raft cluster:

Listing 5.5. Joining a new node to the Raft cluster

```
kubectl -n vault exec -ti vault-1 -- vault operator \
    raft join http://vault-0.vault-internal:8200 #1
```

Key	Value
---	-----
Joined	true

Next, execute the following command three times providing a different unseal key as accomplished with `vault-0`:

```
kubectl -n vault exec -it vault-1 -- vault operator unseal
```

Unseal Key (will be hidden):

Once these steps have been completed on the `vault-1` instance, perform the `vault operator join` and `vault operator unseal` commands on the `vault-2` instance.

To confirm that the highly available Vault cluster is ready, login to the `vault-0` instance using the Initial Root Token provided by the `vault operator init` command in listing [5.2](#):

```
kubectl -n vault exec -it vault-0 -- vault login
```

Token (will be hidden):

Success! You are now authenticated. The token information display is already stored in the token helper. You do NOT need to run "va again. Future Vault requests will automatically use this token.

Key	Value
--	-----
token	s.cm8HyaIxR2MPseDxTv0U7ugD
token_accessor	Fx23YMxqiYabU8u5Ptt2qpCH
token_duration	∞
token_renewable	false
token_policies	["root"]
identity_policies	[]
policies	["root"]

Now that you have logged in with the *root* token, confirm all members have successfully joined to the vault cluster by listing all Raft members:

```
kubectl -n vault exec -ti vault-0 -- vault operator raft list-peer
```

Node	Address
--	-----
8b58cb62-7da7-5e8d-298b-95d4e8203ea5	vault-0.vault-internal:8201
4cef7124-d40a-bba8-4dac-4830936ceea3	vault-1.vault-internal:8201
d1a8d5fd-82be-feb7-a3e7-ca298d81d050	vault-2.vault-internal:8201

Notice how *vault-0* is listed as the *leader* with the other two members as the *follower* as indicated by the *State* column. If three Raft instances do not appear, confirm that each member was successfully joined in listing [5.5](#).

At this point, the highly available Vault has been deployed to the *minikube* instance.

Secrets Engine

We spoke earlier about the types of storage backends that can be used as durable storage for vault information. The actual storage, generation and encryption of data within Vault itself is facilitated by one of the supported *Secrets Engines*. Secrets Engines can perform simple operations, like storing or reading data. However, more complex secrets engines may call out to

external resources to generate assets on demand. The *transit secrets engine*, as described in chapter 4, aided in the encryption and decryption of values stored in etcd, but for applications looking to retrieve values stored in Vault like the one that we are looking to implement in this chapter, the *Key/Value*, or kv secrets engine can be used.

The kv engine does just like it sounds like it does. Enables the storage of key/value pairs within Vault. This secrets engine has evolved over time and there are as of this writing, two versions that can be used.

- **KV Version 1**

Non versioned storage of key/value pairs. Updated values overwrite existing values. Smaller storage footprint since there is no requirement to store additional metadata which supports versioning.

- **KV Version 2**

Support for versioned key/value pairs. Enhanced support for avoiding unintentionally overwriting data. Additional storage requirement for the metadata used to track versioning.

Either KV Version 1 or 2 could be used in this case, but given that most implementations that make use of Version 2, it is the one that will be used.

Secrets Engines are enabled on a given *Path* or location within vault. For our application, we will use the agents path.

Before enabling the kv engine, first list all of the enabled secrets engines:

```
kubectl -n vault exec -it vault-0 -- vault secrets list
```

Path	Type	Accessor	Description
---	---	-----	-----
cubbyhole/	cubbyhole	cubbyhole_4cc71c5d	per-token private secret
identity/	identity	identity_9f9aa91a	identity store
sys/	system	system_8f066be3	system endpoints used for policy and

Now, enable the kv-v2 Secrets Engine on the agents path to enable the storing of key/value pair secrets:

Listing 5.6. Enabling the kv secrets engine

```
kubectl -n vault exec -it vault-0 -- vault secrets enable \
-path=agents \ #1
-version=2 \ #2
kv-v2 #3
```

Success! Enabled the kv-v2 secrets engine at: agents/

With the engine enabled on the agents path, let's store a few values. Each agent will have several attributes associated with their key:

- name
- email
- location

Create a new entry for agent bill

```
kubectl -n vault exec -it vault-0 -- vault kv put agents/bill \
name="Bill Smith" \
email="bill@acme.org" \
location="New York, USA"
```

Key	Value
--	-----
created_time	2021-05-29T17:20:48.24905171Z
deletion_time	n/a
destroyed	false
version	1

Retrieving the value stored in Vault is just as easy

```
kubectl -n vault exec -it vault-0 -- vault kv get agents/bill

===== Metadata =====


| Key           | Value                         |
|---------------|-------------------------------|
| --            | -----                         |
| created_time  | 2021-05-29T17:20:48.24905171Z |
| deletion_time | n/a                           |
| destroyed     | false                         |
| version       | 1                             |



===== Data =====


| Key      | Value         |
|----------|---------------|
| --       | -----         |
| email    | bill@acme.org |
| location | New York, USA |


```

```
name      Bill Smith
```

Now that we have confirmed that we were able to add and retrieve a secret successfully, add a few more agents:

```
kubectl -n vault exec -it vault-0 -- vault kv put agents/jane \
  name="Jane Doe" \
  email="jane@acme.org" \
  location="London, United Kingdom"
kubectl -n vault exec -it vault-0 -- vault kv put agents/maria \
  name="Maria Hernandez" \
  email="maria@acme.org" \
  location="Mexico City, Mexico"
kubectl -n vault exec -it vault-0 -- vault kv put agents/james \
  name="James Johnson" \
  email="james@acme.org" \
  location="Tokyo, Japan"
```

At this point, there should be four (4) secrets stored in vault.

Application Access and Security

The root token that is currently being used to interact with Vault has unrestricted access to all of Vault's capabilities and is not recommended to be used within applications interacting with the Vault server. Instead, applying the Principle of Least Privilege, a separate method for the application to authenticate against Vault should be used. Vault supports multiple *Auth Methods* for a consumer to identify itself as including *Username & Password*, *TLS Certificates*, and the aforementioned *Token* just to name a few.

An *Auth Method* can be associated with one or more *Policies* that define the privileges against different paths in Vault. Each policy is associated with *Capabilities* which provide fine-grained control against a particular path.

Figure 5.4. The Relationships between Capabilities, Policies, and Auth Methods



The following are the set of capabilities available in Vault:

- **create**
Allows creating data against a given path
- **read**
Allows reading the data at a given path
- **delete**
Allows deletion of against a given path
- **list**
Allows the listing of values against a given path

Policies are written either in JSON or HCL (HashiCorp Configuration Language [compatible with JSON]) format and are submitted to the Vault server using the CLI at creation time.

Create a new file called *agents-policy.hcl* that defines the Policy that provides access to *read* and *list* the values within the *agents* path

Listing 5.7. Creating a policy to govern access to vault content

```

path "agents/data/*" {
    capabilities = ["list", "read"] #1
}

path "agents/metadata/*" {
    capabilities = ["list", "read"]
}

```



Important

Versioned content, as in kv version 2 stores content prefixed with data/ which was omitted in kv version 1 and must be accounted for when designing policies.

Create a new policy called agents_reader by copying the .hcl policy file that was created in listing [5.7](#) to the vault pod and creating the policy

```
cat agents-policy.hcl | \
  kubectl -n vault exec -it vault-0 -- vault policy write agents_
```

```
Unable to use a TTY - input is not a terminal or the right kind o
Success! Uploaded policy: agents_reader
```

Now, create a new token that can be used by the *agents* application and provides only access to the agents path and assign it to a variable called AGENTS_APP_TOKEN:

Listing 5.8. Creating a token for the agents application

```
export AGENTS_APP_TOKEN=$(kubectl -n vault exec -it vault-0 \
  -- vault token create -policy=agents_reader -format=yaml | \ #1
  grep client_token | awk '{ print $2 }') #2
```

You can view information about the token by looking up the details of the token:

```
kubectl -n vault exec -it vault-0 -- vault token lookup $AGENTS_A
```

Key	Value
--	-----
accessor	9N8JDsdVrGfspYILad3DxZUq
creation_time	1622320692
creation_ttl	768h
display_name	token
entity_id	n/a
expire_time	2021-06-30T20:38:12.973925141Z
explicit_max_ttl	0s
id	s.1ErBLPR3QTrNks8HhLNQcpjv #1
issue_time	2021-05-29T20:38:12.973933315Z
meta	<nil>
num_uses	0
orphan	false
path	auth/token/create

```
policies          [agents_reader default]
renewable         true
ttl               767h51m20s
type
```

If the command returned an error or no values at all, confirm the value of the \$AGENTS_APP_TOKEN value that was assigned in listing [5.8](#).

By default, the Time to Live (TTL), or the period of time in which the token is valid, is set for 32 days. In many enterprise organizations, limiting the length in time for which a token is valid increases the security posture and reduces the threat vector in the event a token becomes compromised. To explicitly set the TTL of a token, the -ttl flag can be added to the vault token create command to customize the duration a token is valid.

Confirm that the token can only view resources in the agents path as described by the agents_reader policy. First, back up the current root token to another location within the vault pod so that we can restore access later on.

```
kubectl -n vault exec -it vault-0 -- \
cp /home/vault/.vault-token \
/home/vault/.vault-token.root
```

Now, login with the new token created in listing [5.8](#):

```
echo $AGENTS_APP_TOKEN | kubectl -n vault exec -it vault-0 -- vau
```

Unable to use a TTY - input is not a terminal or the right kind of Success! You are now authenticated. The token information displayed is already stored in the token helper. You do NOT need to run "vault token renew" again. Future Vault requests will automatically use this token.

Key	Value
--	---
token	s.1ErBLPR3QTrNks8HhLNQcpjv
token_accessor	9N8JDsdVrGfspYILad3DxZUq
token_duration	767h26m48s
token_renewable	true
token_policies	["agents_reader" "default"]
identity_policies	[]
policies	["agents_reader" "default"]

Logging in display key information about traits associated with the token

including the time that it is valid and the permissions that it contains.
Confirm that the keys within the agents path can be listed

```
kubectl -n vault exec -it vault-0 -- vault kv list /agents
```

Keys

```
--  
bill  
james  
jane  
maria
```

If an error occurs, confirm that the policy was created properly starting with listing [5.7](#).

If the keys were listed properly, next attempt to access a resource, such as listing enabled secrets engines, for which the token should not have access

```
kubectl -n vault exec -it vault-0 -- vault secrets list
```

```
Error listing secrets engines: Error making API request.
```

```
URL: GET http://127.0.0.1:8200/v1/sys-mounts  
Code: 403. Errors:
```

```
* 1 error occurred:  
  * permission denied
```

With the restricted level of permissions confirmed, restore the root token by copying the backup token to the default location vault expects so we can once again execute elevated permissions:

```
kubectl -n vault exec -it vault-0 -- \  
 cp /home/vault/.vault-token.root \  
 /home/vault/.vault-token
```

Confirm an elevated request can be executed so that subsequent steps can be implemented

```
kubectl -n vault exec -it vault-0 -- vault secrets list
```

5.1.2 Deploying an Application to Access Vault

With a token that can be used to access the agents path within Vault, let's deploy an application that demonstrates accessing Values within vault. The source code is available at <https://github.com/lordofthejars/kubernetes-secrets-source/tree/master/agentsx>, but the application itself from a programming perspective will not be described in this chapter.

The first step is to create a Kubernetes Secret called agents-vault-token containing our token which will be injected as an environment variable within the application so that it can communicate with Vault.

```
kubectl create secret generic agents-vault-token \
--from-literal=token=$(echo -n $AGENTS_APP_TOKEN | tr -d '\r\n')

secret/agents-vault-token created
```

Create a file called serviceaccount.yml to define a Kubernetes service account called agents to associate with the application. It is always recommended that each workload is executed under a separate service account in order to delegate only the necessary permissions required. This will be demonstrated in further detail in a subsequent section.

Listing 5.9. serviceaccount.yml

```
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: agents
```

Create the agents service account referencing the manifest created in listing [5.9](#)

```
kubectl -n vault apply -f serviceaccount.yml

serviceaccount/agents created
```

Create a file called deployment_token_auth.yml containing the manifest for the application that will use token based authentication using the token defined within the agents-vault-token secret created earlier.

Listing 5.10. deployment_token_auth.yml

```

---  

apiVersion: apps/v1  

kind: Deployment  

metadata:  

  name: agents  

spec:  

  replicas: 1  

  selector:  

    matchLabels:  

      app: agents  

  strategy:  

    rollingUpdate:  

      maxSurge: 25%  

      maxUnavailable: 25%  

    type: RollingUpdate  

  template:  

    metadata:  

      labels:  

        app: agents  

  spec:  

    containers:  

      - env:  

          - name: QUARKUS_VAULT_AUTHENTICATION_CLIENT_TOKEN  

            valueFrom:  

              secretKeyRef: #1  

              key: token  

              name: agents-vault-token  

        image: quay.io/ablock/agents  

        imagePullPolicy: Always  

        name: agents  

        ports:  

          - containerPort: 8080  

            protocol: TCP  

    restartPolicy: Always  

    serviceAccountName: agents

```

Apply the Deployment to the cluster to create the application

```
kubectl -n vault apply -f deployment_token_auth.yml  
deployment.apps/agents
```

With the deployment created, confirm the application is running:

```
kubectl -n vault get pods -l=app=agents
```

NAME	READY	STATUS	RESTARTS	AGE
------	-------	--------	----------	-----

```
agents-595b85fc6-6fdbj    1/1      Running     0          66s
```

The Vault token is exposed to the application via the `QUARKUS_VAULT_AUTHENTICATION_CLIENT_TOKEN` environment variable. The application framework then facilitates the backend communication to the vault server.

Let's test out the application to confirm that values can be received.

The application exposes a restful service at `/agents` endpoint on port 8080 that can be used to query for agents stored in Vault.

Attempt to locate the record for the agent `bill` by first locating the name of the running `agents` pod and then invoking the service at the `/agents/bill` endpoint.

```
AGENTS_POD=$(kubectl get pods -l=app=agents \
-o jsonpath={.items[0].metadata.name})
kubectl -n vault exec -it $AGENTS_POD -- \
curl http://localhost:8080/agents/bill
{"email":"bill@acme.org","location":"New York, USA","name":"Bill"
```

Since we were able to retrieve a valid result, now attempt to retrieve a non-existent value which should return an empty result

```
kubectl -n vault exec -it $AGENTS_POD -- \
curl http://localhost:8080/agents/bad
```

Feel free to query for the other agents that were stored in Vault to fully exercise the application.

In the next section, we will explore how to avoid using Vault tokens to authenticate to Vault.

5.2 Kubernetes Auth Method

The [5.1.2](#) section explored how an application can make use of Vault tokens using the Token Auth Method to interact with the Vault server to access stored values. While using tokens to access vault is fairly straightforward, it

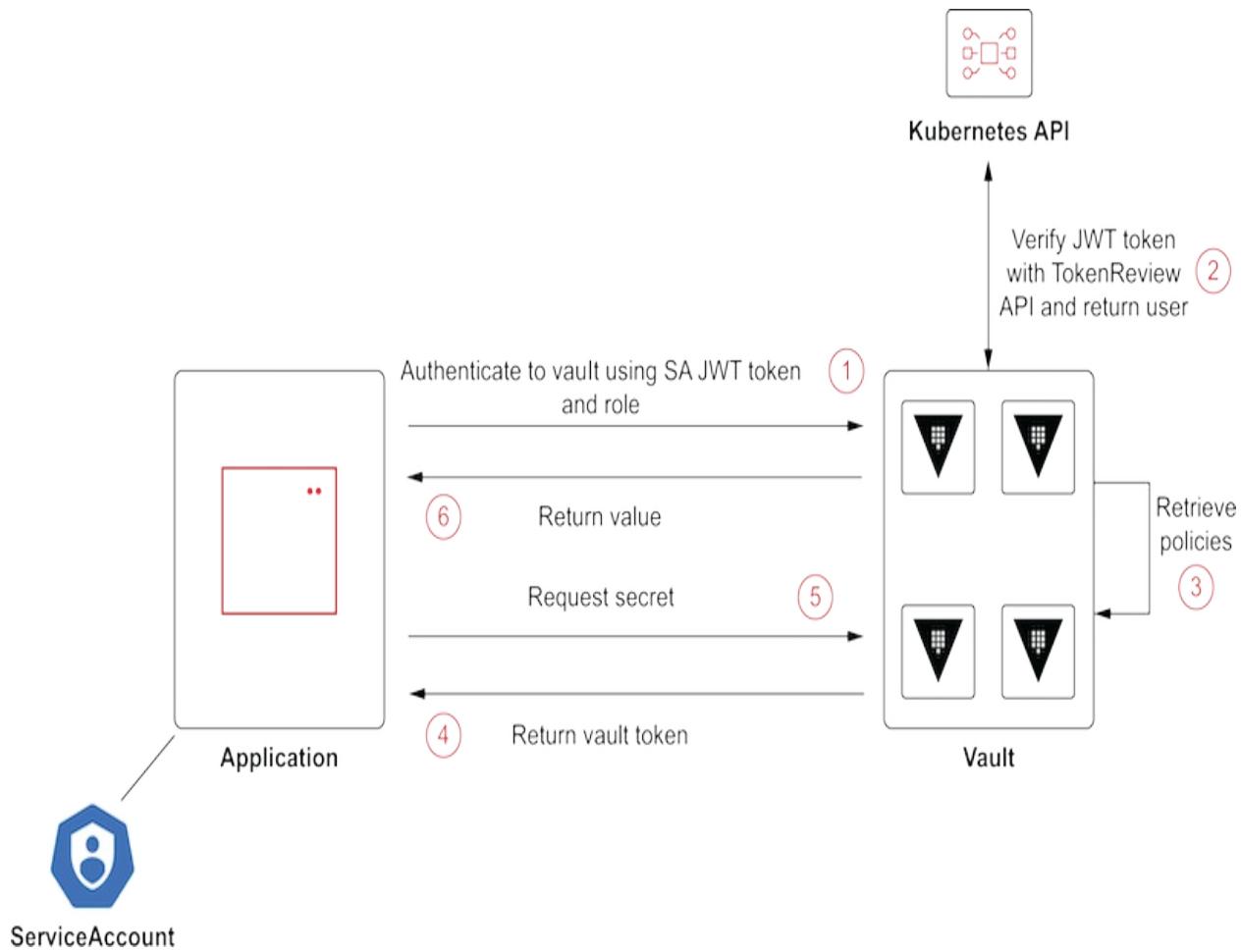
does require the additional step of managing lifecycle of the token, which can in the end, can result in a reduction of the overall security posture.

Vault, as described in [Application Access and Security](#), contains a diverse set of Auth Methods that it supports. Since we are operating within a Kubernetes environment, we can take advantage of a different method, the *Kubernetes Auth Method* to simplify how applications can interact with Vault avoid needing to manage the additional tasks of managing tokens to access Vault.

The Kubernetes Auth Method does not do away with the concept of tokens (quite the opposite), but instead of managing separate tokens that originate in Vault, it makes use of the JWT (JSON Web Tokens) that are associated with Kubernetes Service Accounts. Running applications on Kubernetes interact with Vault using the Service account token that is mounted within the pod (`/var/run/secrets/kubernetes.io/serviceaccount/token`). *Roles* are then created within Vault that map Kubernetes service accounts and Vault policies that define the level of access that is granted.

A diagram of the components involved in the Kubernetes Auth Method is shown in figure [5.5](#):

Figure 5.5. An overview of the components involved in the Kubernetes Auth Method



5.2.1 Configuring Kubernetes Auth

The first step that must be completed is that the Kubernetes Auth Method must be enabled within Vault

```
kubectl -n vault exec -it vault-0 -- vault auth enable kubernetes
Success! Enabled kubernetes auth method at: kubernetes/
```

Since Vault will be interacting with Kubernetes service account, it must be able to verify who the submitted token is associated with and if it is still valid. Fortunately, Kubernetes exposes the TokenReview API for just this purpose and essentially performs a reverse lookup of JWT tokens. By authenticating against the TokenReview API with a given JWT token of a ServiceAccount, details about the account will be returned including, but not limited to, the username.

For Vault to interact with the Kubernetes TokenReview API to inspect tokens provided by applications, it itself must be given permissions to make such requests.

Create a new Service Account called `vault-tokenreview` in a file called `vault-tokenreview-serviceaccount.yml` that will be used to by the Kubernetes Auth Method as shown in listing [5.11](#).

Listing 5.11. vault-tokenreview-serviceaccount.yml

```
---  
apiVersion: v1  
kind: ServiceAccount  
metadata:  
  name: vault-tokenreview #1
```

Now create the `vault-tokenreview` service account using the manifest created in listing [5.11](#).

```
kubectl -n vault apply -f vault-tokenreview-serviceaccount.yml  
serviceaccount/vault-tokenreview
```

With the `vault-tokenreview` service account now created, it must then be granted permissions to make requests against the Token Review API. There is a included Kubernetes *ClusterRole* that provides this level of access called `system:auth-delegator`. Create a new *ClusterRoleBinding* called `vault-tokenreview-binding` in a file called `vault-tokenreview-binding.yml` containing the following:

Listing 5.12. vault-tokenreview-binding.yml

```
apiVersion: rbac.authorization.k8s.io/v1  
kind: ClusterRoleBinding  
metadata:  
  name: vault-tokenreview-binding  
roleRef:  
  apiGroup: rbac.authorization.k8s.io  
  kind: ClusterRole  
  name: system:auth-delegator  
subjects:  
  - kind: ServiceAccount
```

```
name: vault-tokenreview
namespace: vault
```

Create the *ClusterRoleBinding* from the manifest contained in the `vault-tokenreview-binding.yml` file.

```
kubectl -n vault apply -f vault-tokenreview-binding.yml
clusterrolebinding.rbac.authorization.k8s.io/vault-tokenreview-bi
```

Since the JWT of the *vault-tokenreview* Service Account is needed by Vault to communicate to Kubernetes, execute the following set of commands to first find the name of the Kubernetes secret that contains the JWT token and then the base64 decoded token value stored within the secret.

Listing 5.13. Setting Vault TokenReview Variables

```
SA_SECRET_NAME=$(kubectl -n vault get serviceaccount vault-tokenreview -o jsonpath='{.secrets[0].name}') #1
VAULT_TOKENREVIEW_SA_TOKEN=$(kubectl -n vault get secret $SA_SECRET_NAME -o jsonpath='{.data.token}' | base64 -d) #2
```

Next, specify the configuration of the Kubernetes cluster by providing the location of the Kubernetes API, certificate of the Certificate Authority and the JWT of the service account obtained in listing [5.13](#).

```
kubectl -n vault exec -it vault-0 -- vault write auth/kubernetes/
  kubernetes_host="https://kubernetes.default.svc" \
  kubernetes_ca_cert="@/var/run/secrets/kubernetes.io/serviceaccount/ca.crt" \token_reviewer_jwt=$VAULT_TOKENREVIEW_SA_TOKEN
```

Success! Data written to: auth/kubernetes/config

Now that Vault is capable of authenticating requests, applications that want to obtain resources from within Vault must have a role associated to them in order to gain access to values stored within Vault. A role consists of the *name* and *namespace* of the Service Account submitting the request along with a set of Vault policies. Recall in listing [5.10](#), the *agents* application is running using a service account called *agents*. While there was little distinction of which Service Account was used to run the application previously, it is needed at this point in order to facilitate the integration with Vault using the

Kubernetes Auth Method.

Execute the following to create a new *role* within vault called *agents*

Listing 5.14. Creating a Vault Role

```
kubectl -n vault exec -it vault-0 -- \  
  vault write auth/kubernetes/role/agents \ #1  
    bound_service_account_names=agents \ #2  
    bound_service_account_namespaces=vault \ #3  
    policies=agents_reader #4
```

```
Success! Data written to: auth/kubernetes/role/agents
```

The creation of the role within vault enables the *agents* application to make use of the Kubernetes auth method to retrieve values from vault.

5.2.2 Testing and Validating Kubernetes Auth

To test and validate the integration of Kubernetes Auth within the *agents* application, first remove any existing artifacts that may still exist from the prior Token Auth Method based approach as they no longer will be needed

```
kubectl -n vault delete deployment agents  
kubectl -n vault delete secrets agents-vault-token
```



Note

Feel free to ignore any errors related to resources not being found. This step is to ensure that you have a fresh environment that implements the Kubernetes Auth Method

In the prior deployment of the *agents* application, the QUARKUS_VAULT_AUTHENTICATION_CLIENT_TOKEN contained the value of the token used to authenticate against vault. When migrating to the Kubernetes Auth Method, the QUARKUS_VAULT_AUTHENTICATION_KUBERNETES_ROLE will be used instead and reference the *agents* role created in Vault in listing [5.14](#).

Create a file called deployment_kubernetes_auth.yaml containing the

following *Deployment* definition.

Listing 5.15. deployment_kubernetes_auth.yml

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: agents
spec:
  replicas: 1
  selector:
    matchLabels:
      app: agents
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: agents
    spec:
      containers:
        - env:
            - name: QUARKUS_VAULT_AUTHENTICATION_KUBERNETES_ROLE
              value: "agents"
          image: quay.io/ablock/agents
          imagePullPolicy: Always
          name: agents
          ports:
            - containerPort: 8080
              protocol: TCP
        restartPolicy: Always
        serviceAccountName: agents
```

Now, create the deployment from the manifest created in listing [5.15](#).

```
kubectl -n vault apply -f deployment_kubernetes_auth.yml
deployment.apps/agents
```

Once the application has started, query for agent maria to confirm that the value stored in vault can be successfully retrieved, thus validating that the

Kubernetes Auth Method has been successfully integrated.

```
kubectl -n vault exec -it $(kubectl get pods -l=app=agents \
-o jsonpath={.items[0].metadata.name}) -- \
curl http://localhost:8080/agents/maria

{"email":"maria@acme.org","location":"Mexico City, Mexico","name"
```

A successful response demonstrates how the Kubernetes Auth method can be used to retrieve secrets from vault without explicitly providing the application a vault token to authenticate. If a successful result was not returned, confirm the steps as described in this section.

5.3 The Vault Agent Injector

The Kubernetes Auth Method simplified how applications deployed on Kubernetes can access values stored within Vault. One of the challenges presented by making use of either the Token or Kubernetes Auth methods as described in [5.2](#) is that the application needs to be Vault aware. In many cases, especially in legacy applications or those provided by third party vendors, it may not be possible to modify the source code in order to configure this type of integration.

To overcome these challenges, several approaches emerged using patterns in the Kubernetes ecosystem to address how values stored within Vault are made available to applications. Each leaned on a key characteristic of a Pod in Kubernetes where volumes could be shared between containers using an *emptyDir* volume type. A separate container could then be packaged within the pod whose responsibility is to facilitate the interaction with Vault and provide the secret values to the application through the shared volume.

Two patterns in Kubernetes were adopted to support this approach:

- **Init Containers**

A container or set of containers that execute before the application containers are started. In the context of Vault, assets are retrieved from Vault and placed in a shared volume that is pre-populated for the application to consume.

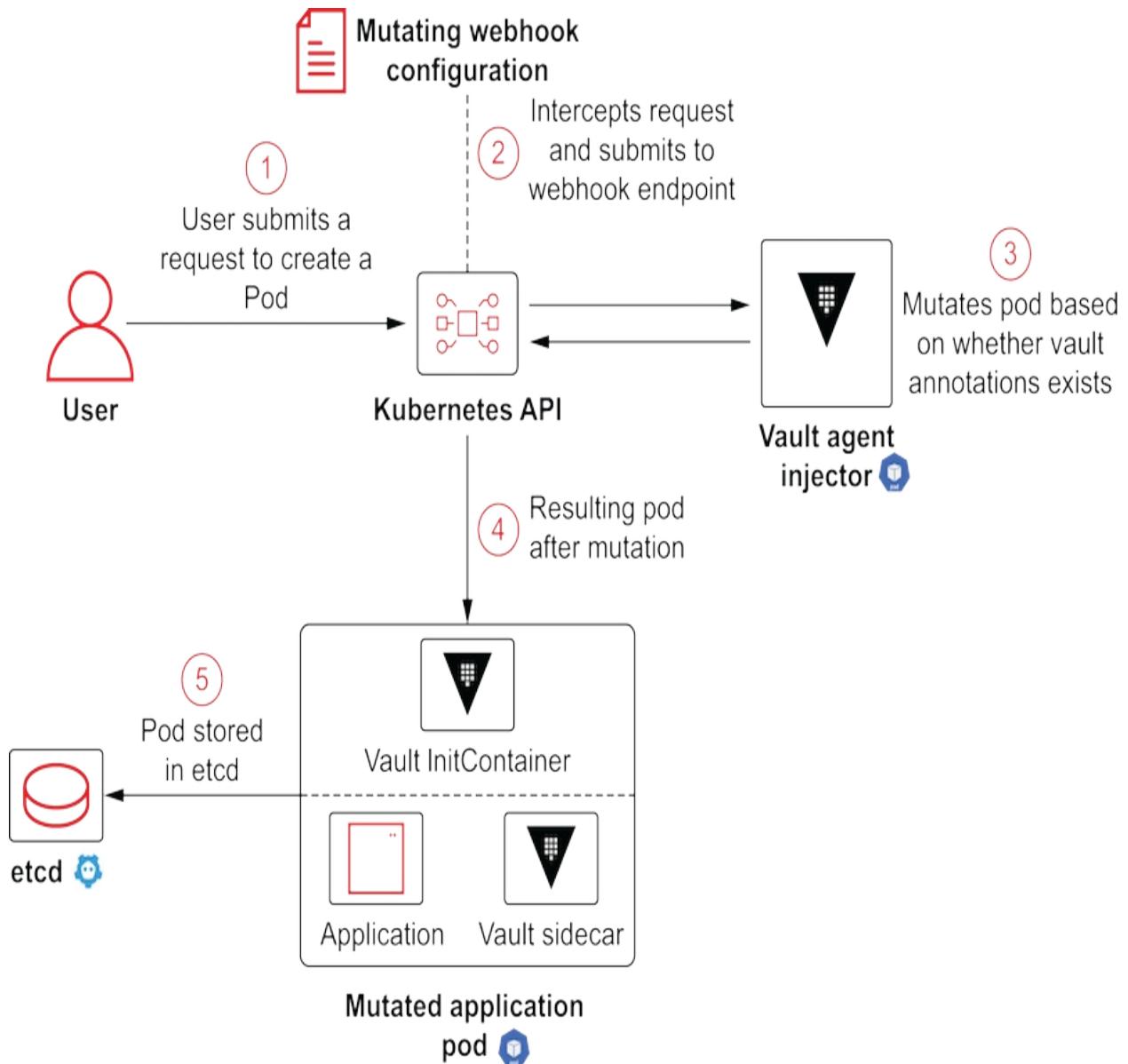
- **Sidecars**

Containers that run along side the application container. In the context of Vault, they would continue to interact with Vault and refresh the content of the shared volume with assets from Vault.

To avoid the burden of requiring end users to develop and maintain their own set of containers for interacting with Vault as well as providing a mechanism for automatically injecting Vault aware containers into pods, the Vault Agent Injector from HashiCorp was created.

The Vault Agent Injector runs as a Pod in Kubernetes and monitors for applications seeking to become Vault aware based on annotations declared within their pod. Once the injector has been installed to a cluster, for any other pod is created, an admission webhook is sent to the Vault Agent Injector with the details of the pod. If specific annotations, and in particular `vault.hashicorp.com/agent-inject: true`, are present within the pod, the definition of pod itself modified to automatically inject an *initContainer* and/or *sidecar* container. This process leverages the Kubernetes MutatingWebhookConfiguration feature that allows for the modification of Kubernetes resources before they are persisted to *etcd* and is detailed in figure [5.6](#):

Figure 5.6. A pod being modified by the MutatingWebhookConfiguration to inject the Vault Agent Injector at admission time



5.3.1 Configurations to Support Kubernetes Vault Agent Injection

To demonstrate how an application can have values stored in Vault be injected with minimal changes to the application itself, let's once again use the *agents* application as the target. Let's first define a new *secret_* within Vault called *config* containing properties related to the application. Through the Vault Agent Injector, the values provided within the secret will be added to a file within the application pod.

First, define the config Secret in Vault by executing the following command

Listing 5.16. Creating a Key/Value Secret

```
kubectl -n vault exec -it vault-0 -- \
  vault kv put agents/config \ #1
  mission="Kubernetes Secrets" \ #2
  coordinator="Manning Publishing"
```

With the new value added, the next step is to modify the Deployment for the *agents* application by defining several annotations needed to not only support automatic container injection, but also to customize how the values are presented in the pod.

Create a new file called `deployment_vault_agent.yml` containing the following

Listing 5.17. deployment_vault_agent.yml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: agents
spec:
  replicas: 1
  selector:
    matchLabels:
      app: agents
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
    metadata:
      annotations:
        vault.hashicorp.com/agent-inject: "true" #1
        vault.hashicorp.com/role: "agents" #2
        vault.hashicorp.com/agent-inject-secret-config.properties
        vault.hashicorp.com/agent-inject-template-config.properties
          {{- with secret "agents/config" }}
            {{- range $k, $v := .Data.data }}
              {{$k}}: {{$v}}
            {{- end }}
          {{- end }}
```

```

{{- end }}
labels:
  app: agents
spec:
  containers:
    - image: quay.io/ablock/agents
      imagePullPolicy: Always
      name: agents
      ports:
        - containerPort: 8080
          protocol: TCP
    restartPolicy: Always
    serviceAccountName: agents

```

The Vault Agent Injector builds upon the Kubernetes Auth Method, so a portion of the configurations that were applied in the [5.2.1](#) section will be reused.

The `vault.hashicorp.com/role` denotes the name of the role associated with the Service Account running the pod as well as the policies in order to grant access to content within Vault.

The annotation beginning with `vault.hashicorp.com/agent-inject-secret-` is used to define the file that will ultimately be created within the application pod. The value of this annotation refers to the name of the Secret within vault. The secret config within the agents path was created in listing [5.16](#). The remainder of the annotation key refers to the name of the file that will be created in the pod. So, an annotation with the key `vault.hashicorp.com/agent-inject-secret-config.properties` results in a file named `config.properties` within the pod.

Finally, the annotation beginning with `vault.hashicorp.com/agent-inject-template-` refers to a template in the Consul language and defines how the content of the secret is rendered. Similar to the annotation beginning with ``vault.hashicorp.com/agent-inject-secret-``, the remaining portion references the name of the file that will be created. The template defined here merely loops through all of the keys and values contained within the secret.

If an existing `agents` deployment is present in the cluster, delete it to ensure that any of the existing integrations with vault as described in either the [5.1.2](#) or [5.2](#) sections are not used moving forward.

```
kubectl -n vault delete deployment agents  
deployment.apps/agents
```

Now, execute the following command to create the *agents* deployment which will make use of the Kubernetes Vault Agent Injector.

```
kubectl -n vault apply -f deployment_vault_agent.yml  
deployment.apps/agents
```

If you observe the state of the *agents* pod, you will notice several differences with this deployment.

```
kubectl -n vault get pod -l=app=agents
```

NAME	READY	STATUS	RESTARTS	AGE
agents-795fc5565-6f6cg	2/2	Running	0	79s

First, notice under the *READY* column, there are now 2 containers. The additional container is the sidecar responsible for keeping the contents of the secrets sourced from Vault up to date. In addition, before either the sidecar or agents container started, an *initContainer* called *vault-agent-init* also present in the pod pre-seeded the an emptyDir volume mounted at */vault/secrets* (This path can be modified via an annotation) with the contents from the Vault secret. Since the *initContainer* must successfully complete before starting the primary containers, and since both containers within the pod are currently running, you can be assured that the Secret values have been retrieved from Value.

Let's verify by viewing the contents of the file located at */vault/secrets/config.properties* within the agents container.

Listing 5.18. Viewing the file created by the vault secrets injector

```
kubectl -n vault exec -it $(kubectl get pods -l=app=agents \  
-o jsonpath={.items[0].metadata.name}) \  
-c agents -- cat /vault/secrets/config.properties #2  
  
coordinator: Manning Publishing  
mission: Kubernetes Secrets
```

Feel free to update the contents of the config secret within Vault. The sidecar container bundled within the pod will routinely check the status of the secret and update the contents of the file within the application container.

By using the Kubernetes Vault Agent Injector, Vault secrets can be provided automatically to application without any changes to the application, abstracting the use of Vault entirely while providing the benefits of referencing sensitive values stored in HashiCorp Vault.

5.4 Summary

- HashiCorp Vault can be installed quickly and easily to a Kubernetes environment using a Helm chart
- Unsealing a HashiCorp Vault instance is the process of obtaining the plaintext root key to enable access to the underlying data
- The Kubernetes Auth Method uses a Kubernetes Service Account to authenticate with HashiCorp Vault
- The Kubernetes TokenReview API provides a method for extracting user details from a JWT token
- The HashiCorp Vault Agent Injector mutates the definition of a pod allowing it to obtain secrets stored in Vault

6 Accessing Cloud Secret Stores

This chapter covers

- Using Container Storage Interface (CSI) and the Secrets Store CSI driver to inject secrets as volumes from cloud secret stores
- Populating cloud secrets into Kubernetes cluster as Kubernetes Secrets
- Using auto rotation of secrets in the Secret Storage CSI driver to increase security posture
- Consuming sensitive information from cloud secret stores

Chapter 5 introduced HashiCorp Vault to securely store and manage sensitive assets for applications deployed to Kubernetes and demonstrate how both applications and Vault can be configured to provide seamless integration with one another.

This chapter expands the idea introduced in the previous chapter of using an external secrets management tool to store secrets and injecting them inside the Pod either as a volume or as an environment variable. But in this chapter we'll focus on cloud secret stores like Google Secret Manager, Azure Key Vault, or AWS Secrets Manager.

First, we'll learn about Container Storage Interface (CSI) and Secret Store CSI driver and using them to inject secrets stored in HashiCorp Vault. Then we'll see how to inject secrets using secret store CSI driver as Kubernetes secrets and the usage of secret auto rotation. Finally, the integration between CSI driver and Google Secret Manager, Azure Key Vault, and AWS Secrets Manager is shown so secrets are injected directly from the secret store to the Pod.

6.1 Container Storage Interface & Secret Store CSI drivers

As we have seen thus far, the etcd database stores the Kubernetes secrets

either un-encrypted (by default) or encrypted, as shown in Chapter 4. But what if we don't want to store my secrets in the etcd and store and manage them outside of the Kubernetes cluster? One option seen in Chapter 5 is to use HashiCorp Vault and HashiCorp Vault Agent to store secrets and inject them into a Pod.

HashiCorp Vault is an option, but nowadays, cloud providers also offer their key vaults; for example, Google has Google Secret Manager or Amazon with AWS Secrets Manager. What's happen if we want to use them to store secrets instead of etcd or Hashicorp Vault? Could we use the same approach previously but inject secrets as volume or environment variables from an external cloud store instead of etcd?

The answer is YES!. But before we see how to do it, we need to introduce the Container Storage Interface (CSI) initiative, a standard for exposing arbitrary block and file storage storage systems to containerized workloads like Kubernetes.

6.1.1 Container Storage Interface (CSI)

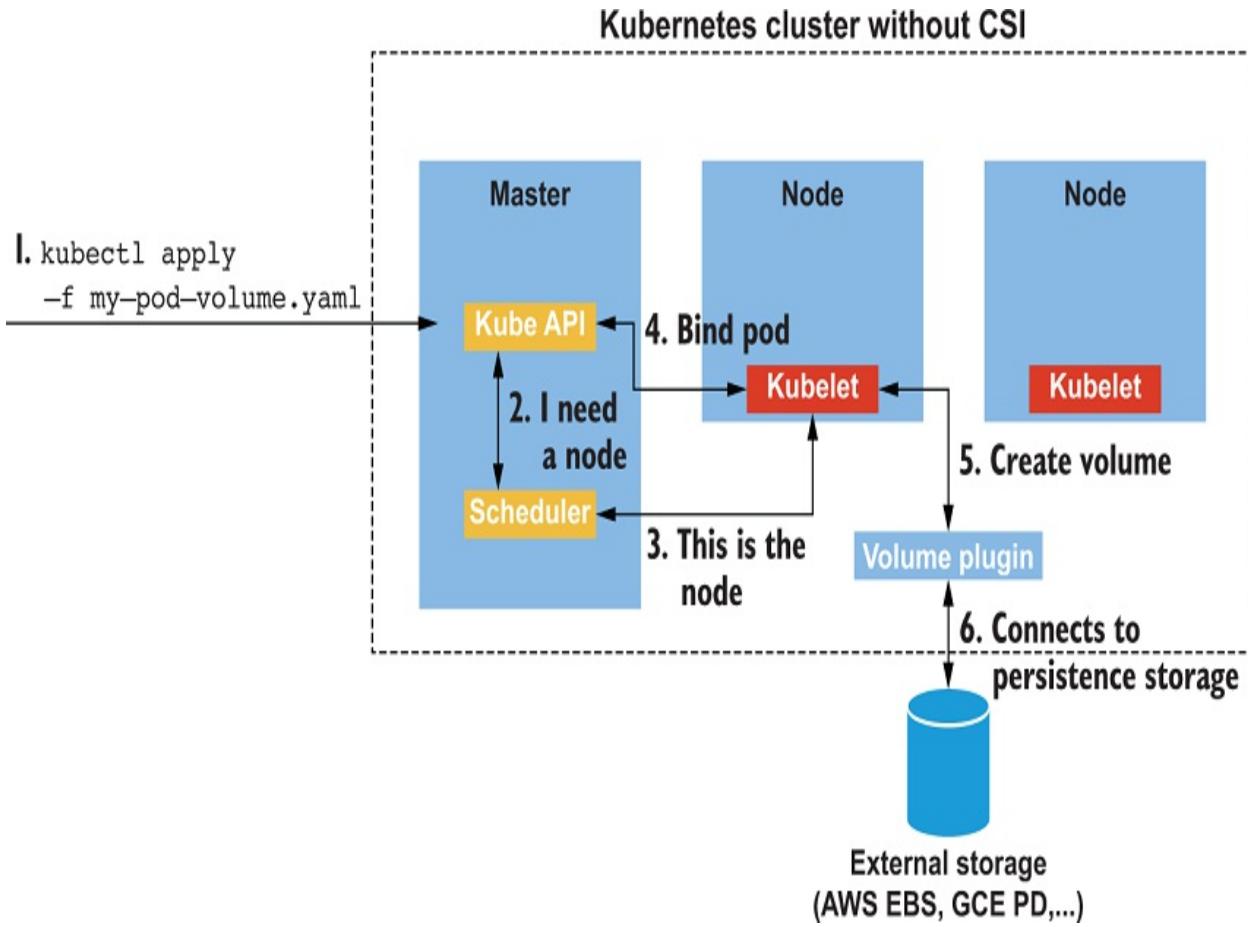
As we've seen in Chapter 2, a Kubernetes volume is a directory containing some data access to the containers running inside pods. The physical storage of the volume is determined by the volume type used.

The volume is mapped during the Pod initialization executing the following steps:

The API server receives a command to create a new Pod to the Kubernetes cluster. The scheduler finds a node that meets the desired criteria and sends the Pod definition to the node. Then node kubelet reads the Pod definition, sees that we want to attach a volume to the Pod container, and creates the volume through the *volume plugins* part. This volume plugin is responsible for connecting to the configured persistence storage.

Figure [6.1](#) summarizes all these steps:

Figure 6.1. Process of creating a Volume from Pod perspective



As we can see, the *volume plugins* are something that belongs to the Kubernetes core. This approach has the following drawbacks:

- Volume plugin development is coupled to the Kubernetes development and release cycle. Any new supported volume (and volume plugin) requires a new version of Kubernetes.
- Any correction in a volume plugin (fix bug, improvement, ...) requires a new release of Kubernetes.
- Since volume plugins are inside Kubernetes, the source code is open; this is not a problem so far until we need to make the plugin private.

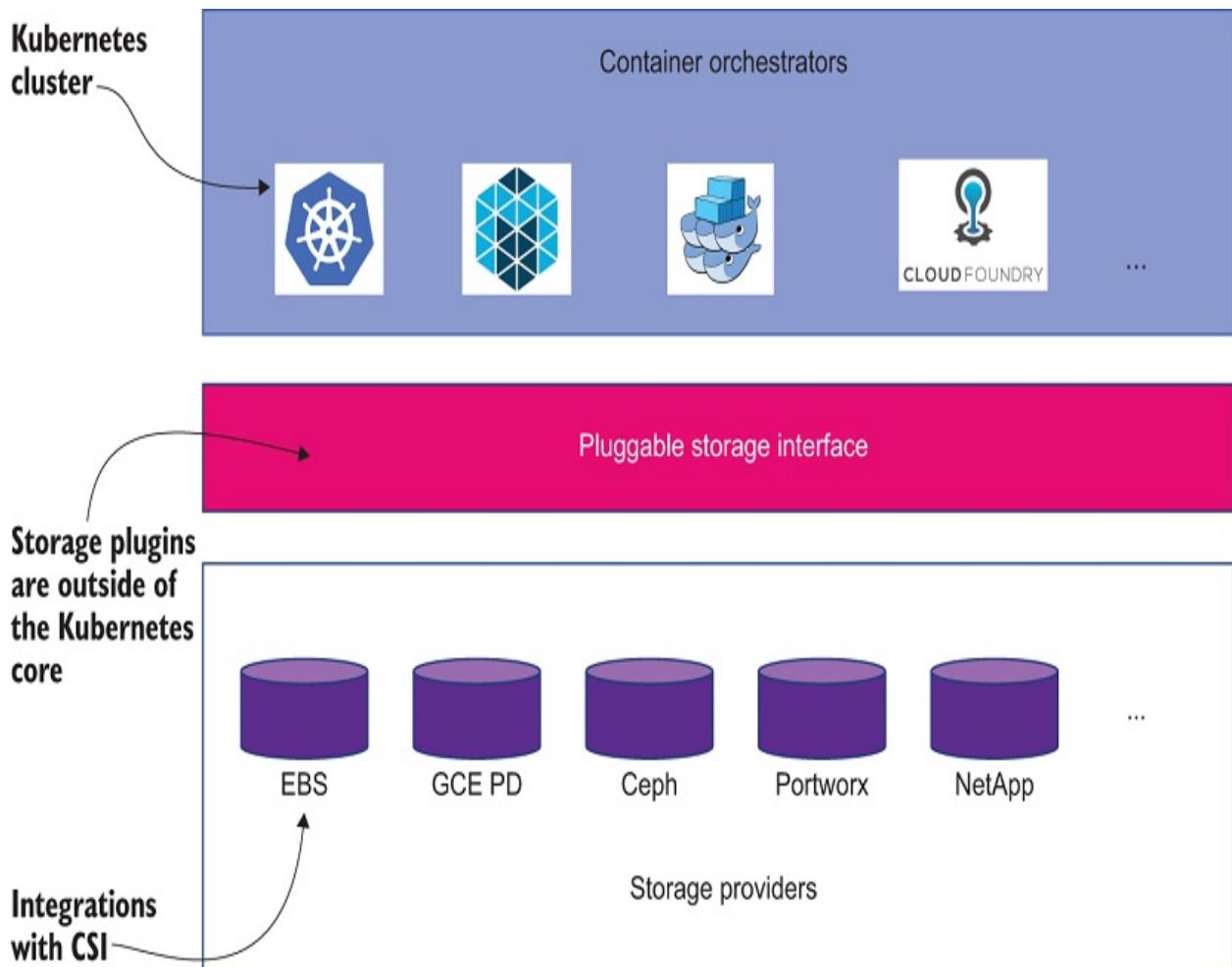
Container Storage Interface (CSI) is an initiative to unify the storage interface of Container Orchestrator (CO) Systems like Kubernetes, Mesos, Docker Swarm, etc., combined with storage vendors like Ceph, Azure Disk, GCE persistent disk, etc.

The first implication of CSI is that any implementation is guaranteed to work

with all container orchestrators. The second implication is the location of the CSI elements. They are outside the Container Orchestrator core (i.e., Kubernetes core), making them free to develop and release independently of the CO.

In the figure [6.2](#), a quick overview of the Container Storage Interface is seen:

Figure 6.2. CSI schema



The CSI driver is the bridge between container clusters and the persistent storage implementing the operations required by the CSI specification for the specific storage. CSI drivers may provide the following functionalities:

- The creation of persistent external storage.
- The configuration of the persistent external storage.

- Manages all input/output (I/O) between cluster and storage.
- Advanced disk features such as snapshots and cloning.

Some of the CSI drivers are Alicloud Disk, AWS Elastic Block Storage, Azure disk, CephFS, DigitalOcean Block Storage, or GCE Persistent Disk.

6.1.2 Container Storage Interface and Kubernetes

CSI gained general availability (GA) status at version 1.13 of Kubernetes and can be used with Kubernetes Volumes components (Persistent Volumes, Persistent Volumes Claim and Storage Class) as usually done.

Kubernetes has some components which don't belong to the core to interact with the external pluggable container storage. This interaction occurs through Google Remote Procedure Calls (*gRPC*) on domain sockets.

A Kubernetes cluster with CSI installed has the following components:

- **Kubernetes core**

This is the core of Kubernetes where most of the things introduced in this book live.

- **CSI external components**

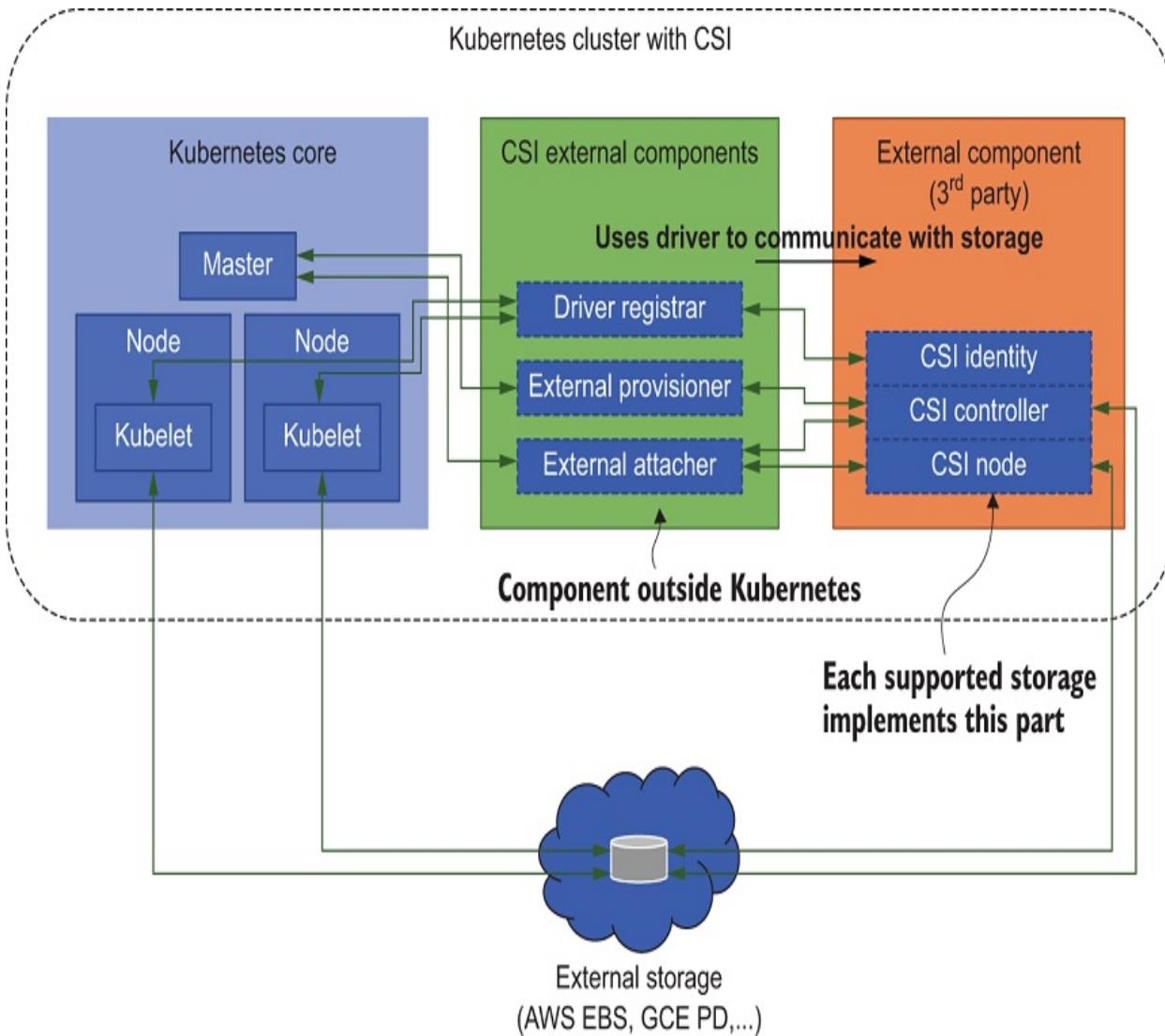
It is a set of Linux containers that contains common logic to trigger appropriate Kubernetes events to the *CSI driver*. Although using these containers isn't mandatory, they help reduce the amount of boilerplate code required to implement the CSI driver.

- **Third-party external components**

This is a vendor-specific implementation (CSI driver) to communicate with its persistent storage solution.

Each of these components has sub-components, and the most important ones can be seen in the figure [6.3](#):

Figure 6.3. Kubernetes cluster with CSI



For the scope of this book, this is all we need to know about CSI specification and its integration with Kubernetes. A deep understanding of the system would be required to implement a specific CSI driver for a persistent storage system.

Thanks to this separation of concerns, we can now write and deploy plugins exposing new storage systems in Kubernetes without having to touch the Kubernetes core, so no further release of Kubernetes is required.

6.1.3 Container Storage Interface and Secrets

Container Storage Interface is designed so there is a standard way to expose

all the different storage systems to Kubernetes and container workloads. With this standard in place, all you have to do is deploy the plug-in (CSI driver) on your cluster. Can this be extended to another kind of systems such as secret stores?

Secret Store CSI is no different from any other CSI driver, allowing Kubernetes to attach a secret placed in an external secret store into Pods as a volume. After the Volume is attached, the secret is available in the container's file system.

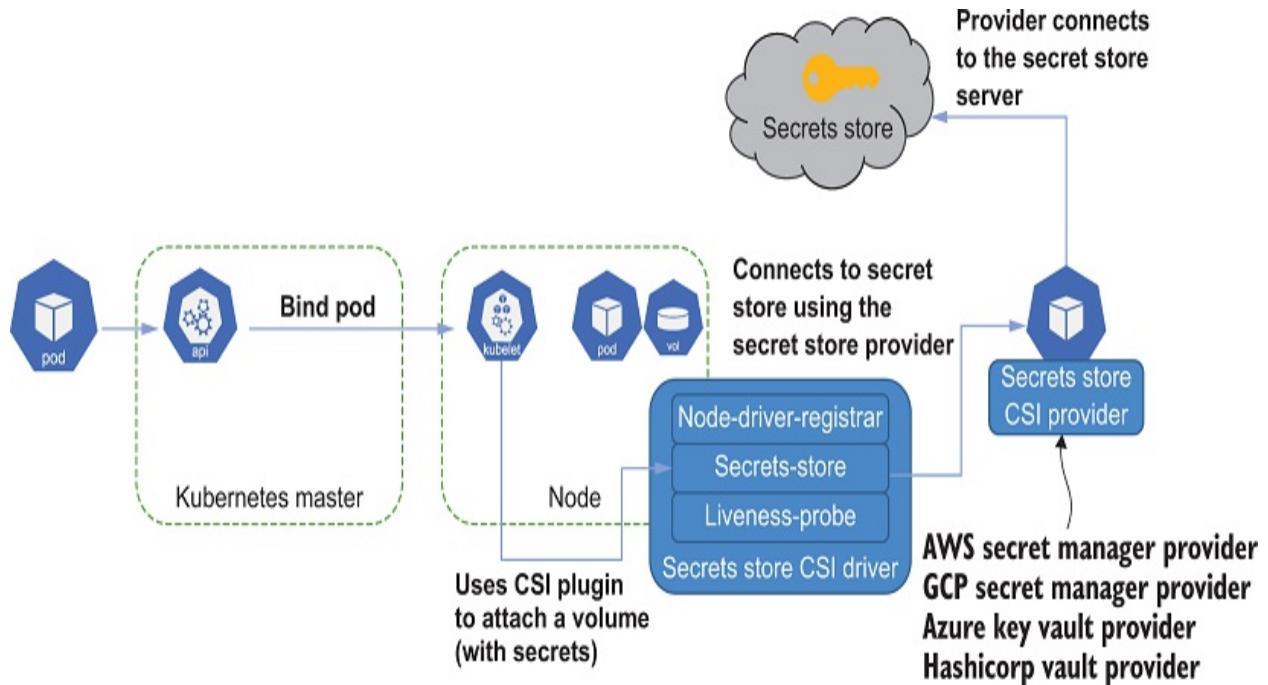
One of the aspects that make Secret Store CSI driver extensible is the presence of the secret store CSI provider. Instead of connecting to a secret store directly, the driver has a new level of abstraction in the form of a provider, so depending on the secret store used, the only element we need to install is the specific provider.

At the time of writing this book, the following secret stores are supported:

- AWS Secrets Manager, AWS Systems Manager Parameter Store (AWS Provider)
- Azure Key Vault (Azure Provider)
- Google Secret Manager (GCP Provider)
- HashiCorp Vault (Vault Provider)

Figure [6.4](#) shows an overview of the Secret Store CSI architecture:

Figure 6.4. Kubernetes cluster with CSI



Before we use CSI and Secret Store CSI driver, we need to create a Kubernetes cluster and install CSI and secret store CSI driver.

6.1.4 Installing pre-requisites

Let's start a new minikube instance by running the following command in a terminal window:

Listing 6.1. Start Minkube

```
minikube start -p csi --kubernetes-version='v1.21.0' --vm-driver=
```

6.1.5 Install the Secrets Store CSI Driver

The latest version of the Secret Store CSI driver at this time is `0.1.0`; although we might think it's an immature project, the truth is that it has been under development for a long time.

Run the commands shown in listing [6.2](#) to install Secret Store CSI:

Listing 6.2. Install Secret Store CSI driver

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes-sig  
kubectl apply -f https://raw.githubusercontent.com/kubernetes-sig  
kubectl apply -f https://raw.githubusercontent.com/kubernetes-sig  
kubectl apply -f https://raw.githubusercontent.com/kubernetes-sig  
kubectl apply -f https://raw.githubusercontent.com/kubernetes-sig
```

Since we'll use the mapping between secrets-store content as Kubernetes Secrets and auto-rotation of secrets later, some additional RBAC permissions are required to enable it.

Listing 6.3. Install RBAC

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes-sig  
kubectl apply -f https://raw.githubusercontent.com/kubernetes-sig
```



In Windows nodes

In case of running Kubernetes in Windows Nodes (notice that minikube runs inside a Linux Virtual Machine so we don't need to run the command), we'll need to run the following command too:

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes-sig
```

To validate the installation of the Secret Store CSI driver, run the commands shown in listing [6.4](#) to validate that Pods are running correctly:

Listing 6.4. Validate Secret Store CSI driver

```
kubectl get pods --namespace=kube-system #1
```

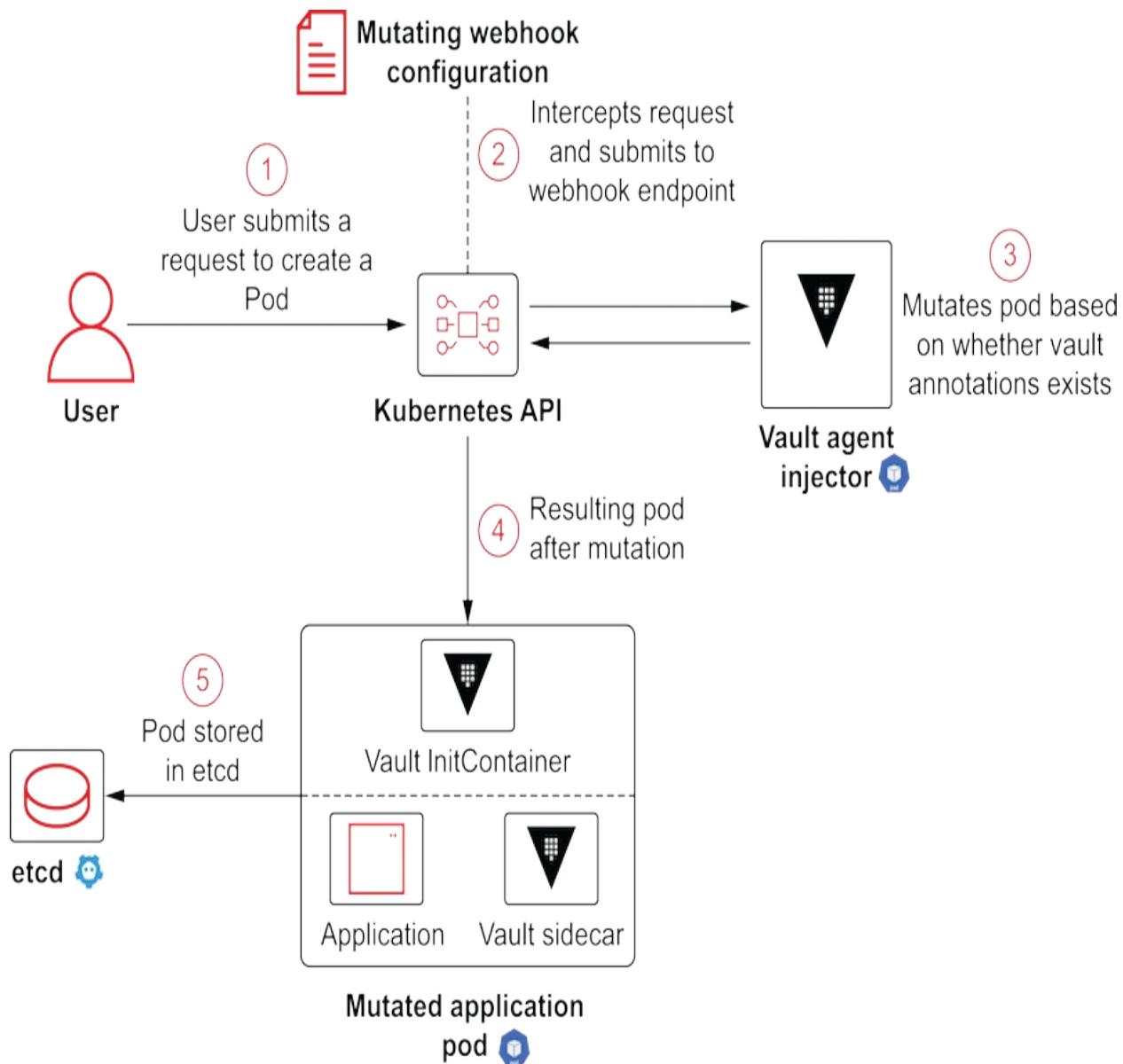
We should see the Secrets Store CSI driver Pods running on each agent node:

csi-secrets-store-8xlcn	3/3	Running	0	4m
-------------------------	-----	---------	---	----

We installed the Secrets Store CSI Driver into the Kubernetes cluster; it's time to select a Secret Store CSI provider and deploy it. For the first example, we'll use HashiCorp Vault as it's an agnostic secret store, and we've already used it in the previous chapter. But the most significant difference is how a secret is injected into the Pod. In one of the examples of Chapter 5, we saw

the HashiCorp Vault agent in charge of doing the injection. Figure 6.5 reminds you how HashiCorp Vault agent works.

Figure 6.5. A pod being modified by the MutatingWebhookConfiguration to inject the Vault Agent Injector at admission time



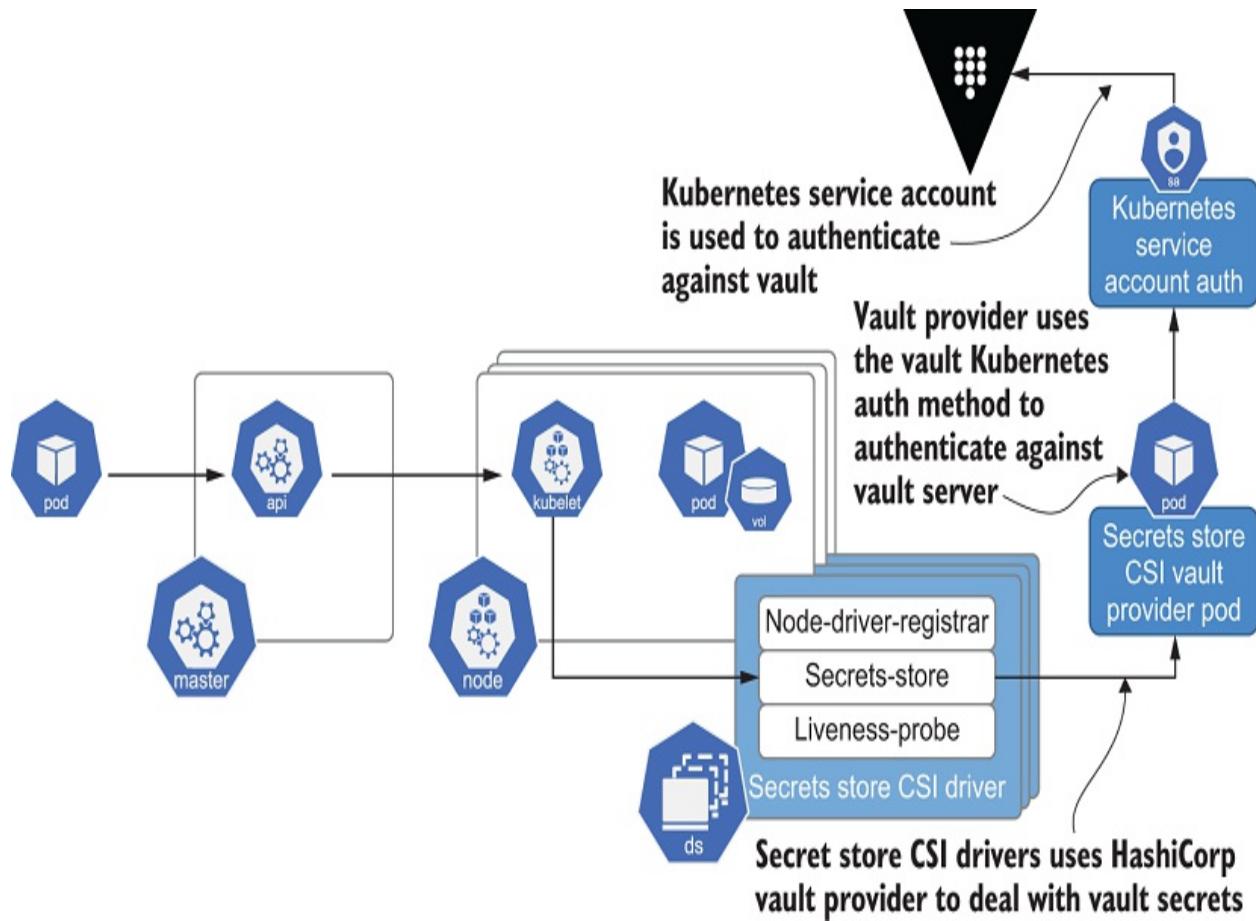
However, in this chapter, no agent will be used, and the Secret Store CSI driver will inject the secrets.

6.1.6 HashiCorp Vault Secrets through Secret Store CSI

Volume

We are now going to use HashiCorp Vault as secret store and consume the secrets using the Secret Store CSI driver and the HasiCorp Vault provider. Figure 6.6 shows an overview of what we implement in this section:

Figure 6.6. Secret Store CSI driver with HashiCorp Vault provider



The first thing we need is a HashiCorp Vault instance running in the Kubernetes cluster. We've already deployed a HashiCorp Vault instance in Kubernetes in Chapter 5; let's do it again in the current cluster:

To get started, first add the Hashicorp repository to Helm:

Listing 6.5. Adding Helm chart

```
helm repo add hashicorp https://helm.releases.hashicorp.com #1
```

"hashicorp" has been added to your

Then, retrieve the latest updates from the remote repositories:

```
helm repo update #1
```

```
Hang tight while we grab the latest from your chart repositories.  
...Successfully got an update from the "hashicorp" chart repository  
Update Complete. ✨ Happy Helming! ✨
```

Execute the following command to install the chart in development mode, and Vault CSI provider enabled:

Listing 6.6. Deploying HashiCorp Vault with Secret Store CSI provider

```
helm install vault hashicorp/vault \  
  --set "server.dev.enabled=true" \  
  --set "injector.enabled=false" \  
  --set "csi.enabled=true" #4
```

Run the command shown in [6.7](#) to wait until HashiCorp Vault deployment is up and running:

Listing 6.7. Wait until HashiCorp Vault is ready

```
kubectl wait --for=condition=ready pod -l app.kubernetes.io/name=
```

Create a secret inside HashiCorp Vault

Let's create a key-value secret inside Vault by opening an interactive shell session on Vault Pod:

Listing 6.8. Opening interactive shell

```
kubectl exec -it vault-0 -- /bin/sh #1
```

From this point, the commands issued are executed on the Vault container. Let's create the secret at the path `secret/pass` with a `my_secret_password` value:

Listing 6.9. Creating Secret

```
vault kv put secret/pass password="my_secret_password" #1

Key          Value
---          -----
created_time 2021-08-03T04:59:49.920719431Z
deletion_time n/a
destroyed     false
version       1
```

Configure Kubernetes authentication

The primary authentication method with Vault when using the Vault CSI Provider is the service account attached to the Pod. For this reason, we need to enable the Kubernetes authentication method and configure it so it can be used by the Vault CSI driver.

Still inside Vault container run the commands shown in listing [6.10](#):

Listing 6.10. Enable and configure Kubernetes auth method

```
vault auth enable kubernetes #1

Success! Enabled kubernetes auth method at: kubernetes/

vault write auth/kubernetes/config \
  issuer="https://kubernetes.default.svc.cluster.local" \
  token_reviewer_jwt="$(cat /var/run/secrets/kubernetes.io/serv
  kubernetes_host="https://$KUBERNETES_PORT_443_TCP_ADDR:443" \
  kubernetes_ca_cert=@/var/run/secrets/kubernetes.io/serviceacc

Success! Data written to: auth/kubernetes/config
```

Reading secrets using the Secrets Store CSI-Driver requires read permissions of all mounts and access to the secret itself.

Create a policy named `csi-internal-app`.

Listing 6.11. Applying Vault policy

```
vault policy write csi-internal-app - <<EOF #1
```

```
path "secret/data/pass" { #2
    capabilities = ["read"] #3
}
EOF
```

```
Success! Uploaded policy: csi-internal-app
```

Finally, create a Kubernetes authentication role named `my-app` that binds this policy with a Kubernetes service account named `app-sa`. The role is used in the Vault CSI Provider configuration (we'll see in the following section), and the service account is used to run the Pod.

Listing 6.12. Creating Kubernetes authentication role

```
vault write auth/kubernetes/role/my-app \
  bound_service_account_names=app-sa \
  bound_service_account_namespaces=default \
  policies=csi-internal-app \
  ttl=120m
```

```
Success! Data written to: auth/kubernetes/role/my-app
```

Lastly, exit the Vault Pod to come back to the computer.

Listing 6.13. Exiting Shell session

```
exit
```

Define a `SecretProviderClass` resource

The `SecretProviderClass` Kubernetes custom resource describes the configuration parameters given to the Secret Store CSI driver. The Vault CSI provider requires the following parameters:

- The address of the Vault server
- The name of the Vault Kubernetes authentication role
- The secrets to inject into Pod

The `SecretProviderClass` definition to connect to Vault installation is shown in listing [6.14](#):

Listing 6.14. vault-spc.yaml

```
apiVersion: secrets-store.csi.x-k8s.io/v1alpha1
kind: SecretProviderClass
metadata:
  name: vault-secrets #1
spec:
  provider: vault #2
  parameters:
    vaultAddress: "http://vault.default:8200" #3
    roleName: "my-app" #4
  objects: |
    - objectName: "my-password" #5
      secretPath: "secret/data/pass" #6
      secretKey: "password" #7
```

Apply the vault-secrets SecretProviderClass by running the following command:

Listing 6.15. Applying the vault-secrets SecretProviderClass

```
kubectl apply -f vault-spc.yaml -n default #1
```

Deploy a Pod with secret mounted

With everything in place, it's time to create a service account with the name app-sa (the same name configured in Configure Kubernetes authentication section) and one Pod creating a volume from the secret store CSI made above. The volume configuration now contains a csi section where we set the CSI driver to use, in this case, the secret store one (secrets-store.csi.k8s.io) and also the SecretProviderClass name created above (vault-secrets).

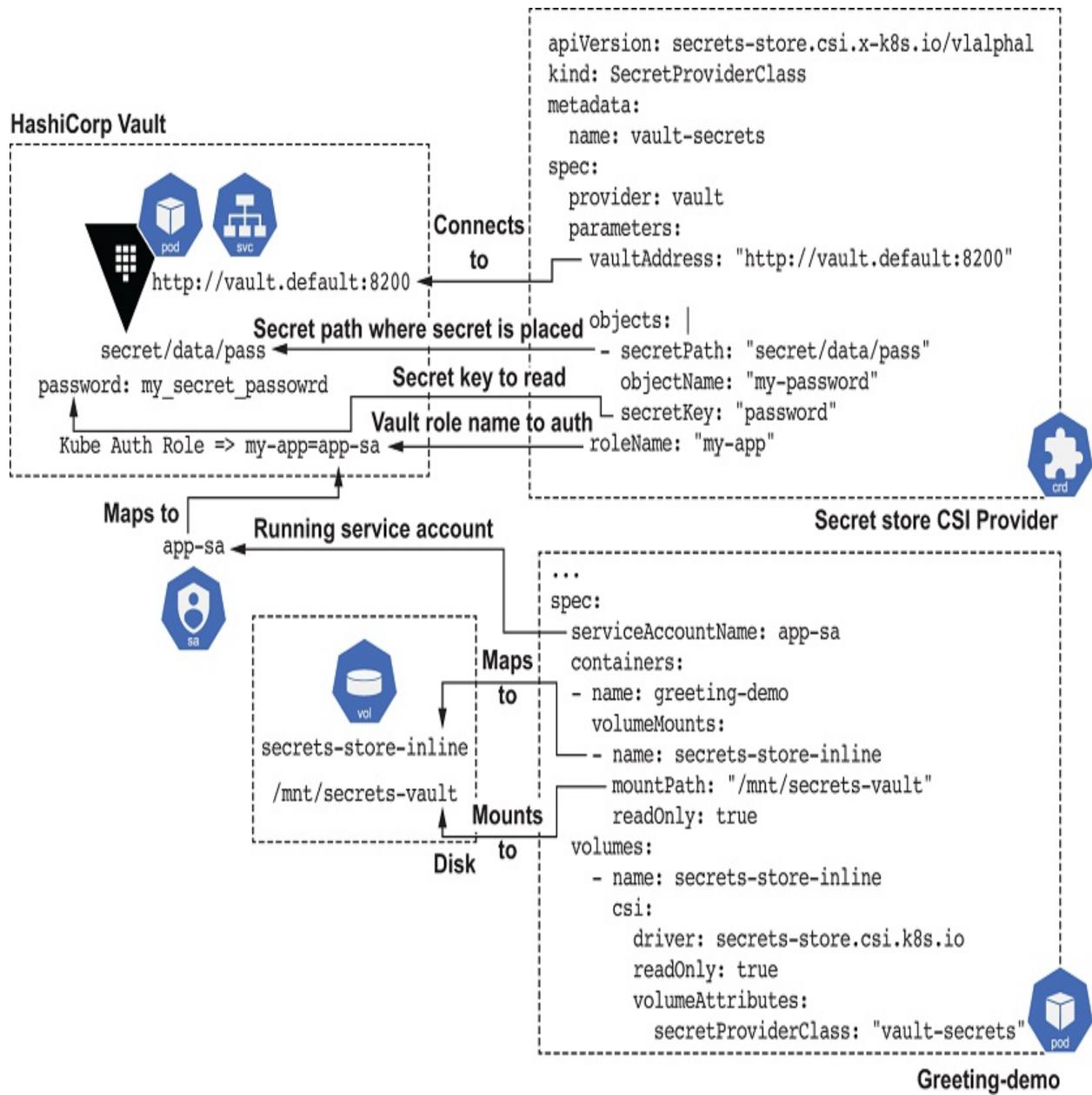
Listing 6.16. vault-app-pod.yaml

```
kind: ServiceAccount
apiVersion: v1
metadata:
  name: app-sa #1
---
kind: Pod
apiVersion: v1
```

```
metadata:
  name: greeting-demo
  labels:
    app: greeting
spec:
  serviceAccountName: app-sa #2
  containers:
    - image: quay.io/lordofthejars/greetings-jvm:1.0.0
      name: greeting-demo
      volumeMounts: #3
        - name: secrets-store-inline
          mountPath: "/mnt/secrets-vault" #4
          readOnly: true
  volumes:
    - name: secrets-store-inline
      csi: #5
        driver: secrets-store.csi.k8s.io #6
        readOnly: true
        volumeAttributes:
          secretProviderClass: "vault-secrets" #7
```

Figure [6.7](#) shows the relationship between all these elements:

Figure 6.7. Relationship between Secret provider, Pod and Vault



Deploy the Pod by running the following command:

Listing 6.17. Applying the Pod with CSI volume

```
kubectl apply -f vault-app-pod.yaml -n default #1
```

When the Pod is deployed, it will contain a volume with all the Vault secrets mounted as files.

Wait until the Pod is running:

Listing 6.18. Wait until greeting Pod is running

```
kubectl wait --for=condition=ready pod -l app=greeting --timeout=
```

Finally, we validate that the secret is mounted in `/mnt/secrets-vault` as specified in `volumesMount` section, and the file containing the secret is `my-password` as set in the `objectName` field. Run the command shown in listing [6.19](#):

Listing 6.19. Read the injected secret

```
kubectl exec greeting-demo -- cat /mnt/secrets-vault/my-password  
my_secret_password% #1
```

Let's recap, we've seen how to install CSI and the Secrets Store CSI Driver. Finally, we've created a `SecretProviderClass` resource to configure the secret store CSI driver to use HashiCorp Vault as secret storage.

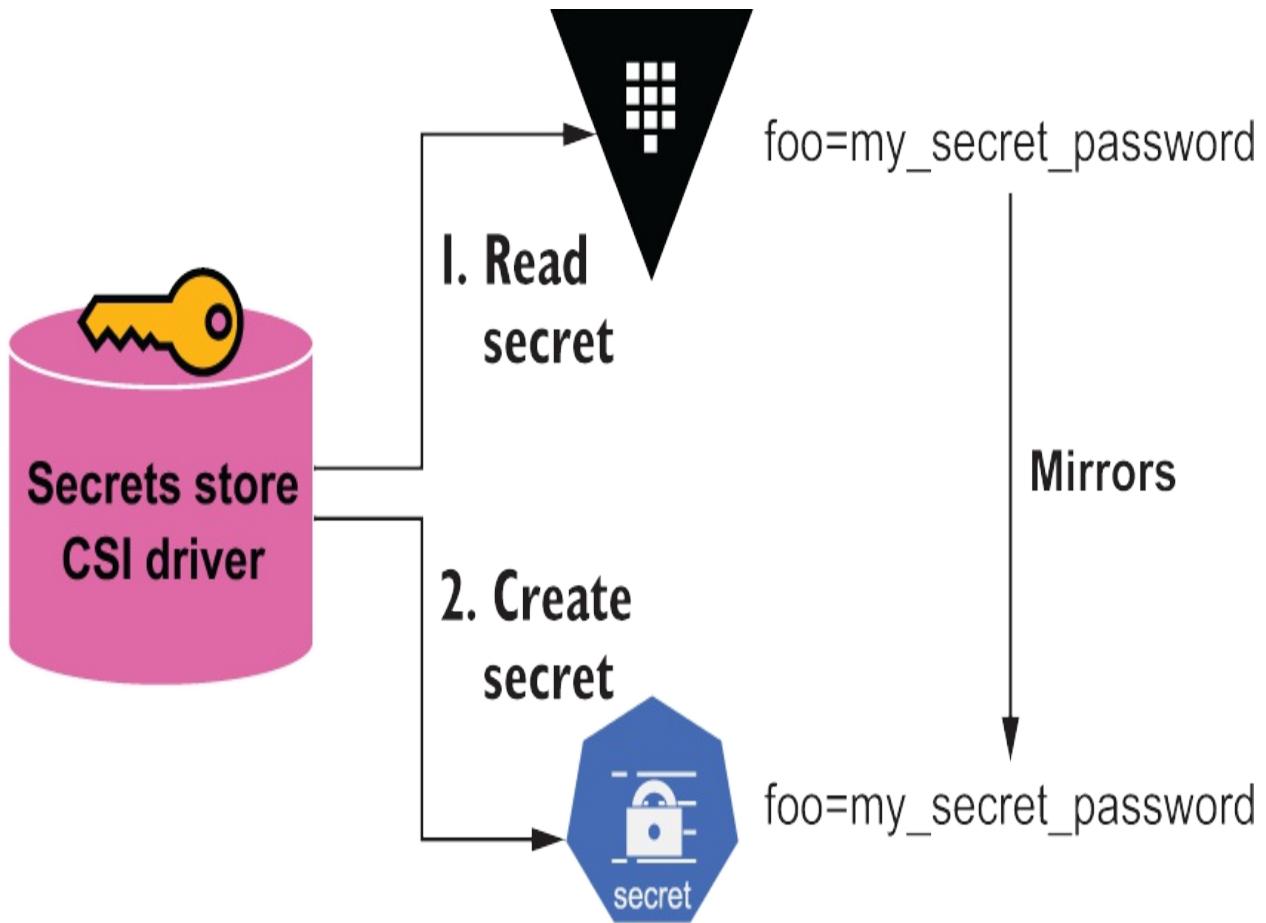
We now understand how Secret Store CSI driver works, we've seen that the secret is mounted as a volume and mounted to disk. But in some cases, you may need to mount these secrets as Kubernetes secrets. Let's explore how to do it.

6.2 Synchronize CSI secrets as Kubernetes Secrets

Synchronization is especially useful when the application (usually legacy application) needs to read secrets either as an environment variable or reading secrets directly using the Kubernetes API server.

Secret Store CSI driver mounts the secrets to disk using Kubernetes Volumes, but mirroring these secrets to Kubernetes Secrets is supported too by using the optional `secretObjects` field in the `SecretProviderClass` custom resource.

Figure 6.8. Secret Store CSI driver mirrors secrets as Kubernetes Secrets



Important

The volume mount is still required and must be defined for the Sync With Kubernetes Secrets.

We will expand the previous HashiCorp Vault example by also mapping the secret as a Kubernetes Secret.

6.2.1 Preparing the namespace

Before mapping the secrets as Kubernetes Secrets, we may delete the `SecretProviderClass` resource and undeploy the `greeting-demo` Pod to have a clean environment.

In a terminal window, run the following commands:

Listing 6.20. Cleaning up environment

```
kubectl delete -f vault-app-pod.yaml -n default #1  
kubectl delete -f vault-spc.yaml -n default #2
```

With the example undeployed, we can start the synchronization example.

6.2.2 Define a SecretProviderClass resource with secretObjects

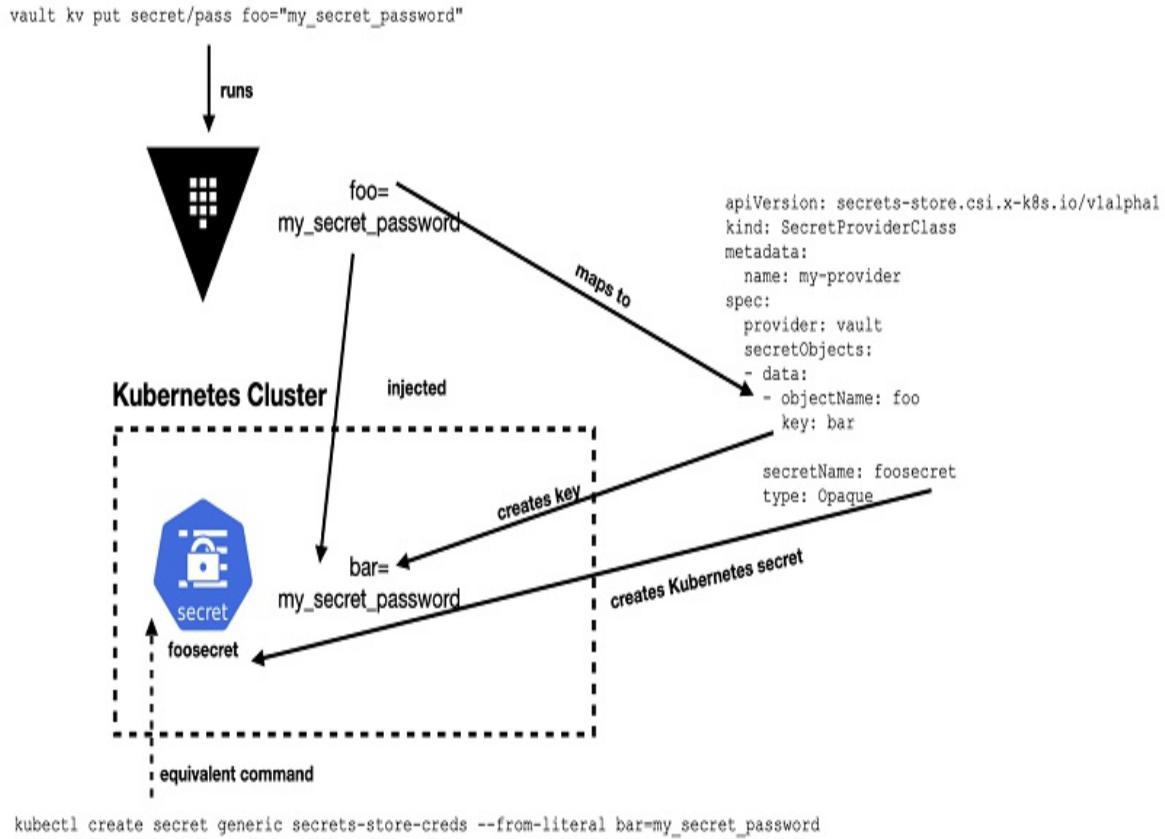
To mount the secrets from secret storage using the Secret Store CSI driver, we modify the SecretProviderClass resource adding the secretObjects field. This field is used to define how to map a secret from a secret store to a Kubernetes Secret.

In the following snippet, we see how the value of a secret placed at the secret store with named foo is mapped into a key named bar of an opaque Kubernetes Secret with name foosecret:

```
apiVersion: secrets-store.csi.x-k8s.io/v1alpha1  
kind: SecretProviderClass  
metadata:  
  name: my-provider  
spec:  
  provider: vault  
  secretObjects:  
  - data:  
    - key: bar #1  
      objectName: foo #2  
      secretName: foosecret #3  
      type: Opaque #4
```

We see the relationship between the secret storage, the SecretProviderClass definition and Kubernetes Secret in the figure [6.9](#):

Figure 6.9. Secret store, SecretProviderClass and Kubernetes Secrets



About supported secret types

At the time of writing this book the following secret types are supported:

- Opaque
- kubernetes.io/basic-auth
- bootstrap.kubernetes.io/token
- kubernetes.io/dockerconfigjson
- kubernetes.io/dockercfg
- kubernetes.io/ssh-auth
- kubernetes.io/service-account-token
- kubernetes.io/tls

Let's open the `vault-spc.yaml` created previously and add the `secretObjects` section to make as Kubernetes Secret, the data stored in the

HashiCorp Vault:

Listing 6.21. vault-spc.yaml

```
apiVersion: secrets-store.csi.x-k8s.io/v1alpha1
kind: SecretProviderClass
metadata:
  name: vault-secrets
spec:
  provider: vault
  secretObjects: #1
  - data:
    - key: password
      objectName: my-password #2
      secretName: my-secret #3
      type: Opaque
  parameters: #4
    vaultAddress: "http://vault.default:8200"
    roleName: "my-app"
  objects: |
    - objectName: "my-password"
      secretPath: "secret/data/pass"
      secretKey: "password"
```

Apply the resource:

Listing 6.22. Applying the SecretProviderClass

```
kubectl apply -f vault-spc.yaml #1
```

With SecretProviderClass deployed, it's time to deploy the Pod.

Deploy a Pod with secret mounted and secret objects

The secrets will only sync once you start a Pod mounting the secrets. List the current secrets in the default namespace before deploying the Pod.

Listing 6.23. List secrets

```
kubectl get secrets -n default #1
```

NAME	TYPE
------	------

```
sh.helm.release.v1      helm.sh/release.v1
vault-csi-provider-token-chhwj  kubernetes.io/service-account-toke
vault-token-qj2jk        kubernetes.io/service-account-toke
```

With everything in place, let's redeploy the Pod defined in the previous section:

Listing 6.24. Applying the Pod with CSI volume

```
kubectl apply -f vault-app-pod.yaml -n default #1
```

Wait until the Pod is running:

Listing 6.25. Wait until greeting Pod is running

```
kubectl wait --for=condition=ready pod -l app=greeting --timeout=
```

List the secret after the Pod has been deployed to inspect that the secret is now created.

Listing 6.26. List secrets

```
kubectl get secrets -n default
```

NAME	TYPE
my-secret	Opaque
sh.helm.release.v1.vault.v1	helm.sh/release.v1
vault-csi-provider-token-chhwj	kubernetes.io/service-account-toke
vault-token-qj2jk	kubernetes.io/service-account-toke

Describe the secret to validate that is created correctly.

Listing 6.27. Describe secret

```
kubectl describe secret my-secret
```

Name:	my-secret #1
Namespace:	default
Labels:	secrets-store.csi.k8s.io/managed=true #2
Annotations:	<none>
Type:	Opaque #3

```
Data
=====
password: 18 bytes #4
```

The equivalent kubectl command to create a secret like the one created by Secret Store CSI would be:

```
kubectl create secret generic my-secret --from-literal password=m
```

Finally, we delete the Pod to validate that it is automatically deleted since no Pods are consuming the secret.

Listing 6.28. Delete Pod

```
kubectl delete -f vault-app-pod.yaml -n default #1
```

And list the secrets again and we'll see that the `my-secret` secret is removed as no Pod is using it.

Listing 6.29. List secrets

```
kubectl get secrets -n default
```

NAME	TYPE
sh.helm.release.v1.vault.v1	helm.sh/release.v1
vault-csi-provider-token-chhwj	kubernetes.io/service-account-token
vault-token-qj2jk	kubernetes.io/service-account-token



Tip

Once the secret is created, we may set it as an environment variable with any Kubernetes secret.

```
env:
- name: MY_SECRET
  valueFrom:
    secretKeyRef:
      name: my-secret
      key: password
```

Remember that we are creating Kubernetes Secrets, and as such they need to be managed as shown in Chapters 3 and 4, encrypting them in Git repository, and enabling KMS to store them in Kubernetes encrypted and not encoded.

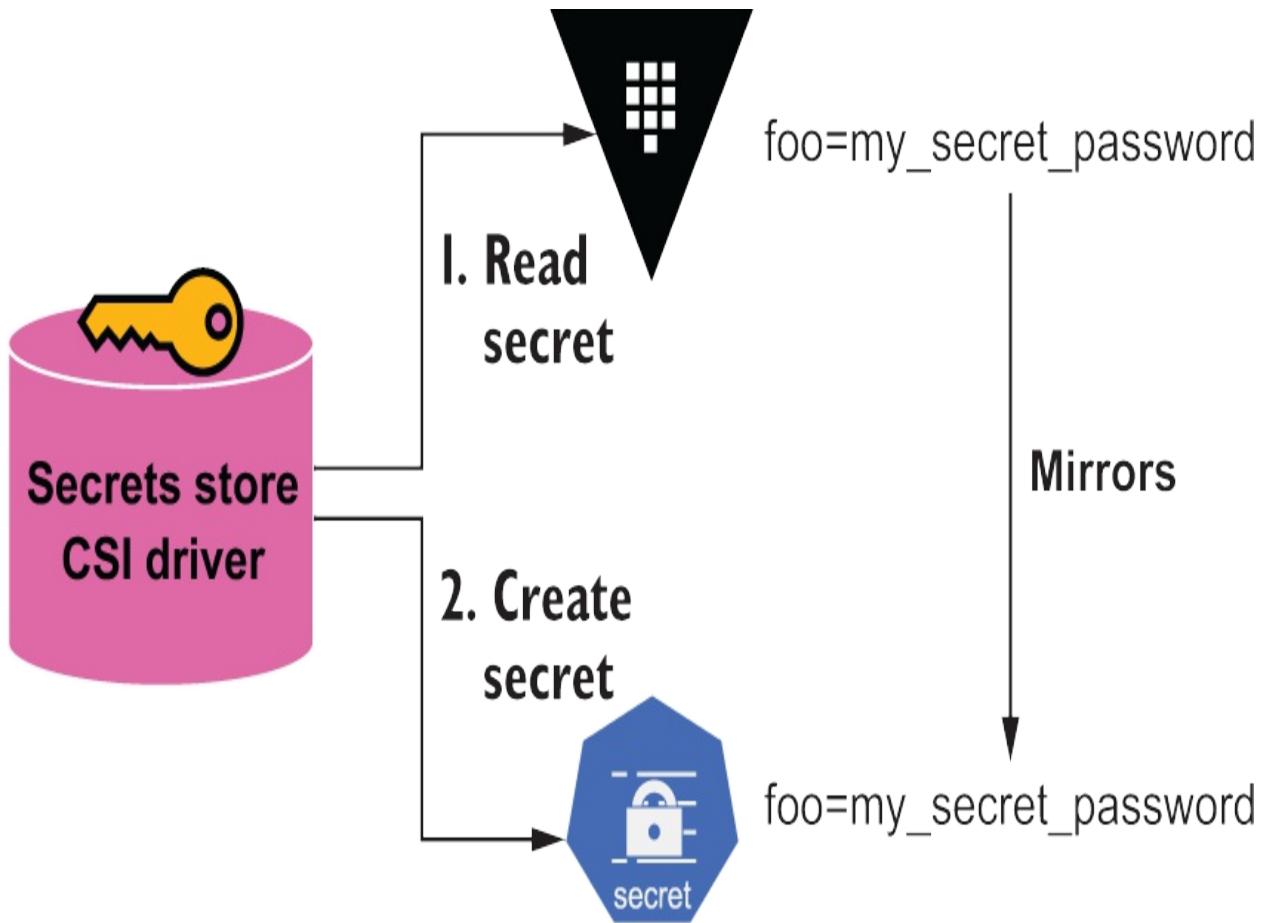
So far, we've seen how to mount secrets from secret stores as Kubernetes volumes and as Kubernetes Secrets too using the secret store CSI driver. But what's happen when the secret is updated in the secret store?

Secret Store CSI Driver supports the Secret auto rotation.

6.3 Autorotation of secrets to increase security posture

Imagine you receive an attack to your system, and some of your secret data is compromised, one of the first things you should do is to regenerate secrets to new values (rotation of the secrets). Secret Store CSI driver can detect an update of a secret in the external secret store after the Pod is running and populate this change in the corresponding volume content and Kubernetes Secrets if they are used as shown in [6.2](#).

Figure 6.10. Secret Store CSI driver mirros secrets as Kubernetes Secrets



(Auto)Rotation of secrets is the process that periodically change secrets data. There are two big wins on changing secret data, the first one to make it more difficult for an attacker to get a value and it's still valid on the system, the second one is in the case there is a secret data leak, we'll need to rotate the secrets as quick as possible to avoid major problems.

It's important to keep in mind Secret Store CSI driver only updates the place where the secret is stored (volume or Kubernetes Secret), but the application consuming these secrets needs to implement some logic to react to these changes and read the new secret value.

For example, in mounting the secret as a mounted volume, the application will need to watch for changes from it. If the secret is injected as an environment variable, then the Pod needs to be restarted to get the latest secret as an environment variable.



Tip

Projects like <https://github.com/stakater/Reloader> helps you on doing rolling upgrades on Pods automatically when a change on the associated ConfigMap or Secret is detected.

6.3.1 Preparing the namespace

We may undeploy the greeting-demo Pod (although it shouldn't be necessary) before using the auto-rotation feature to have a brand new Pod, for example.

If you have not deleted the Pod previously, run the following command in a terminal window:

Listing 6.30. Cleaning up environment

```
kubectl delete -f vault-app-pod.yaml -n default #1
```

The autorotation feature is disabled by default when the Secret Store CSI driver is installed. To enable this feature, Secrets Store CSI Driver Pod should start with `--enable-secret-rotation` flag to true and setting the rotation poll interval (frequency checking if a secret has changed) using the `rotation-poll-interval` flag.

To enable auto-rotation, let's stop the previous deployment by running the following command in a terminal:

Listing 6.31. Undeploy Secret Store CSI driver

```
kubectl delete -f https://raw.githubusercontent.com/kubernetes-si
```

Then we modify the deployment file of the Secret Store CSI Driver to configure `--enable-secret-rotation` and `rotation-poll-interval` flags. In this example, we set the `rotation-poll-interval` time to 1 minute, which means that every minute, the driver will query the secret store, checking if the value has been changed or not.

Listing 6.32. install-csi-polling.yaml

```
kind: DaemonSet #1
apiVersion: apps/v1
metadata:
  name: csi-secrets-store
  namespace: kube-system
spec:
  selector:
    matchLabels:
      app: csi-secrets-store
  template:
    metadata:
      labels:
        app: csi-secrets-store
      annotations:
        kubectl.kubernetes.io/default-logs-container: secrets-sto
    spec:
      serviceAccountName: secrets-store-csi-driver
      containers:
        - name: node-driver-registrar
          image: k8s.gcr.io/sig-storage/csi-node-driver-registrar
          args:
            - --v=5
            - --csi-address=/csi/csi.sock
            - --kubelet-registration-path=/var/lib/kubelet/plugin
          env:
            - name: KUBE_NODE_NAME
              valueFrom:
                fieldRef:
                  apiVersion: v1
                  fieldPath: spec.nodeName
          imagePullPolicy: IfNotPresent
      volumeMounts:
        - name: plugin-dir
          mountPath: /csi
        - name: registration-dir
          mountPath: /registration
      resources:
        limits:
          cpu: 100m
          memory: 100Mi
        requests:
          cpu: 10m
          memory: 20Mi
      - name: secrets-store
        image: k8s.gcr.io/csi-secrets-store/driver:v0.1.0
```

```

args:
  - "--endpoint=$(CSI_ENDPOINT)"
  - "--nodeid=$(KUBE_NODE_NAME)"
  - "--provider-volume=/etc/kubernetes/secrets-store-cs"
  - "--metrics-addr=:8095"
  - "--enable-secret-rotation=true" #2
  - "--rotation-poll-interval=1m" #3
  - "--filtered-watch-secret=true"
  - "--provider-health-check=false"
  - "--provider-health-check-interval=2m"
env:
  - name: CSI_ENDPOINT
    value: unix:///csi/csi.sock
  - name: KUBE_NODE_NAME
    valueFrom:
      fieldRef:
        apiVersion: v1
        fieldPath: spec.nodeName
imagePullPolicy: IfNotPresent
securityContext:
  privileged: true
ports:
  - containerPort: 9808
    name: healthz
    protocol: TCP
livenessProbe:
  failureThreshold: 5
  httpGet:
    path: /healthz
    port: healthz
  initialDelaySeconds: 30
  timeoutSeconds: 10
  periodSeconds: 15
volumeMounts:
  - name: plugin-dir
    mountPath: /csi
  - name: mountpoint-dir
    mountPath: /var/lib/kubelet/pods
    mountPropagation: Bidirectional
  - name: providers-dir
    mountPath: /etc/kubernetes/secrets-store-csi-provid
resources:
  limits:
    cpu: 200m
    memory: 200Mi
  requests:
    cpu: 50m

```

```

        memory: 100Mi
    - name: liveness-probe
      image: k8s.gcr.io/sig-storage/livenessprobe:v2.3.0
      imagePullPolicy: IfNotPresent
      args:
        - --csi-address=/csi/csi.sock
        - --probe-timeout=3s
        - --http-endpoint=0.0.0.0:9808
        - -v=2
      volumeMounts:
        - name: plugin-dir
          mountPath: /csi
      resources:
        limits:
          cpu: 100m
          memory: 100Mi
        requests:
          cpu: 10m
          memory: 20Mi
    volumes:
      - name: mountpoint-dir
        hostPath:
          path: /var/lib/kubelet/pods
          type: DirectoryOrCreate
      - name: registration-dir
        hostPath:
          path: /var/lib/kubelet/plugins_registry/
          type: Directory
      - name: plugin-dir
        hostPath:
          path: /var/lib/kubelet/plugins/csi-secrets-store/
          type: DirectoryOrCreate
      - name: providers-dir
        hostPath:
          path: /etc/kubernetes/secrets-store-csi-providers
          type: DirectoryOrCreate
    nodeSelector:
      kubernetes.io/os: linux

```

Finally, deploy the Secret Store CSI driver with the modification done above:

Listing 6.33. Deploy Secret Store CSI driver with auto-rotation enabled

```
kubectl apply -f install-csi-polling.yaml #1
```

Wait until the Pod is running:

Listing 6.34. Wait until driver Pod is running

```
kubectl wait --for=condition=ready pod -l app=csi-secrets-store -
```

6.3.2 Deploy the Pod with secret mounted

Let's redeploy the Pod defined in the previous section to inject the secret rotation example:

Listing 6.35. Applying the Pod with CSI volume

```
kubectl apply -f vault-app-pod.yaml -n default #1
```

Wait until the Pod is running:

Listing 6.36. Wait until greeting Pod is running

```
kubectl wait --for=condition=ready pod -l app=greeting --timeout=
```

Finally, we validate that the secret is mounted in `/mnt/secrets-vault` as specified in `volumesMount` section, and the file containing the secret is `my-password` as set in the `objectName` field. Run the command shown in listing [6.37](#):

Listing 6.37. Read the injected secret

```
kubectl exec greeting-demo -- cat /mnt/secrets-vault/my-password  
my_secret_password% #1
```

6.3.3 Updating the secret

Let's update the key with a new secret value inside Vault by opening an interactive shell session on Vault Pod:

Listing 6.38. Opening interactive shell

```
kubectl exec -it vault-0 -- /bin/sh #1
```

From this point, the commands issued are executed on the Vault container. Let's update the secret at the path `secret/pass` with the `my_new_secret_password` value:

Listing 6.39. Updating Secret

```
vault kv put secret/pass password="my_new_secret_password" #1
```

Key	Value
--	-----
created_time	2021-08-09T17:26:26.665179027Z
deletion_time	n/a
destroyed	false
version	2 #2

Type `exit` to quit the `kubectl exec` command.

Wait at least for one minute until the Secret Store CSI drivers poll the secret, detects the change and populate it to the Pod and run the command again to print the secret value:

Listing 6.40. Read the injected secret

```
kubectl exec greeting-demo -- cat /mnt/secrets-vault/my-password  
my_new_secret_password% #1
```

Type `exit` to quit the `kubectl exec` command.

Notice that the secret value has been injected without having to restart the Pod.

We now understand how Secrets Store CSI Driver works; however, no cloud secret stores were used in the example. We'll expand this example to store secrets in a public cloud secret storage in the following section.

6.4 Consuming secrets from cloud secret stores

So far, we've used HashiCorp Vault as secret storage in this chapter and in Chapter 5. But if you're using a public cloud as a platform to deploy the

Kubernetes cluster, you might use the secret storage service they provide in their infrastructure. For example, in the case of using Azure cloud provider, you may be using Azure Key Vault as a secret store, in this section you'll see how to consume secrets from there using the secret store CSI driver.

Secret Storage CSI drivers supports AWS Secrets Manager and AWS Systems Manager Parameter Store, Azure Key Vault, and Google Secret Manager. We'll implement the same example of the previous section, but consuming the secrets from public cloud secret stores instead of HashiCorp Vault.



Important

The process of integrating a secret storage and the Secrets Store CSI looks the same in all cases:

1. Install & Configure the secret store
2. Install the Secret Store CSI Provider according the secret store used (Vault, AWS, Azure, ...)
3. Configure the `SecretProviderClass` with the secret store configuration parameters

The biggest difference is on the installation and configuration of the secret store. In this section, we'll show you how to integrate cloud secret stores to CSI, but we're assuming you have an account to the cloud providers to deploy the secret store and a basic knowledge of them.

Let's start by integrating secret store CSI driver to Azure Key Vault.

6.4.1 Azure Key Vault

Azure Key Vault is an Azure cloud service that provides a secure store for secrets (keys, passwords, certificates, ...). To run Azure Key Vaults we need an Azure account with at least free services subscription as Azure Key Vault cannot run outside of the Azure cloud.

Install and Configure Azure Key Vault

To install Azure Key Vault, log in to your Azure subscription to create a service principal with policies to access the key vault.

Listing 6.41. Azure login

```
az login #1
```

When logged to Azure, we can create a service principal executing the command shown in listing [6.42](#):

Listing 6.42. Azure creation service principal

```
az ad sp create-for-rbac --skip-assignment --name alex #1
```

The command returns a JSON document with some sensitive parameters that we'll need to configure the CSI driver later on.

```
{
  "appId": "7d3498f8-633e-4c58-bbaa5-1b2a015017a7", #1
  "displayName": "alex",
  "name": "http://alex",
  "password": "2CAT7NvT90zrLneTdi3..rYnU.M4_qGIMP", #2
  "tenant": "66ee79ad-f624-4a81-b14e-319d7dd9e699"
}
```



Tip

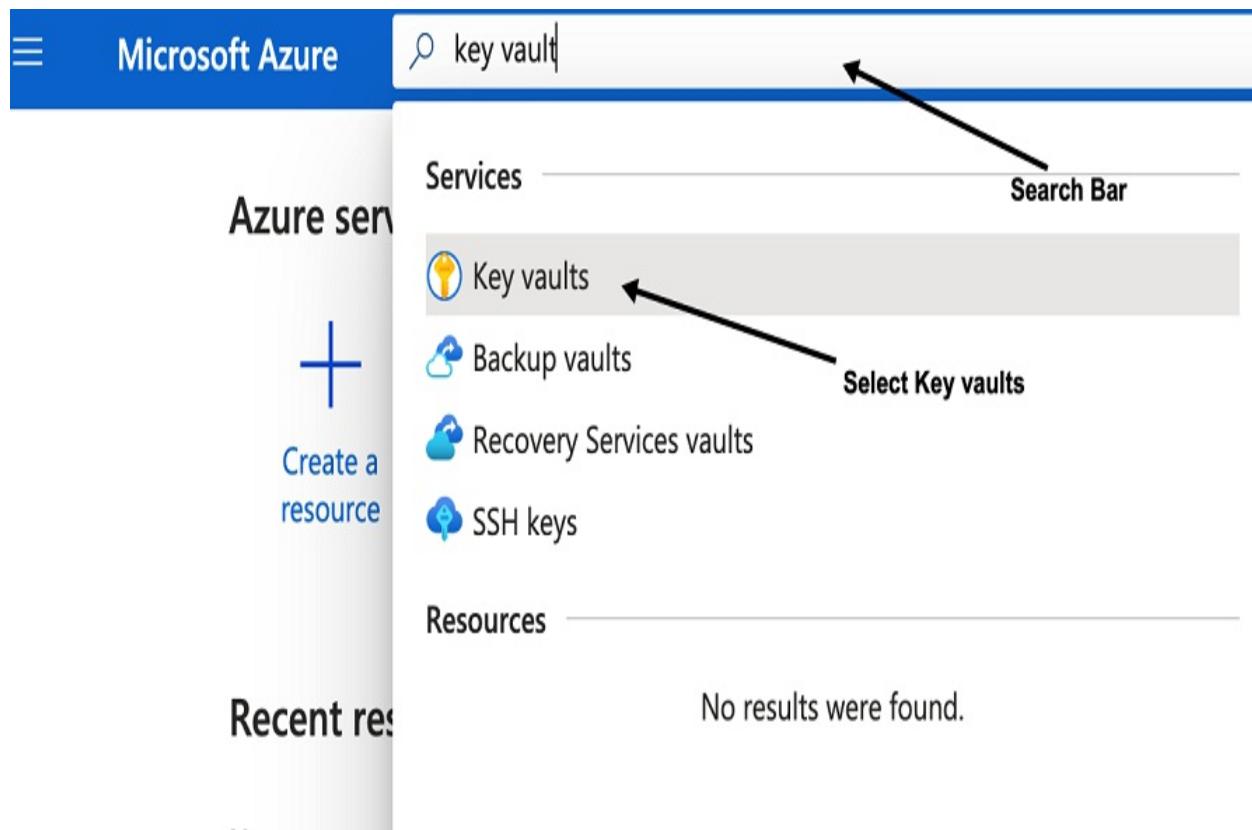
Creating a specific service principal for Azure Key Vault isn't mandatory, and we could use the default service principal.

Then let's create the Azure Key Vault through the Azure portal.

Go to <https://portal.azure.com/> and log in with your account.

In the portal, go to the top search bar, and type Key Vault to get the Azure Key Vault resource and click on it as shown in figure [6.11](#):

Figure 6.11. Azure portal



At the Key Vaults section, click on the `create key vault` button to start creating an Azure Key Vault instance.

Figure 6.12. Azure Key Vault portal. Cloud providers tweak their web interfaces on a ongoing basis.

Home >

Key vaults

Default Directory  ...

Key vaults section

+ Create  Manage deleted vaults  Manage view Refresh Export to CSV Open query Assign tags  Fe

Filter for any field... Subscription == all Resource group == all Location == all Add filter

Showing 0 to 0 of 0 records.

Name ↑↓	Type ↑↓	Resource group ↑↓	Location ↑↓
---------	---------	-------------------	-------------



No key vaults to display
Safeguard cryptographic keys and other secrets used by cloud apps and services.

Button to create a new Key vault → **Create key vault**

[Learn more](#)

Fill the Create key vault wizard with the parameters shown in figure 6.13:

Figure 6.13. Create Azure Key Vault

Microsoft Azure Search resources, services, and docs (G+)

Home > Key vaults >

Create key vault

CREATE FOR COMPLIANCE

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * (New) alex Create new

Resource group * (New) alex Create new

Instance details

Key vault name * alexvault

Region * Germany West Central Select region near you

Pricing tier * Standard

Recovery options

Soft delete protection will automatically be enabled on this key vault. This feature allows you to recover or permanently delete a key vault and secrets for the duration of the retention period. This protection applies to the key vault and the secrets stored

Review + create < Previous Next : Access policy >

The Key vault name in this example is alexvault, but it could be any other name and other region and pricing tier.

Push the Review + create button, review the values, and click on the Create button.

Then the overview of the created key vault is shown. At this point, click on the JSON View link to show the same content but in the JSON format and get the tenant ID of the key vault.

Figure 6.14. Overview Azure Key Vault

The screenshot shows the Azure Key Vault overview page for a vault named 'alexvault'. The top navigation bar includes 'Home > alexvault >'. Below it, there's a search bar and standard navigation buttons: Delete, Move, Refresh, Open in mobile. The main content area is titled 'alexvault overview' and contains the following details:

Essentials	
Resource group (change)	: alex
Location	: Germany West Central
Subscription (change)	: [REDACTED]
Subscription ID	: [REDACTED]

On the right side, there are more details:

Vault URI	: https://alexvault.vault.azure.net/
Sku (Pricing tier)	: Standard
Directory ID	: [REDACTED]
Directory Name	: Default Directory
Soft-delete	: Enabled
Purge protection	: Disabled

Below the essentials section, there's a 'Tags' section with a note: 'Tags (change) : Click here to add tags'. At the bottom left, there are links for 'Get started', 'Monitoring', 'Tools + SDKs', and 'Tutorials'. A prominent red box highlights the 'Get started' link. On the far right, there's a link 'Click here to get Key vault configuration as JSON' with an arrow pointing to a red box labeled 'JSON View'.

We need to create the secret to be injected into the Pod. Click on Secrets section on the left menu and push the Generate/Import button on the top-left menu as shown in figure [6.15](#):

Figure 6.15. Create Azure Key Vault secret

The screenshot shows the Azure Key Vault Secrets page for a vault named "alexvault". The top navigation bar includes a search bar, a "Generate/Import" button (which is highlighted with a red arrow), a "Refresh" button, a "Restore Backup" button, and a "Manage deleted secrets" button. Below the navigation is a table header with columns for "Name", "Type", and "Status". A message states "There are no secrets available." On the left side, there is a sidebar with links for "Overview", "Activity log", "Access control (IAM)", "Tags", "Diagnose and solve problems", and "Events". At the bottom of the sidebar, under "Settings", there is a list of options: "Keys", "Secrets" (which is highlighted with a red arrow and has a callout "Click on Secrets Settings"), and "Certificates".

Fill the name field with the value password and the value field with my_password and push the Create button.

Figure 6.16. Create Azure Key Vault secret

Create a secret

... Create a secret in created
Key vault

Upload options

Manual

Name * ⓘ

password

Key name

Value * ⓘ

Secret value

Content type (optional)

Set activation date ⓘ

Set expiration date ⓘ

Enabled

Yes No

Tags

0 tags

Create the secret by
pushing this button

Create



Note

A secret can be created using az CLI tool by running az keyvault secret set --vault-name "alexvault" --name "password" --value "my_password"

Finally, we need to assign the permissions from the service principal created

before to the key vault that we've just created previously. Get back to the terminal window and execute the following commands:

Listing 6.43. Assigning permissions

```
az keyvault set-policy -n $KEYVAULT_NAME --key-permissions get --
az keyvault set-policy -n $KEYVAULT_NAME --secret-permissions get
az keyvault set-policy -n $KEYVAULT_NAME --certificate-permission
```

We are now ready to return to Kubernetes and configure it accordantly to consume secrets from Azure Key Vault.

Azure Key Vault CSI Driver

Let's install and configure the Azure Key Vault CSI Driver.

First of all, we create a new Kubernetes namespace to deploy the Azure example:

Listing 6.44. Create Azure namespace

```
kubectl create namespace azure #1
kubectl config set-context --current --namespace=azure #2
```

Install the Azure Secret Store CSI Provider as shown in listing [6.45](#) to inject secrets from Azure Key Vault into Pods using the CSI interface.

Listing 6.45. Install the Azure Secret Store CSI provider

```
kubectl apply -f https://raw.githubusercontent.com/Azure/secrets-
```



Tip

In the case of Windows nodes, you need to apply the `provider-azure-installer-windows.yaml` file.

Check that the provider is running by executing the following command:

Listing 6.46. Check the Azure Secret Store CSI provider

```
kubectl get pods -n azure #1
```

NAME	READY	STATUS	RESTAR
csi-secrets-store-provider-azure-6fcdc	1/1	Running	1



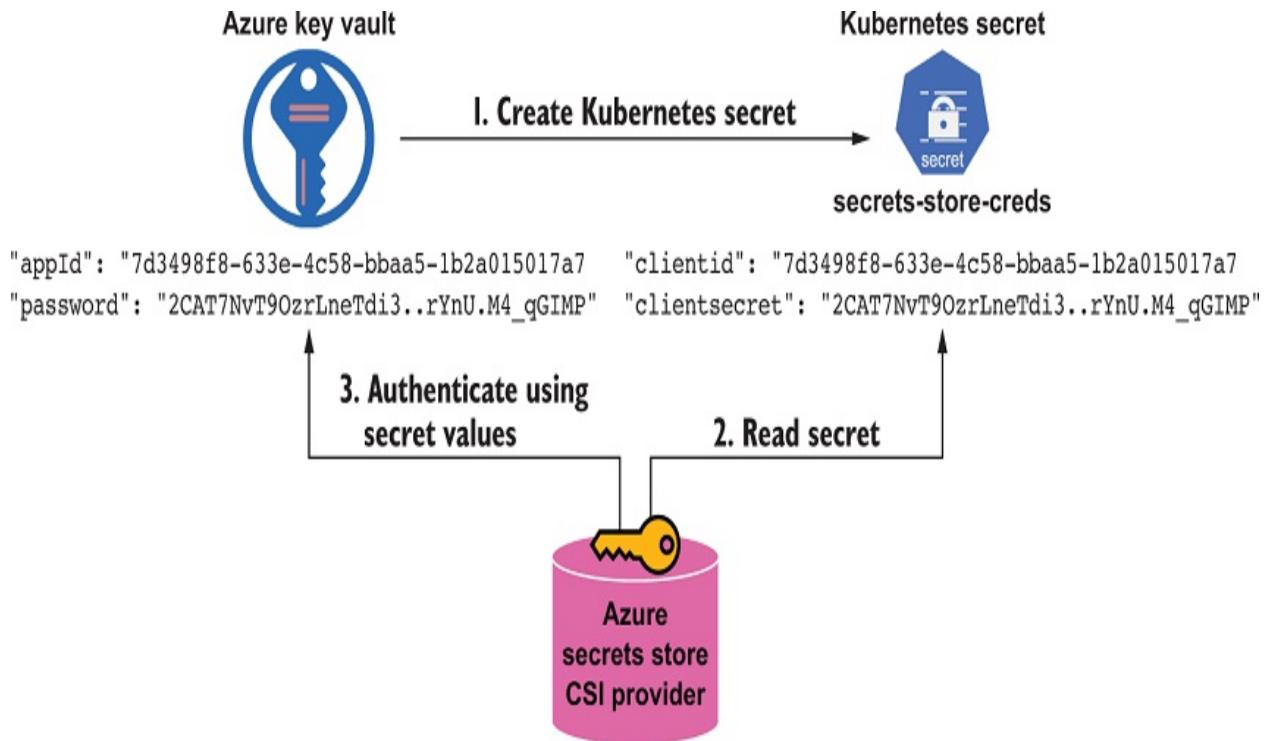
Warning

For AKS clusters, the provider needs to be installed in the `kube-system` namespace to establish connectivity to the Kube API Server.

Then we set the service principal credentials as Kubernetes secrets accessible by the Azure Secrets Store CSI driver. The driver uses this secret to log in to the remote Azure Key Vault. The secret must contain `clientid` and `clientsecret` keys set to the service principal app ID and password fields, respectively.

Figure 6.17 shows an overview of the authentication steps:

Figure 6.17. Authentication process of the Azure Secret Store CSI provider



[Listing 6.47](#) shows the creation of a Kubernetes Secrets with the Azure credentials:

Listing 6.47. Create Kubernetes secret

```
kubectl create secret generic secrets-store-creds --from-literal
```



Important

Secret must be created in the same namespace as the application Pod. Moreover, as any Kubernetes Secret, it needs to be managed as shown in Chapter 3 and 4.

Moreover, we can label the secrets (like the one created just above) used in the nodePublishSecretRef section (see below) to limit the amount of memory used by the CSI driver.

Listing 6.48. Create Kubernetes secret

```
kubectl label secret secrets-store-creds secrets-store.csi.k8s.io
```

Before creating a Pod with the secret, the last step is to configure the SecretProviderClass with the Azure Key Vault name and the Azure Key Vault tenant id.

Listing 6.49. azure-spc.yaml

```
apiVersion: secrets-store.csi.x-k8s.io/v1alpha1
kind: SecretProviderClass
metadata:
  name: azure-manning  #1
spec:
  provider: azure  #2
  parameters:
    keyvaultName: "alexvault"  #3
  objects: |
    array:
      - |
        objectName: password  #4
        objectType: secret
```

```
tenantId: "77aa79ad-f624-7623-b14e-33447dd9e699" #5
```

Apply the azure-manning SecretProviderClass by running the following command:

Listing 6.50. Applying the azure-manning SecretProviderClass

```
kubectl apply -f azure-spc.yaml -n azure #1
```

Deploy a Pod with secret mounted

The configuration is similar as we see in the HashiCorp Vault example; we've got the csi section where we set the CSI driver to secrets-store.csi.k8s.io and also the SecretProviderClass name created above (azure-manning). But in this case, we set the nodePublishSecretRef pointing out the Kubernetes secret created previously with the Azure service principal credentials (secrets-store-creds) to access the Azure Key Vault.

Listing 6.51. azure-app-pod.yaml

```
kind: Pod
apiVersion: v1
metadata:
  name: greeting-demo
  labels:
    app: greeting
spec:
  containers:
    - image: quay.io/lordofthejars/greetings-jvm:1.0.0
      name: greeting-demo
      volumeMounts: #1
        - name: secrets-store-inline
          mountPath: "/mnt/secrets-azure" #2
          readOnly: true
  volumes:
    - name: secrets-store-inline
      csi:
        driver: secrets-store.csi.k8s.io
        readOnly: true
        volumeAttributes:
          secretProviderClass: "azure-manning" #3
```

```
nodePublishSecretRef:  
  name: secrets-store-creds #4
```

Deploy the Pod by running the following command:

Listing 6.52. Applying the Pod with CSI volume

```
kubectl apply -f azure-app-pod.yaml -n azure #1
```

Wait until the Pod is running:

Listing 6.53. Wait until greeting Pod is running

```
kubectl wait --for=condition=ready pod -l app=greeting --timeout=
```

Finally, we validate that the secret is mounted in /mnt/secrets-azure as specified in the volumesMount section, and the file containing the secret is named the secret name (password).

Run the command shown in listing [6.54](#):

Listing 6.54. Read the injected secret

```
kubectl exec greeting-demo -- cat /mnt/secrets-azure/password  
my_password% #1
```

From the point of view of the Secret Store CSI, there hasn't been much difference between using Azure Key Vault or HashiCorp Vault; if we look back in this section, we'll see that most of the effort was in installing and configuring Azure Key Vault, the CSI part is similar and only takes the last paragraphs of this section.

Other configuration parameters

Azure CSI provider has other configuration parameters not shown in the previous example; in the following snippet, we can see a SecretProviderClass example with all possible parameters.

```
apiVersion: secrets-store.csi.x-k8s.io/v1alpha1
```

```

kind: SecretProviderClass
metadata:
  name: azure-kvname
spec:
  provider: azure
  parameters:
    usePodIdentity: "false" #1
    useVMManagedIdentity: "false" #2
    userAssignedIdentityID: "client_id" #3
    keyvaultName: "kvname"
    cloudName: "" #4
    cloudEnvFileName: "" #5
    objects: | #6
      array:
        - |
          objectName: secret1
          objectAlias: SECRET_1 #7
          objectType: secret #8
          objectVersion: "" #9
        - |
          objectName: key1
          objectAlias: ""
          objectType: secret
          objectVersion: ""
          objectFormat: "pem" #10
          objectEncoding: "utf-8" #11
  tenantId: "tid"

```

Provide Identity to Access Key Vault

In this section, we've provided the identity to access Key Vault using a Service Principal. When writing this book, this is the only way to connect to Azure Key Vault from a non-Azure environment. But, if you are in an Azure environment (i.e., AKS), then these other authentication modes are supported:

- AAD Pod Identity
- User-assigned Managed Identity
- System-assigned Managed Identity

We know how to use Secret Store CSI driver to inject secrets stored in Azure Key Vault. As we mentioned earlier, Secret Store CSI driver supports other providers apart from Azure.

In the following section we see the same example but storing the secret at GCP Secret Manager.

6.4.2 GCP Secret Manager

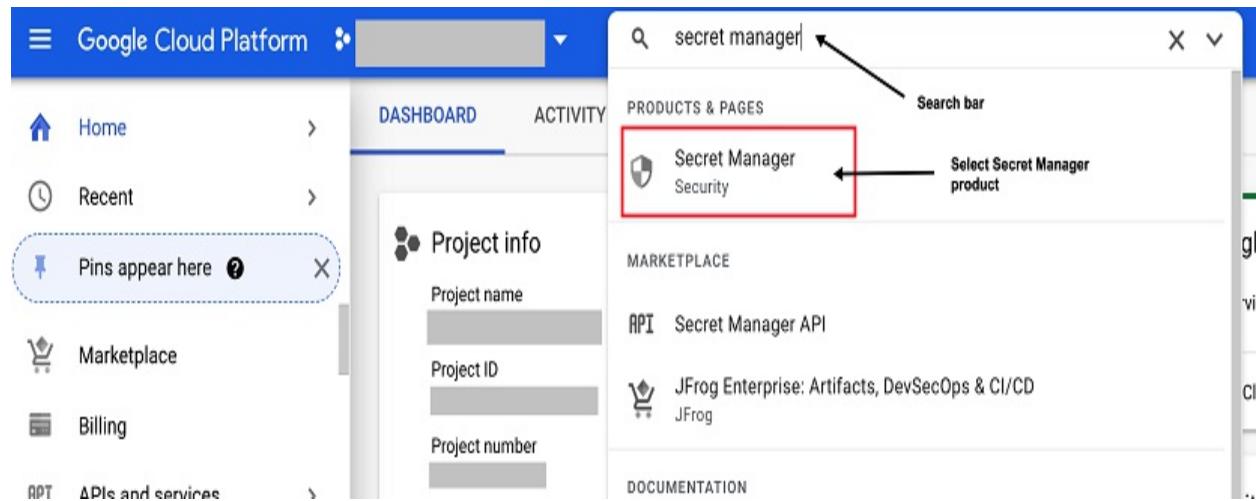
GCP Secret Manager is an Google cloud service that provides a secure store for secrets (keys, passwords, certificates, ...). To run GCP Secret Manager we need a GCP account with Secret Manager installed as it cannot run outside of the Google cloud.

Install and Configure GCP Secret Manager

To install GCP Secret Manager, log in to GCP console (<https://console.cloud.google.com/home>) with your GCP account and enable Secret Manager.

In the search bar placed at top, search for Secret Manager and click it on the search results as shown in figure 6.18;

Figure 6.18. Search for Secret Manager



Then enable the Secret Manager by clicking on the Enable button:

Figure 6.19. Enable Secret Manager



Secret Manager API

Google Enterprise API

Stores sensitive data such as API keys, passwords, and certificates. Provides convenience while...



With Secret Manager enabled, we create a new secret to be injected into the Pod using CSI. Select Secret Manager resource from Security section in the left menu, and push the Create Secret button as shown in figure 6.20:

Figure 6.20. Create Secret

The screenshot shows the Google Cloud Platform interface under the 'Security' section. A search bar at the top right contains the text 'secret'. An annotation points from the text 'Search Bar' to the search input field. On the left sidebar, there is a list of security services. An annotation points from the text 'Click here to access to Secret Manager section' to the 'Secret Manager' item, which is highlighted with a red border. Another annotation points from the text 'Finally creates a new secret by clicking this link' to the '+ CREATE SECRET' button.

- Security
- Security Command Center
- reCAPTCHA Enterprise
- BeyondCorp Enterprise
- Identity-Aware Proxy
- Access Context Manager
- VPC Service Controls
- Binary Authorization
- Data Loss Prevention
- Key Management
- Certificate Authority Servi...
- Secret Manager**

Finally creates a new secret by clicking this link

+ CREATE SECRET

Click here to access to Secret Manager section

Search Bar

secret

Fill the create secret form with Name to app-secret and Value to my_password as shown in figure 6.21:

Figure 6.21. Create Secret

Create secret

Secret details

Set the secret name

This will create a secret with the secret value in the first version. [Learn more](#)

Name

app-secret

The name should be identifiable and unique within this project.

Secret value

Input your secret value or import it directly from a file.

Upload file

BROWSE

Maximum size: 64 KiB

sets the secret value

Secret value

my_password

A list of all create secrets is shown when we create a secret. Then click on the name of the created secret to inspect the details of the secret and get the resource name as we need later on for the GCP Secret Store CSI provider configuration:

Figure 6.22. Select Created Secret

The screenshot shows the Google Cloud Secret Manager interface. At the top, there is a blue header bar with a dropdown menu, a search icon, and the text "secret manager". Below the header, the title "Secret Manager" is displayed next to a "CREATE SECRET" button. A tooltip message says: "Try accessing secrets in the IDE using Cloud Code. [Learn more](#)". The main area contains a brief description: "Secret Manager lets you store, manage, and secure access to your application" followed by a link "[Learn more](#)". Below this, there is a "Filter" input field with the placeholder "Enter property name or value". A table lists secrets:

	Name	Location	Encryption
<input type="checkbox"/>	app-secret	Automatically replicated	Google-managed

A red box highlights the "app-secret" name in the table. An arrow points from the text "Click to enter to secret details" above the table towards the "app-secret" row.

Write down the Resource ID value (`projects/466074950013/secrets/app-secret`) as we need it as a parameter later on when the `SecretProviderClass` is created. We see an example of an overview of a secret in the figure [6.23](#):

Figure 6.23. Overview of the Created Secret

Secret: "app-secret"

projects/466074950013/secrets/app-secret

SECRET DETAILS

OVERVIEW **VERSIONS** **PERMISSIONS**

Name	app-secret
Replication policy	Automatically replicated
Encryption	Google-managed
Rotation	Not scheduled
Notifications	None
Labels	None
Created on	August 5, 2021 at 11:10:30 AM GMT+2
Expiration	Never
Resource ID	projects/466074950013/secrets/app-secret

Resource ID required by Secret Store CSI Driver

The final step before leaving GCP console is to export GCP service account credential keys to authenticate against the Secret Manager instance. Typically this JSON file is downloaded automatically when the keys are added to the service account. In case you don't have any key, you add a new one by clicking on Service Accounts menu and then Add Key button as shown in figure [6.24](#):

Figure 6.24. Overview of the Created Secret

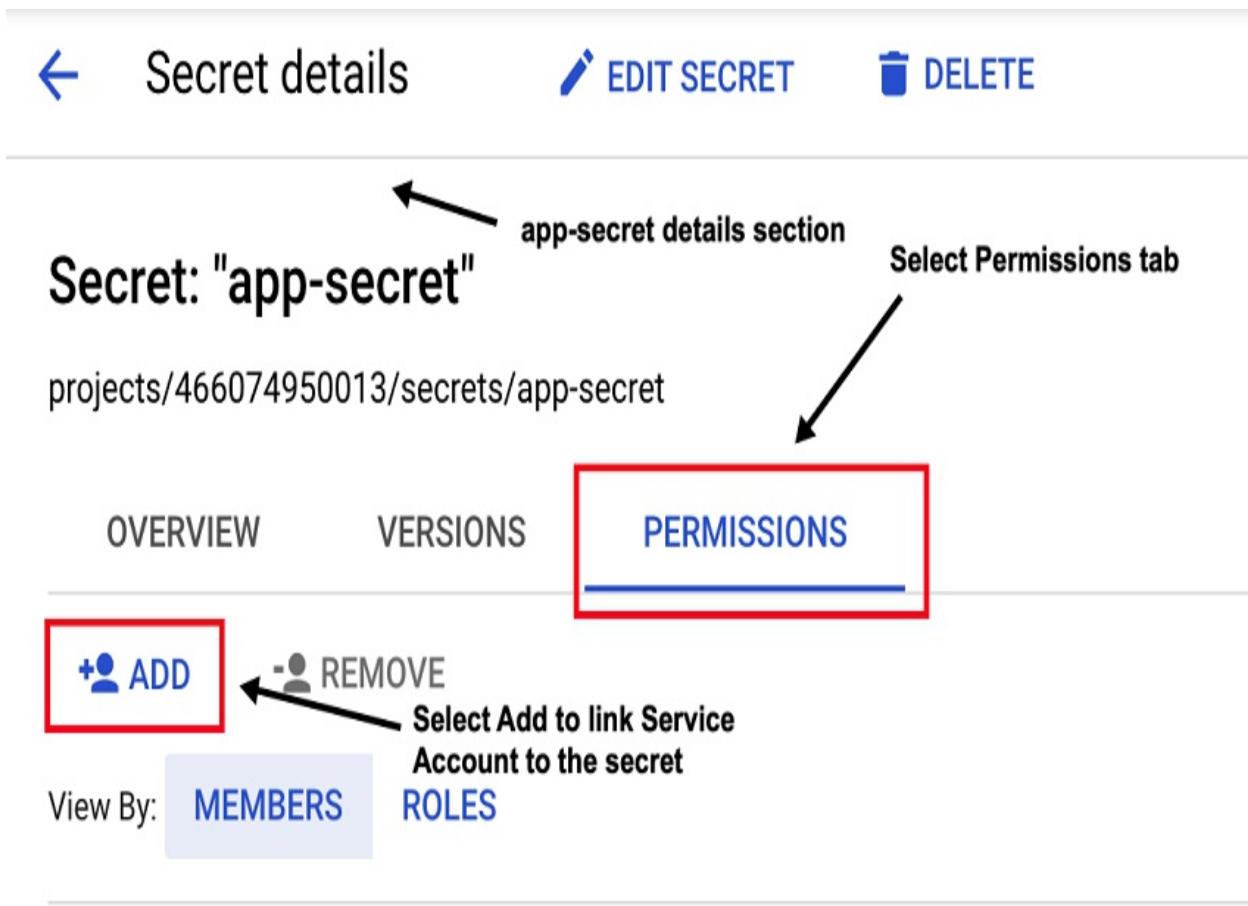
The screenshot shows the Google Cloud Platform IAM & Admin interface for managing service accounts. The left sidebar lists various options: IAM, Identity & Organisation, Policy troubleshooter, Policy Analyser, Organisation Policies, **Service Accounts**, Workload Identity Federat..., Labels, Tags, and Manage resources. The 'Service Accounts' option is highlighted with a red box and has an annotation pointing to it with the text 'Click on Service Accounts section'. The main content area shows the 'App Engine default service account' under the 'DETAILS' tab. The 'KEYS' tab is selected and highlighted with a blue underline. An annotation points to the 'KEYS' tab with the text 'Select Keys tab'. A warning message in a grey box states: 'Service account keys could pose a security risk if compromised. We recommend that you avoid downloading service account keys using [Workload Identity Federation](#). You can learn more about the best way to authenticate service accounts on Google Cloud [here](#)'. Below the message, there's a note: 'Add a new key pair or upload a public key certificate from an existing key pair.' and 'Block service account key creation using [organisation policies](#). [Learn more about setting organisation policies for service accounts](#)'. A red box highlights the 'ADD KEY' button, and an annotation points to it with the text 'Click here to create a new access key'. A table below shows a single key entry: Type (Service account), Status (Active), Key creation date (8 Aug 2021), Key expiry date (1 Jan 10000), and a trash icon.

Type	Status	Key	Key creation date	Key expiry date
Service account	Active	[Redacted]	8 Aug 2021	1 Jan 10000

It's important to generate a key in JSON format and with enough permissions to access to the Secret Manager.

Finally, go back to the secret overview page, click on Permissions tab and Add button to add the previous service account as an account that can consume the secret. This is shown at the figure [6.25](#):

Figure 6.25. Give Service Account permissions to access the Secret



GCP Secret Manager CSI Driver

Let's install and configure the GCP Secret Manager CSI Driver.

First of all, we create a new Kubernetes namespace to deploy the GCP example:

Listing 6.55. Create GCP namespace

```
kubectl create namespace gcp #1  
kubectl config set-context --current --namespace=gcp #2
```

Install the GCP Secret Store CSI Provider as shown in [6.56](#) to inject secrets from GCP Secret Manager into Pods using the CSI interface.

Listing 6.56. Install the GCP Secret Store CSI provider

```
kubectl apply -f https://raw.githubusercontent.com/GoogleCloudPla
```

Check that the provider is running by executing the following command:

Listing 6.57. Check the GCP Secret Store CSI provider

```
kubectl get pods -n kube-system #1
```

NAME	READY	STATUS	RESTARTS
coredns-558bd4d5db-4l9tk	1/1	Running	4
csi-secrets-store-8xlcn	3/3	Running	11
csi-secrets-store-provider-gcp-62jb5	1/1	Running	0



Warning

The provider is installed in the `kube-system` namespace to establish connectivity to the Kube API Server.

Then we set the service account credentials as Kubernetes secrets accessible by the GCP Secrets Store CSI driver. The driver uses this secret to log in to the remote GCP Secret Manager. The secret must have a key `key.json` with a value of an exported GCP service account credential.

Listing 6.58. Create Kubernetes secret namespace

```
KEY_GCP='{"private_key_id": "123", "private_key": "a-secret", "tok  
kubectl create secret generic secrets-store-creds --from-literal
```



Important

Secret must be created in the same namespace as the application Pod. Moreover, as any Kubernetes Secret, it needs to be managed as shown in Chapter 3 and 4.

Before creating a Pod with the secret, the last step is to configure the `SecretProviderClass` with the secret Resource ID and the file name where the content of the secret is written.

Listing 6.59. gcp-spc.yaml

```
apiVersion: secrets-store.csi.x-k8s.io/v1alpha1
kind: SecretProviderClass
metadata:
  name: gcp-manning
spec:
  provider: gcp
  parameters:
    secrets: |
      - resourceName: projects/466074950013/secrets/app-secret/ve
        fileName: app-secret #2
```

Apply the gcp-manning SecretProviderClass by running the following command:

Listing 6.60. Applying the gcp-manning SecretProviderClass

```
kubectl apply -f gcp-spc.yaml -n gcp #1
```

Deploy a Pod with secret mounted

The configuration is similar as we see in the HashiCorp Vault example; we've got the csi section where we set the CSI driver to secrets-store.csi.k8s.io and also the SecretProviderClass name created above (gcp-manning). But in this case, we set the nodePublishSecretRef pointing out the Kubernetes secret created previously with the GCP service account credentials (secrets-store-creds) to access the GCP Secret Manager.

Listing 6.61. gcp-app-pod.yaml

```
kind: Pod
apiVersion: v1
metadata:
  name: greeting-demo
  labels:
    app: greeting
spec:
  containers:
    - image: quay.io/lordofthejars/greetings-jvm:1.0.0
      name: greeting-demo
      volumeMounts: #1
```

```
- name: secrets-store-inline
  mountPath: "/mnt/secrets-gcp" #2
  readOnly: true
volumes:
- name: secrets-store-inline
  csi:
    driver: secrets-store.csi.k8s.io
    readOnly: true
    volumeAttributes:
      secretProviderClass: "gcp-manning" #3
    nodePublishSecretRef:
      name: secrets-store-creds #4
```

Deploy the Pod by running the following command:

Listing 6.62. Applying the Pod with CSI volume

```
kubectl apply -f gcp-app-pod.yaml -n gcp #1
```

Wait until the Pod is running:

Listing 6.63. Wait until greeting Pod is running

```
kubectl wait --for=condition=ready pod -l app=greeting --timeout=
```

Finally, we validate that the secret is mounted in /mnt/secrets-gcp as specified in the volumesMount section, and the file containing the secret is named app-secret as specified in fileName field.

Run the command shown in listing [6.64](#):

Listing 6.64. Read the injected secret

```
kubectl exec greeting-demo -- cat /mnt/secrets-gcp/app-secret
my_secret% #1
```

Provide Identity to GCP Secret Manager

In this section, we've provided the identity to access GCP Secret Manager using a Service Account. When writing this book, this is the only way to

connect to GCP Secret Manager from a non-GCP environment. But these other authentication modes are supported:

- Pod Workload Identity
- GCP Provider Identity

We know how to use Secret Store CSI driver to inject secrets stored in GCP Secret Manager.

In the following section we see the same example but storing the secret at AWS Secret Manager.

6.4.3 AWS Secret Manager

AWS Secret Manager is an AWS cloud service that provides a secure store for secrets (keys, passwords, certificates, ...). To run AWS Secret Manager we need an AWS account with AWS Secret Manager installed as it cannot run outside of the AWS cloud. Moreover, another requirement of AWS Secret Store CSI provider is that at time of writing this book, it may run in an Amazon Elastic Kubernetes Service (EKS) 1.17+.

We aren't going to explain the whole process of preparing an EKS cluster, and we'll assume that EKS cluster is up and configured to be used and eksctl CLI tool isntalled on your machine.

Create Secret in AWS Secret Manager

As we've done in the previous sections, we need to store a secret into the secret store, in this case AWS Secret Manager, to be consumed by the secret store CSI driver.

We use aws CLI tool (<https://docs.aws.amazon.com/cli/latest/userguide/installing.html>) to create the secret named AppSecret and with value my_secret.

Listing 6.65. Create Secret

```
REGION=us-east-2 #1
```

```
aws --region "$REGION" secretsmanager create-secret --name AppSe
```

Then we create an IAM access policy to access the secret created in the previous step. This step is important as we'll associate this access policy to the Kubernetes service account running the greeting-demo Pod in a similar way used in the HashiCorp Vault Kubernetes authentication mode.

Run the command shown in listing [6.66](#) to create an access policy with name the greeting-deployment-policy to access to the `AppSecret`secret. The policy name is important as we'll need to make a link between the policy and the Kubernetes service account.

Listing 6.66. Create Access Policy

```
$(aws --region "$REGION" --query Policy.Arn --output text iam cre
  "Version": "2012-10-17",
  "Statement": [ {
    "Effect": "Allow", #2
    "Action": ["secretsmanager:GetSecretValue", "secretsmanag
      "Resource": ["arn:aws:secretsmanager:*::secret:AppSecret-?
    } ]
  }
})
```

arn:aws:iam::aws:policy/greeting-deployment-policy #5

We need an IAM OIDC provider for the cluster to create the association between the the IAM access policy and the Kubernetes service account, if you don't have already done, create one by running the following command:

Listing 6.67. Create IAM OIDC provider

```
eksctl utils associate-iam-oidc-provider --region="$REGION" --clu
```

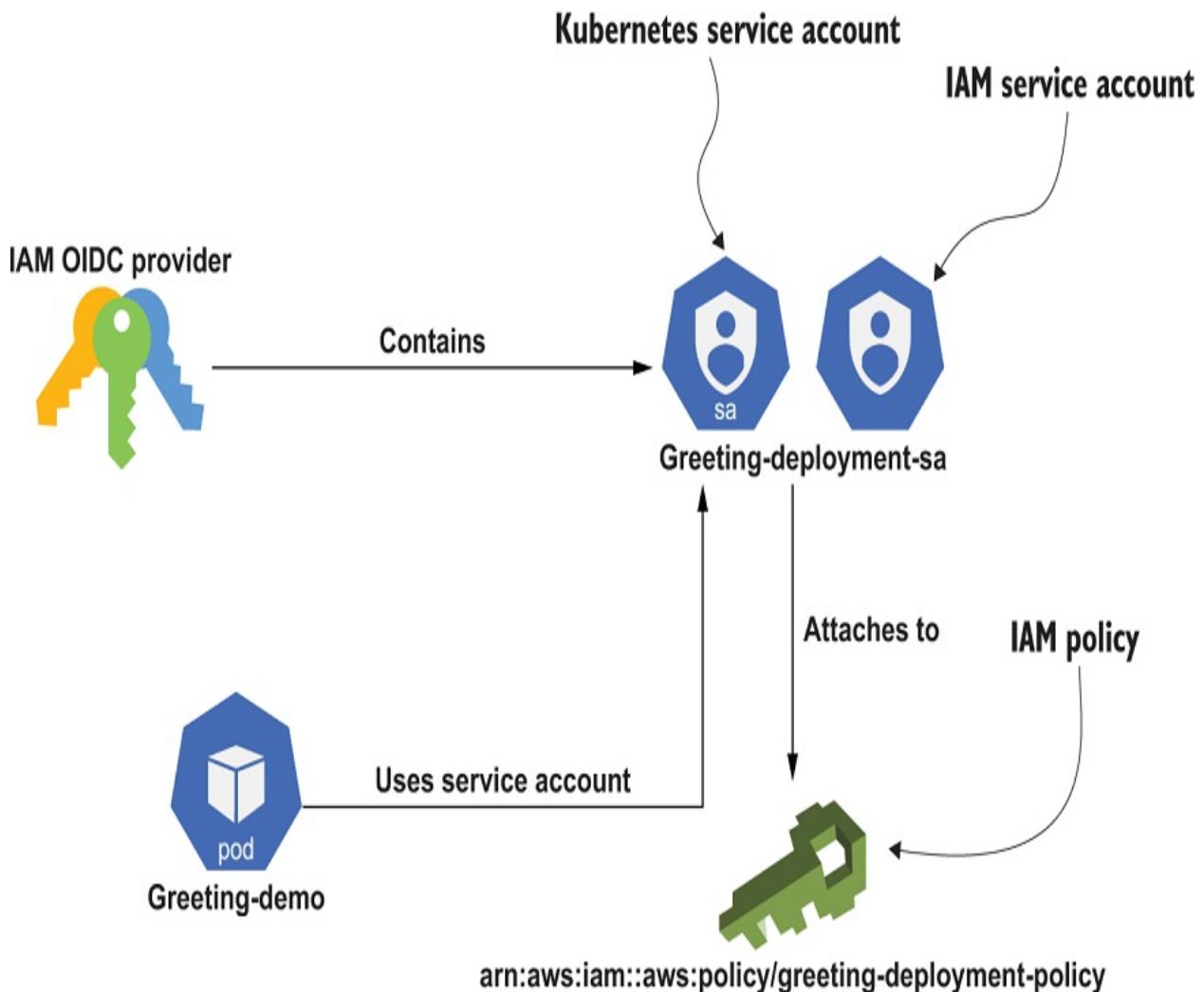
Finally, we create an IAM service account to be used by the Pod, and associate it to the IAM access policy (arn:aws:iam::aws:policy/greeting-deployment-policy) created above. In this case, we use greeting-deployment-sa as the service account name, and we'll use it as a Kubernetes Service Account name too.

Listing 6.68. Create Access Policy

```
eksctl create iamserviceaccount \
--name greeting-deployment-sa #1
--region="$REGION" --cluster "$CLUSTERNAME" \
--attach-policy-arn "arn:aws:iam::aws:policy/greeting-deploym
--approve --override-existing-serviceaccounts
```

The process is summarized in the figure [6.26](#):

Figure 6.26. Pod-Volume connection



AWS Secret Manager CSI Driver

Let's install and configure the AWS Secret Manager CSI Driver.

First of all, we create a new Kubernetes namespace to deploy the AWS

example:

Listing 6.69. Create AWS namespace

```
kubectl create namespace aws #1  
kubectl config set-context --current --namespace=aws #2
```

Install the AWS Secret Store CSI Provider as shown in listing [6.70](#) to inject secrets from AWS Secret Manager into Pods using the CSI interface.

Listing 6.70. Install the AWS Secret Store CSI provider

```
kubectl apply -f https://raw.githubusercontent.com/aws/secrets-st
```

Check that the provider is running by executing the following command:

Listing 6.71. Check the GCP Secret Store CSI provider

```
kubectl get pods -n kube-system #1
```

NAME	READY	STATUS	RESTARTS
coredns-558bd4d5db-419tk	1/1	Running	4
csi-secrets-store-8xlcn	3/3	Running	11
csi-secrets-store-provider-aws-34bj1	1/1	Running	0



Warning

The provider is installed in the `kube-system` namespace to establish connectivity to the Kube API Server.

Before creating a Pod with the secret, the last step is to configure the `SecretProviderClass` with the secret name.

Listing 6.72. aws-spc.yaml

```
apiVersion: secrets-store.csi.x-k8s.io/v1alpha1  
kind: SecretProviderClass  
metadata:  
  name: aws-manning
```

```
spec:  
  provider: aws  
  parameters:  
    objects: |  
      - objectName: "AppSecret" #1  
        objectType: "secretsmanager"
```

Apply the aws-manning SecretProviderClass by running the following command:

Listing 6.73. Applying the aws-manning SecretProviderClass

```
kubectl apply -f aws-spc.yaml -n aws #1
```

Deploy a Pod with secret mounted

The configuration is similar as we see in the HashiCorp Vault example; we've got the csi section where we set the CSI driver to secrets-store.csi.k8s.io and also the SecretProviderClass name created above (aws-manning). But in this case, we set the service account name pointing out to the service account created in the previous step (greeting-deployment-sa) to access the AWS Secret Manager.

Listing 6.74. aws-app-pod.yaml

```
kind: Pod  
apiVersion: v1  
metadata:  
  name: greeting-demo  
  labels:  
    app: greeting  
spec:  
  serviceAccountName: greeting-deployment-sa  
  containers:  
    - image: quay.io/lordofthejars/greetings-jvm:1.0.0  
      name: greeting-demo  
      volumeMounts: #1  
        - name: secrets-store-inline  
          mountPath: "/mnt/secrets-aws" #2  
          readOnly: true  
  volumes:  
    - name: secrets-store-inline
```

```
csi:  
  driver: secrets-store.csi.k8s.io  
  readOnly: true  
  volumeAttributes:  
    secretProviderClass: "aws-manning" #3
```

Deploy the Pod by running the following command:

Listing 6.75. Applying the Pod with CSI volume

```
kubectl apply -f aws-app-pod.yaml -n aws #1
```

Wait until the Pod is running:

Listing 6.76. Wait until greeting Pod is running

```
kubectl wait --for=condition=ready pod -l app=greeting --timeout=
```

Finally, we validate that the secret is mounted in /mnt/secrets-aws as specified in the volumesMount section, and the file containing the secret is named as set in objectName.

Run the command shown in listing [6.77](#):

Listing 6.77. Read the injected secret

```
kubectl exec greeting-demo -- cat /mnt/secrets-aws/AppSecret  
my_secret% #1
```

Other configuration parameters

AWS CSI provider has other configuration parameters not shown in the previous example; in the following snippet, we can see a SecretProviderClass example with all possible parameters.

```
apiVersion: secrets-store.csi.x-k8s.io/v1alpha1  
kind: SecretProviderClass  
metadata:  
  name: aws-manning  
spec:
```

```

provider: aws
parameters:
  objects: |
    - objectName: "AppSecret"
      objectType: "secretsmanager" #1
      objectAlias: "secret" #2
      objectVersion: "latest" #3
      objectVersionLabel: "latest" #4

```

Global Security Considerations when using Secret Store CSI driver

As we mention in Chapter 2, mounting secrets as volumes or inject them as environment variables have some threats that we might consider:

- When the secret is mounted on the filesystem, potential vulnerabilities like directory traversal attack, unforbidden access to node disk, or SSH'd the Pod can become a problem as the attacker may gain access to the secret data. We need to protect against these problems at the application level (against directory traversal) and the Kubernetes level (disable kubectl exec, ...).
- When the secret is injected through environment variables, potential vulnerabilities like logging environment data at the application level or SSH'd the Pod can become a problem as the attacker may read the secret data.
- When syncing secrets to the Kubernetes Secrets store, remember to apply all the security considerations learned in Chapter 4. You may have the secrets securely placed in the external secret store but lose all these confidentialities when moved to Kubernetes secrets.

There are different tools/projects that may help us to detect security threats automatically and provide some hints to solve the security issue. In our opinion there are two tools that combined together may help us to detect and audit security misconfigurations as well as unusual behaviour in our containers.

The first project is KubeLinter (<https://docs.kubelinter.io/>). The project is a static code analysis tool that analyzes Kubernetes YAML files and Helm charts checking for security misconfigurations and best practices. Some of detected issues are running containers in privileged mode, exposing

privileged ports, exposing SSH port, unsetting resource requirements, or reading secrets from environment variables.

The second project is Falco (<https://falco.org/>) This project works at runtime parsing Linux system calls from the Kernel and checking them against a list of rules to validate if they are permitted or not. In the case of a violation of a rule, an alert is triggered and some actions can be taken as a response. Falco comes with a set of rules, some of them are notifying read/writes to well-known directories such as /etc, /usr/bin, /usr/sbin, ownership and Mode changes, or executing SSH binaries such as ssh, scp, sftp, ...

We've seen that the CSI interface and Secret Store CSI drivers are a perfect abstraction for dealing with multiple secret managers.

6.5 Summary

- Container Storage Interface (CSI) is an initiative to unify the storage interface of Container Orchestrators.
- Secret Store CSI is an implementation of the CSI spec to consume secrets from external data stores.
- Secret Store CSI let us inject secrets into Pods from HashiCorp Vault, AWS Secret Manager, GCP Secret Manager, and Azure Key Vault.
- Although Secret Store CSI supports key rotation, the application needs to support it by either watching disk changes or reloading the Pod.

7 Kubernetes-Native Continuous Integration and Secrets

This chapter covers

- Integrating the application for any change using Continuous Integration methodology.
- Implementing Continuous Integration pipelines with Kubernetes-Native Tekton.
- Testing, building, and pushing a Linux container to an external registry with a Kubernetes-Native CI pipeline.

In the previous chapter we've seen how to inject secrets from a secret store to containers. Looking back to all previous chapters, we've learned how to keep secrets secret in the different phases of the lifecycle of an application, it's now time to sum-up of everything and start applying all these security concepts together.

We'll demonstrate how to implement a Kubernetes-Native Continuous Integration pipeline to release an application/service continuously and automatically, yet keeping the secrets secret using Tekton.

What we want to achieve in this chapter is to show how to deliver quality applications rapidly to hit the market sooner and better yet managing the secrets correctly during the whole pipeline so no secrets-leak occurs in this phase of the development.

7.1 Introduction to Continuous Integration

Developing software isn't an individual task, but a team task with a lot of people involved working together and concurrently to create an application. Integrating all the work done by each developer at the end of the process might not be the best strategy as several problems may be found like a merge

hell, components not integrating correctly together, breaking some parts that were working before, ... The best integration strategy to follow is integrating as much and as soon as possible, so any error is detected quickly, and locate and fix them more easily.

Continuous Integration (or *CI*) is a set of practices that automates the integration of code changes from multiple developers into a single repository. The commits to the repository must occur frequently (usually several times per day) and it must trigger an automated process to verify the correctness of the new code.

The ultimate goal of CI is to establish a steady and automated way to build, package, and test applications so any change on the source code is integrated fast, not waiting weeks, and validated after the commit, therefore any break in the integration process is detected in the early stages.

For every commit, the code should run in the following stages:

- **Build**

The code is *compiled*, and *packaged*. The output depends on the platform/language used to develop the application, in the case of Java, it can be a *JAR* or *WAR* file, in Go a binary executable.

- **Test**

The application runs the first batch of tests. These tests aren't end-to-end tests or long tests, but unit tests, some component tests, and a minimal subset of end-to-end tests validating the green path of core business functionalities.

- **Security Checks**

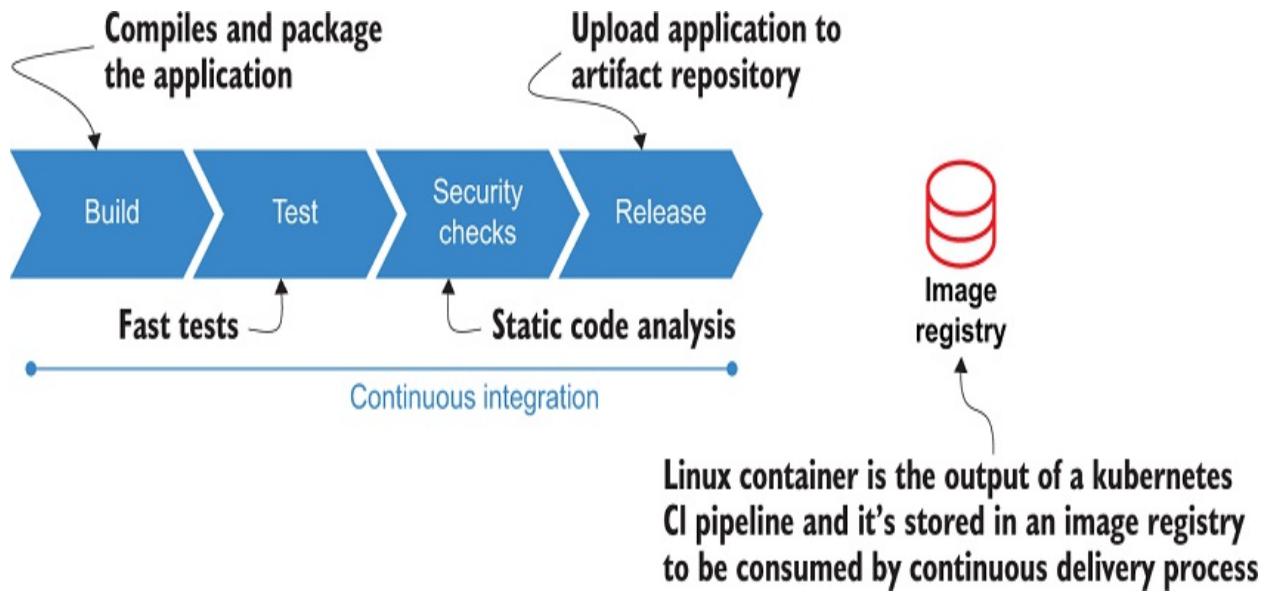
In this stage, the code is analyzed, searching for vulnerabilities or bad practices typically using static code analysis tools.

- **Release**

The delivery artifact is published in an artifact repository. It can be the *JAR* file, the Go executable, or a Linux container.

Figure [7.1](#) summarizes all these steps in a continuous integration pipeline:

Figure 7.1. Common steps that compose a Continuous Integration pipeline



The benefits of Continuous Integration are:

- Integration bugs are detected in the early stages and are easy to fix as the original code didn't change it that much.
- Application is integrated continuously avoiding spending weeks or months integrating all the pieces.
- When a test failure is detected, it's easier to find the cause (as only small changes are done) and in the case of rolling back to a previous version, only a small number of features are lost.
- Since the application is integrated frequently, there is always a version ready for deploying (or releasing) to any of the environments (staging, preproduction, production).



Important

Continuous Integration pipeline must provide quick feedback meaning that it must not take more than 10 minutes as the main goal of this pipeline is to provide fast feedback to developer, notifying any integration error as soon as possible.

After this brief introduction to Continuous Integration, it's time to implement it in a Kubernetes native way using Tekton.

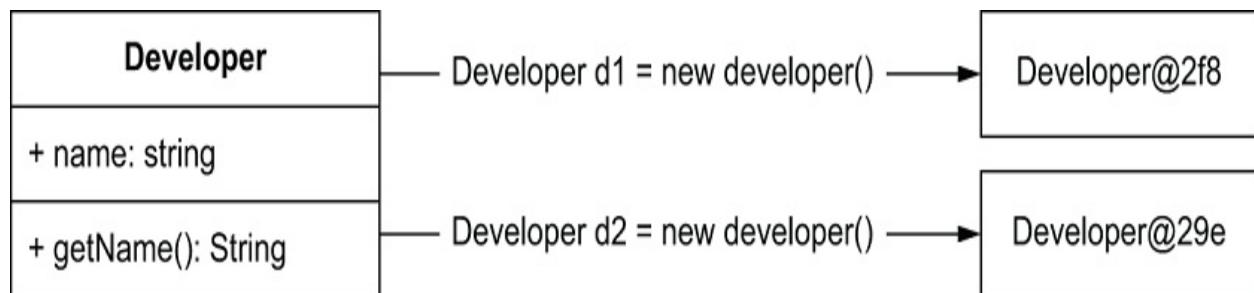
7.2 Tekton

We'll implement a simple Continuous Integration pipeline for a Java application using Tekton, a Kubernetes-native framework to create cloud-native CI pipelines. This pipeline will build the application, containerize it and finally push the container to a container registry.

Tekton (<https://tekton.dev/>) is an open-source, Kubernetes-native project for building Continuous Integration/Continuous Delivery pipelines providing Custom Resource Definitions (*CRDs*) that defines the building blocks we can create and reuse in our pipelines.

The Tekton *CRDs* can be grouped into two big blocks one group that represents any element that define a pipeline, and another group that represents the pipeline execution. If this separation of elements confuses you, we like to do the analogy with classes and instances in programming languages. The class object is the definition of a concept, while an instance of the class is the real object in memory, with specific parameters and can be instantiated multiple times. A developer class definition and creation of two instances of the class is shown in the figure [7.2](#):

Figure 7.2. Class definition vs class instance



Before installing Tekton, we need to create a Kubernetes cluster and deploy a Git server and a container registry.

7.2.1 Installing pre-requisites

Let's start a new minikube instance by running the following command in a terminal window:

Listing 7.1. Start Minkube

```
minikube start -p argo --kubernetes-version='v1.19.0' --vm-driver
```

In this chapter, we'll need a Git repository with writing permissions to some repositories. To avoid relying on an external service (GitHub, GitLab, ...), we're going to deploy a Git server (<https://gitea.io/en-us/>) into the Kubernetes cluster.

Listing 7.2. Deploy Gitea

```
kubectl apply -f https://gist.githubusercontent.com/lordofthejars
```

And wait until Gitea deployment is up and running:

Listing 7.3. Wait until Gitea is ready

```
kubectl wait --for=condition=ready pod -l app=gitea-demo --timeou
```

Git server is accessible within the Kubernetes cluster through the gitea DNS name. Register a new user to the system with the username gitea and password gitea1234 owning the source code used in the chapter.

Listing 7.4. Create Gitea user

```
kubectl exec svc/gitea > /dev/null -- gitea admin create-user --u
```

Finally, the source code used is migrated from GitHub to the internal Git server.

Listing 7.5. Migrate application to Gitea

```
kubectl exec svc/gitea > /dev/null -- curl -i -X POST -H "Content
```

Moreover, a container registry is required to store the containers build during the CI phase. To avoid relying on external service, we're going to deploy a container registry (<https://docs.docker.com/registry/>) into the Kubernetes cluster.

Listing 7.6. Install Docker registry

```
kubectl apply -f https://gist.github.com/lordofthejars
```

And wait until Registry deployment is up and running:

Listing 7.7. Wait until registry is ready

```
kubectl wait --for=condition=ready pod -l app=registry-demo --tim
```

Container images are pulled by Kubernetes nodes, which means that DNS names used in Kubernetes Services are not valid in the physical machines (in this example inminikube node). To make containers pushed to registry be pullable from nodes, we need to add to the Kubernetes node an entry to /etc/hosts with the DNS name and the Kubernetes service IP of registry service.

Get the registry service IP by running the following command:

Listing 7.8. Get registry IP

```
kubectl get service/registry -o jsonpath='{.spec.clusterIP}' #1  
10.111.129.197
```

Then let's access to minikube machine and add this entry to /etc/hosts:

Listing 7.9. Register Registry IP to host

```
minikube ssh -p argo  
  
sudo -i  
echo "10.111.129.197 registry" >> /etc/hosts #1  
  
exit  
exit
```

We are now ready to install Tekton in the Kubernetes cluster.

7.2.2 Installation of Tekton

Let's install Tekton 0.20.1 applying the listing [7.10](#). This command will install all Role Base Access Control (RBAC), Custom Resource Definitions (CRDs), ConfigMaps and Deployments to use Tekton.

Listing 7.10. Install Tekton

```
kubectl apply --filename https://github.com/tektoncd/pipeline/rel  
namespace/tekton-pipelines created  
podsecuritypolicy.policy/tekton-pipelines created  
clusterrole.rbac.authorization.k8s.io/tekton-pipelines-controller  
clusterrole.rbac.authorization.k8s.io/tekton-pipelines-controller  
clusterrole.rbac.authorization.k8s.io/tekton-pipelines-webhook-cl  
role.rbac.authorization.k8s.io/tekton-pipelines-controller create  
role.rbac.authorization.k8s.io/tekton-pipelines-webhook created  
role.rbac.authorization.k8s.io/tekton-pipelines-leader-election c  
serviceaccount/tekton-pipelines-controller created  
serviceaccount/tekton-pipelines-webhook created  
clusterrolebinding.rbac.authorization.k8s.io/tekton-pipelines-con  
clusterrolebinding.rbac.authorization.k8s.io/tekton-pipelines-con  
clusterrolebinding.rbac.authorization.k8s.io/tekton-pipelines-web  
rolebinding.rbac.authorization.k8s.io/tekton-pipelines-controller  
rolebinding.rbac.authorization.k8s.io/tekton-pipelines-webhook cr  
rolebinding.rbac.authorization.k8s.io/tekton-pipelines-controller  
rolebinding.rbac.authorization.k8s.io/tekton-pipelines-webhook-le  
customresourcedefinition.apiextensions.k8s.io/clustertasks.tekton  
customresourcedefinition.apiextensions.k8s.io/conditions.tekton.d  
customresourcedefinition.apiextensions.k8s.io/images.caching.inte  
customresourcedefinition.apiextensions.k8s.io/pipelines.tekton.de  
customresourcedefinition.apiextensions.k8s.io/pipelineruns.tekton  
customresourcedefinition.apiextensions.k8s.io/pipelineresources.t  
customresourcedefinition.apiextensions.k8s.io/runs.tekton.dev cre  
customresourcedefinition.apiextensions.k8s.io/tasks.tekton.dev cr  
customresourcedefinition.apiextensions.k8s.io/taskruns.tekton.dev  
secret/webhook-certs created  
validatingwebhookconfiguration.admissionregistration.k8s.io/valid  
mutatingwebhookconfiguration.admissionregistration.k8s.io/webhook  
validatingwebhookconfiguration.admissionregistration.k8s.io/confi  
clusterrole.rbac.authorization.k8s.io/tekton-aggregate-edit creat  
clusterrole.rbac.authorization.k8s.io/tekton-aggregate-view creat  
configmap/config-artifact-bucket created  
configmap/config-artifact-pvc created  
configmap/config-defaults created  
configmap/feature-flags created  
configmap/config-leader-election created  
configmap/config-logging created
```

```
configmap/config-observability created
configmap/config-registry-cert created
deployment.apps/tekton-pipelines-controller created
service/tekton-pipelines-controller created
horizontalpodautoscaler.autoscaling/tekton-pipelines-webhook crea
poddisruptionbudget.policy/tekton-pipelines-webhook created
deployment.apps/tekton-pipelines-webhook created
service/tekton-pipelines-webhook created
```



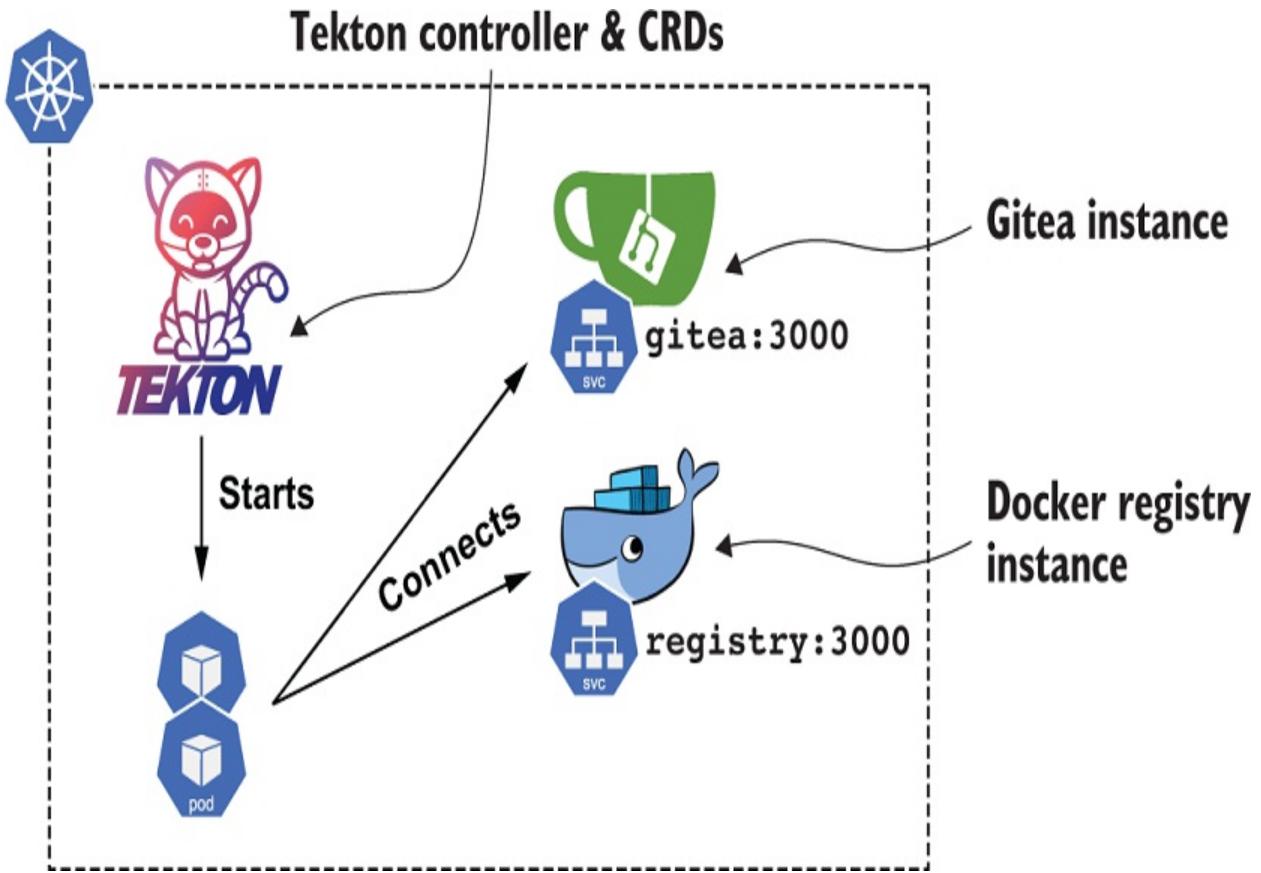
Tip

Tekton CLI is command-line utility used to interact with Tekton resources. Although it isn't mandatory to install it, it helps a lot especially to view what's going on in the pipeline.

To install it, visit <https://github.com/tektoncd/cli/releases/tag/v0.16.0>, download the package for your platform, uncompress it, and copy the `tkn` file into a `PATH` directory so it's accessed from anywhere.

The overall picture of what we've installed so far is seen in the figure [7.3](#):

Figure 7.3. Services deployed inside the cluster (Tekton, SCM and Container Registry)



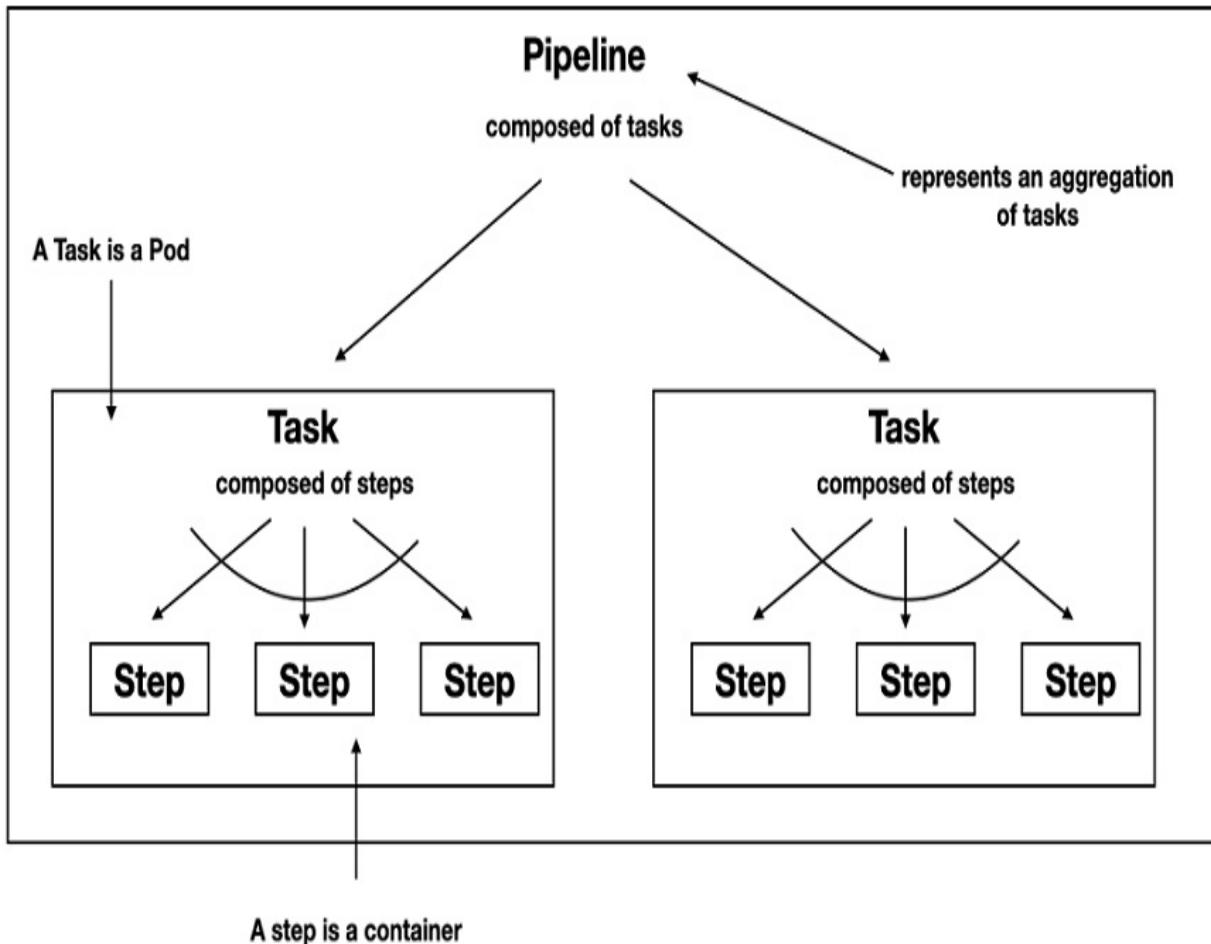
At this point, we're ready to start learning and using Tekton in the Kubernetes cluster.

7.2.3 Tekton Pipelines

In summary, Tekton provides two group of Kubernetes objects to define and execute pipelines.

The first group is a collection of Kubernetes objects for **defining** the tasks and steps used to compose a CI pipeline. The most important objects are Pipeline composed of Tasks that at the same time are composed of Steps as shown in figure [7.4](#) diagram.

Figure 7.4. Relationship between Tekton elements



The second group is a collection of Kubernetes objects for instantiating tasks and pipelines. The most important objects are `PipelineRun`, `TaskRun` and `Triggers`. The latter one is not covered in this book as it is out-of-scope, but triggers enable the execution of a pipeline because of an external event, for example, a commit to the source repository.

Pipeline Resource

A Pipeline Resource is a Kubernetes object that defines a set of resources used as input and output parameters to a Task.

Examples of input resources are a Git repository or a container image. Examples of output resources are a container image or a file.

To set the URL of the Git repository, we create a `PipelineResource` setting

the type to git and the url parameter to the Git Repository location as shown in listing [7.11](#):

Listing 7.11. build-resources.yaml

```
apiVersion: tekton.dev/v1alpha1
kind: PipelineResource
metadata:
  name: git-source
spec:
  type: git #1
  params:
    - name: url #2
      value: https://github.com/lordofthejars/kubernetes-secrets-
```

The created Pipeline Resource is named git-source. We'll refer to it later.

Step

A step represents an operation in the pipeline, for example, *compile an application, run tests or build a Linux container image*. Each step is executed within a provided container image. Any step can mount volumes or use environment variables.

A step is defined in the steps section where you set the name of the step, the container image used in the step, and the command to execute inside that container. Furthermore, we can set the directory where the command is run by using the workingDir attribute.

An example of building a Java application using Apache Maven is shown in listing [7.12](#).

Listing 7.12. build-app-task.yaml

```
steps:
  - name: maven-build
    image: docker.io/maven:3.6-jdk-11-slim #1
    command: #2
      - mvn
    args:
      - clean
```

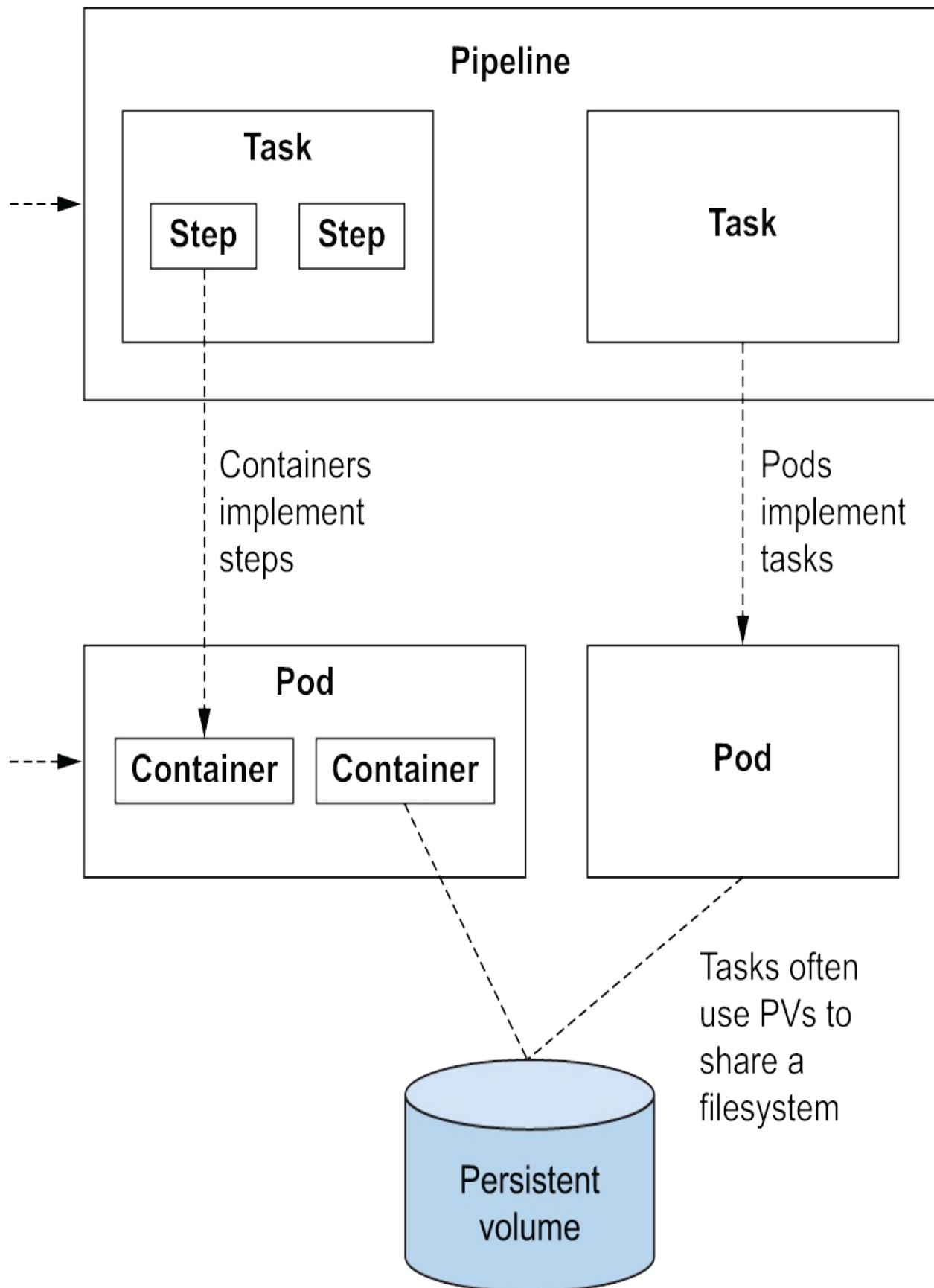
```
- package
workingDir: "/workspace/source/${inputs.params.contextDir}"
```

Task

A Task is a Kubernetes object composed of a list of **steps** in order. Each task is executed in a Kubernetes Pod, where each step is a running container in the pod.

Since all containers within a Pod share resources (CPU, disk, memory), and a task is composed of several steps (containers) running in the same Pod, anything written in the disk by one step is accessible inside any step of the task.

Figure 7.5. Tekton Task, Pod & Container relationship



A Task is configured in the spec section, where you set the list of steps to execute, and optional configuration parameters like input parameters, the input, and output resources required by the task, or volumes.

An example of a Task registering the step defined in the previous section, defining the input parameter for the `workingDir` attribute, and defining an input resource of type git to clone a repository before any step is executed is shown in listing [7.13](#).

Listing 7.13. build-app-task.yaml

```
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: build-app
spec:
  params: #1
    - name: contextDir
      description: the context dir within source
      default: . #2
  resources:
    inputs:
      - name: source #3
        type: git
  steps:
    - name: maven-build #4
    ...
```

Arrived at this point you might be wondering 2 things:

1. Where do we set the Git project repository?
2. Where is the project cloned?

For the first question, the Git repository is configured externally in a `PipelineResource` object.

The second question is easier. The content is cloned at the `/workspace/<name>` directory where name is the *input name* value given at the `git` type. Hence, the Git resource defined previously is cloned at the `/workspace/source` directory.

A Task is just the definition, or the description of the steps to execute, to execute it we need to create a TaskRun object.

TaskRun

A Task Run is a Kubernetes object that instantiate and execute a Tekton Task on cluster. A TaskRun executes each of the steps defined in the Task with the order defined until all of them are executed.

To execute the build-app Task created previously, a TaskRun object is required referencing the Task and setting up the input parameters and resources with specific values as shown in listing 7.14.

Listing 7.14. build-app-task-run.yaml

```
apiVersion: tekton.dev/v1beta1
kind: TaskRun
metadata:
  name: build-app-run
spec:
  params:
    - name: contextDir #1
      value: name-generator
  resources:
    inputs:
      - name: source #2
        resourceRef:
          name: git-source
  taskRef:
    name: build-app #3
```



Note

You might wondering why contextDir needs to be set to a specific value and not leave it with its default value (the root of the repository)?

The reason is how the <https://github.com/lordofthejars/kubernetes-secrets-source.git> repository is organized. If you take a close look at the directory hierarchy, you'll notice that the repository contains both services (*name* and *welcome message*) each one in a directory:

```
greetings
├── README.md
├── mvnw
├── mvnw.cmd
├── pom.xml
└── src
    └── target
name-generator #1
├── README.md
├── mvnw
├── mvnw.cmd
├── pipelines
├── pom.xml
└── src
    └── target
welcome-message
├── README.md
├── mvnw
├── mvnw.cmd
├── pom.xml
└── src
    └── target
└── vault-init
```

Since we are building the *Name Generator* service, we set the Maven's working directory to name-generator.

TaskRun is the way to execute a single task, sometimes we might use them to execute/test a specific task, but most of the times we want to execute the full pipeline with all tasks defined on it.

Pipeline

A Pipeline is a Kubernetes object composed of a list of **tasks** connected to them in a directed acyclic graph.

In the Pipeline definition, we have full control on tasks execution order and conditions, making it possible to set up fan-in/fan-out scenarios for running tasks in parallel or setting up conditions to a Task that should meet before executing it.

Let's create a simple Pipeline using the `build-app` Task created in the previous section.

As with the tasks, a Pipeline can have input parameters and input resources making the pipeline extendable. For this specific example, only the input parameter (Git resource) is configurable from outside the pipeline, the `contextDir` parameter value is hardcoded in the task. Finally, `build-app` task is registered as a pipeline task with input parameter and resource set.

The Pipeline definition should be similar to the one shown in listing [7.15](#).

Listing 7.15. pipeline-name-app.yaml

```
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: name-generator-deployment
spec:
  resources:
    - name: appSource #1
      type: git
  tasks:
    - name: build-app
      taskRef:
        name: build-app #2
      params: #3
        - name: contextDir
          value: name-generator
      resources:
        inputs: #4
          - name: source
            resource: appSource
```

So far, we've seen how to define a Continuous Integration pipeline using Tekton, but no execution has happened yet, as the pipeline needs to be instantiated and input parameters/resources need to be provided. In the following section, we'll see how to execute a Tekton pipeline.

PipelineRun

A Pipeline Run is a Kubernetes object to instantiate and execute a Tekton

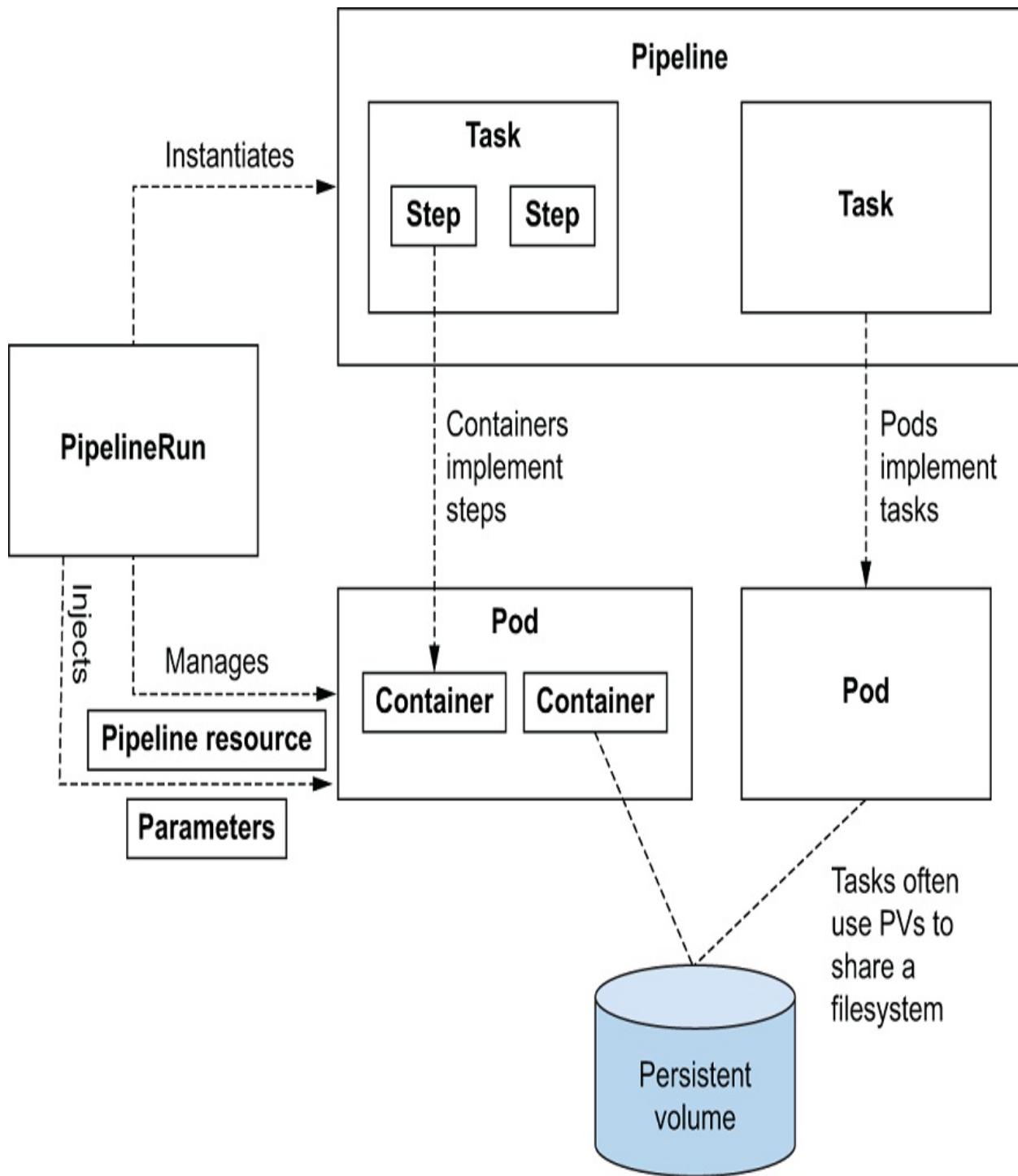
Pipeline on a cluster. A PipelineRun executes each of the defined tasks in the Pipeline, creating automatically a TaskRun for each of them as shown in listing 7.16.

Listing 7.16. pipeline-run-name-app.yaml

```
apiVersion: tekton.dev/v1beta1
kind: PipelineRun
metadata:
  name: build-app-pipeline
spec:
  resources:
    - name: appSource #1
      resourceRef:
        name: git-source
  pipelineRef:
    name: name-generator-deployment #2
```

Figure 7.6 summarizes the basic Tekton elements and how they are inter-related together.

Figure 7.6. Relationship between a PipelineRun and the Tekton resources



We've seen the most important Tekton resources to build a basic Continuous Integration pipeline, but this is far from a real pipeline.

7.3 Continuous Integration for Welcome Message

A real Continuous Integration pipeline in Kubernetes needs at least the following steps:

- Checkout of the code using a Tekton Git resource.
- Define an Apache Maven container in a Tekton step to build and test the application.
- Set container registry credentials as Kubernetes secrets, and define a Buildah container in a Tekton step to build and push the container.

The application used in this chapter is a simple service architecture composed of two services producing a welcome message.

- **name generator service**

It is a service that randomly selects a name from a list of names as shown in listing [7.17](#):

Listing 7.17. NameGeneratorResource.java

```
@Path("/generate")
public class NameGeneratorResource {

    private static final String[] NAMES = new String[] {
        "Ada", "Alexandra", "Burr", "Edson", "Kamesh", "Sebi", "Anna"
    }; #1

    private static final Random r = new Random();

    @GET
    @Path("/name")
    @Produces(MediaType.TEXT_PLAIN)
    @RolesAllowed("Subscriber") #2
    public String generate() {
        return NAMES[generateRandomIndex()]; #3
    }
}
```

- **welcome message service**

It is a service that randomly chooses the welcome message from a database and delegates to *name service* the name of the person you dedicate the greeting as shown in listing [7.18](#):

Listing 7.18. WelcomeResource.java

```

@Path("/welcome")
public class WelcomeResource {

    @RestClient
    NameService nameService; #1

    @ConfigProperty(name = "name-service-token") #2
    String token;

    private static Random r = new Random();

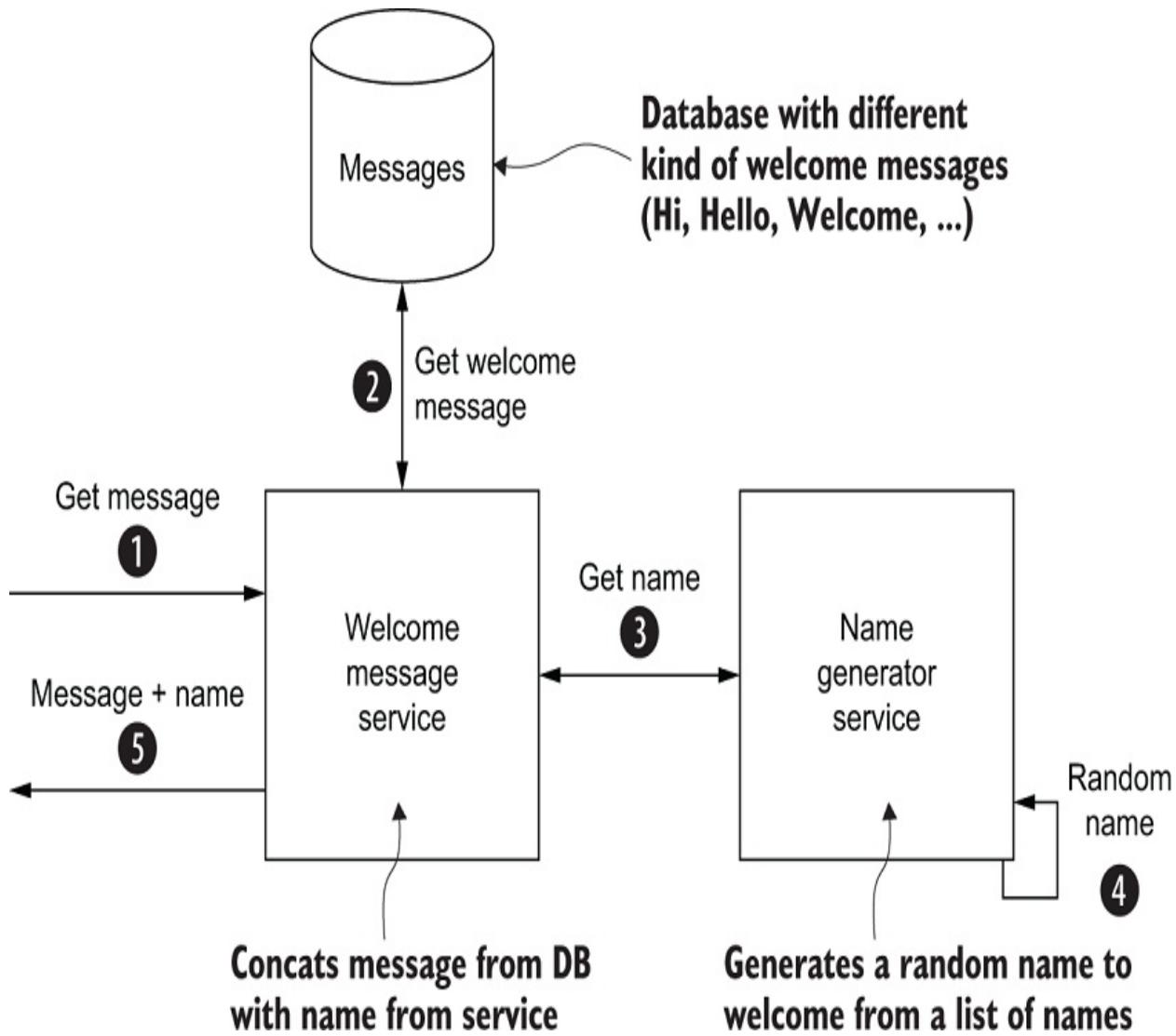
    @GET
    @Path("/message")
    @Produces(MediaType.TEXT_PLAIN)
    public String hello() {
        String welcomeMessage = randomMessage(Welcome.listAll()); #3
        String name = nameService.generateName("Bearer " + token); #4

        return welcomeMessage + " " + name;
    }
}

```

Figure [7.7](#) shows an overview of the application:

Figure 7.7. Overview of the interactions between Welcome and Name services

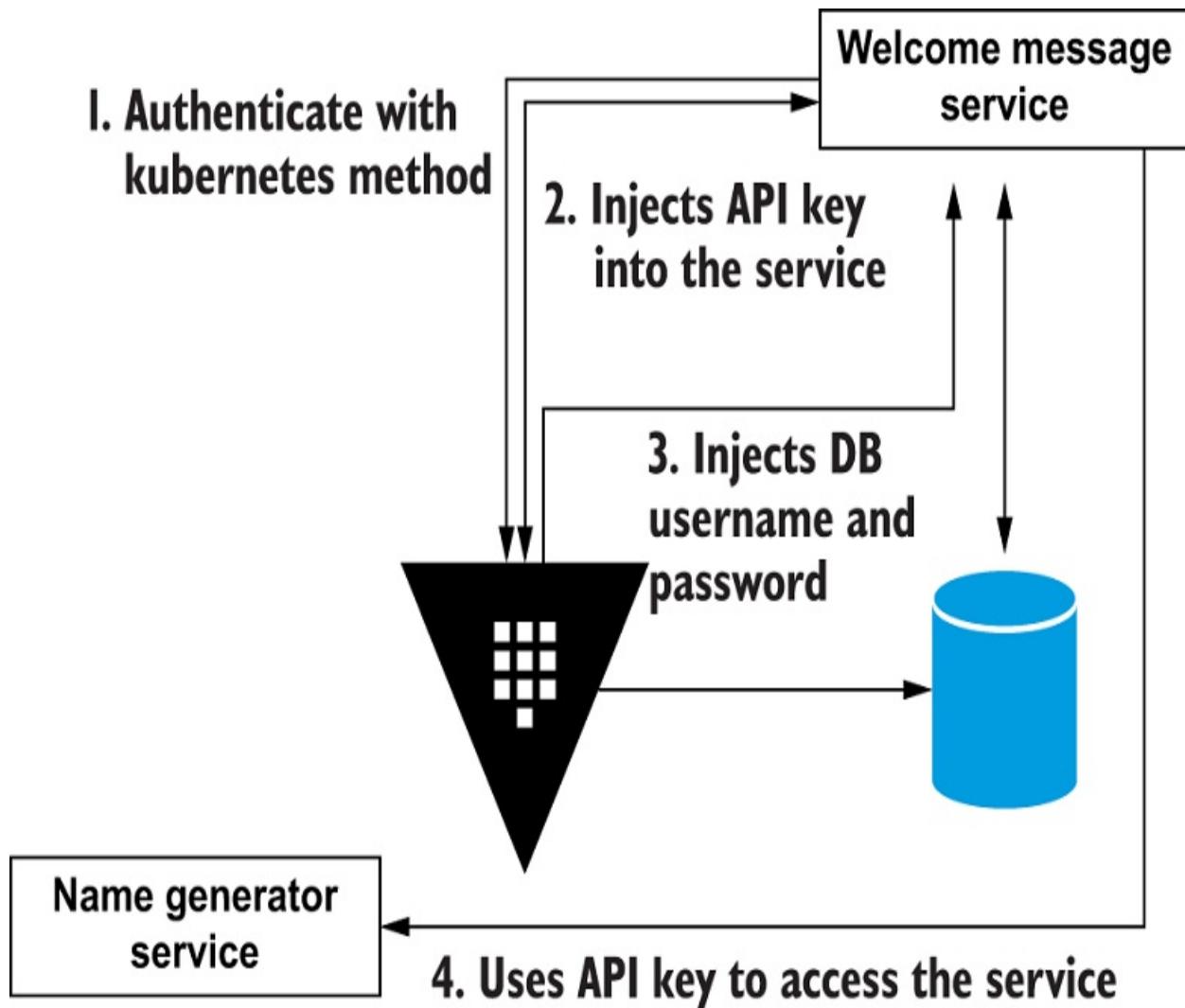


From the security point of view, the following considerations are taken:

- We need to provide an API key to access the *Name Generator* service. This API key is a secret and is stored in the Hashicorp Vault instance.
- Database credentials of *Welcome Message* service are managed by Hashicorp Vault Dynamic Database Credentials.
- Services authenticate against Hashicorp Vault using Kubernetes authentication method.

Figure 7.8 shows an overview of these elements:

Figure 7.8. Security elements

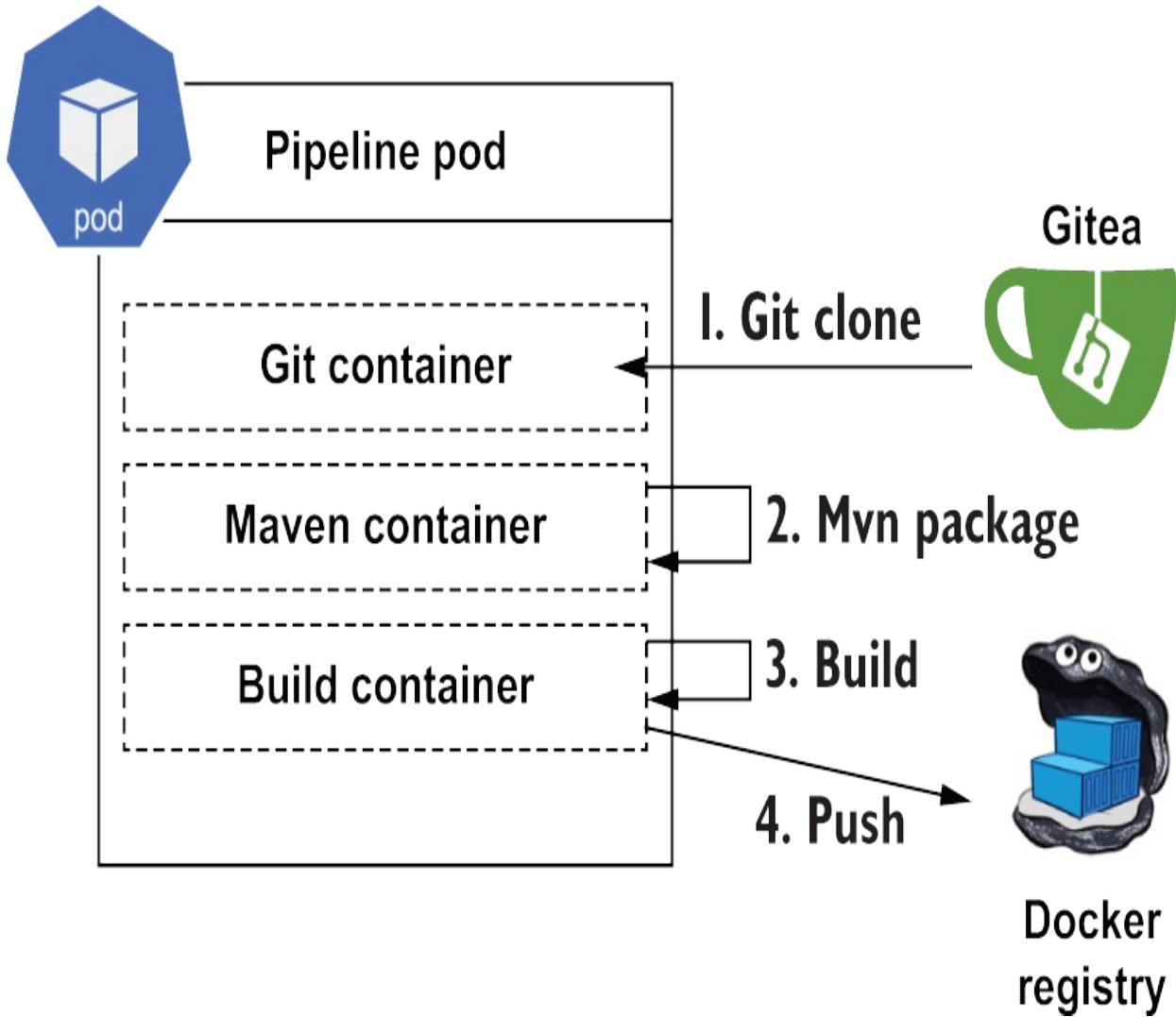


We'll assume that you have some experience with CI/CD, as well as basic knowledge of Git and Linux containers. The principles described in this chapter apply to whichever technology you may end up choosing.

The pipeline execution in terms of Tekton and Kubernetes elements is shown in the figure [7.9](#). A Pod is created with three containers, the first one clones the project from the Gitea server, the project is packaged, and finally in the third container the Linux container with the service is built and pushed to Container registry.

The three containers with the commands that are executed are shown in the figure [7.9](#):

Figure 7.9. List of containers run inside a Pod for Welcome Message service



Each of these steps are implemented as a Tekton step.

7.3.1 Compile and Run tests

We've already seen how to compile and run tests in the previous section using Apache Maven.

Welcome Message service is developed in Java and Apache Maven is used as a building tool.

Listing 7.19. Build service Tekton step

- name: maven-build

```

image: docker.io/maven:3.6.3-jdk-11-slim
command:
  - mvn
args:
  - -DskipTests
  - clean
  - install
workingDir: "/workspace/source/$(inputs.params.contextDir)"

```

7.3.2 Build and Push the container image

Build a container image inside a running container (remember each step is executed inside a container) is a bit complicated because a Docker daemon is required to build a container image. To avoid having to deal with the Docker inside Docker problem, or to build container images within environments where we can't run easily a Docker host such as a Kubernetes cluster, there are some *dockerless* tools that permit building container images without depending on a Docker daemon.

Buildah (<https://buildah.io/>) is a tool to build container images from a `Dockerfile`, inside a container without requiring a Docker daemon.

In the step definition shown in listing 7.20, Buildah is used to build and push the *Welcome Message* container to the container registry. The container name in the form of `registry:`group`:`name`:`tag`` and the location of the `Dockerfile` are provided as parameters.

Listing 7.20. Build and push container image Tekton step

```

- name: build-and-push-image
  image: quay.io/buildah/stable
  script: | #1
    #!/usr/bin/env bash
    buildah bud --layers -t $DESTINATION_IMAGE $CONTEXT_DIR #2
    buildah push --tls-verify=false $DESTINATION_IMAGE docker://$#
  env: #4
    - name: DESTINATION_IMAGE
      value: "$(inputs.params.imageName)"
    - name: CONTEXT_DIR
      value: "/workspace/source/$(inputs.params.contextDir)"
  securityContext: #5
  runAsUser: 0

```

```

    privileged: true
volumeMounts: #6
- name: varlibc
  mountPath: /var/lib/containers

```

Create a new file named `welcome-service-task.yaml` containing both steps defined previously as shown in listing [7.21](#):

Listing 7.21. welcome-service-task.yaml

```

apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: welcome-service-app #1
spec:
  params: #2
    - name: contextDir #3
      description: the context dir within source
      default: .
    - name: imageName #4
      description: the container name url-group-artifact-tag
  resources:
    inputs:
      - name: source
        type: git #5
  steps:
    - name: maven-build #6
      image: docker.io/maven:3.6.3-jdk-11-slim
      command:
        - mvn
      args:
        - clean
        - install
      workingDir: "/workspace/source/${inputs.params.contextDir}"
    - name: build-and-push-image #7
      image: quay.io/buildah/stable
      script: |
        #!/usr/bin/env bash
        buildah bud --layers -t $DESTINATION_IMAGE $CONTEXT_DIR
        buildah push --tls-verify=false $DESTINATION_IMAGE docker
      env:
        - name: DESTINATION_IMAGE
          value: "$(inputs.params.imageName)"
        - name: CONTEXT_DIR
          value: "/workspace/source/${inputs.params.contextDir}"

```

```
  securityContext:
    runAsUser: 0
    privileged: true
  volumeMounts:
    - name: varlibc
      mountPath: /var/lib/containers
  volumes:
    - name: varlibc
      emptyDir: {}
```

Execute the following command to register the Task into the Kubernetes cluster:

Listing 7.22. Register task

```
kubectl apply -f welcome-service-task.yaml #1
```

7.3.3 Pipeline Resource

Welcome Message service repository is stored in the local Git server (Gitea) deployed in the Kubernetes cluster.

Let's set the Git location of the service in a Pipeline Resource.

Create a new file named `welcome-service-resource.yaml` as shown in listing [7.23](#):

Listing 7.23. welcome-service-resource.yaml

```
apiVersion: tekton.dev/v1alpha1
kind: PipelineResource
metadata:
  name: welcome-service-git-source
spec:
  type: git
  params:
    - name: url
      value: http://gitea:3000/gitea/kubernetes-secrets-source.gi
```

Execute the following command to register the Pipeline Resource into the Kubernetes cluster:

Listing 7.24. Register pipeline resource

```
kubectl apply -f welcome-service-resource.yaml #1
```

7.3.4 Pipeline

The last step is defining a pipeline to implement the Continuous Integration pipeline for the *Welcome Message* service. Create a new file named `welcome-service-pipeline.yaml` as shown in listing [7.25](#):

Listing 7.25. welcome-service-pipeline.yaml

```
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: welcome-deployment
spec:
  resources:
    - name: appSource
      type: git
  params:
    - name: imageTag
      type: string
      description: image tag
      default: v1
  tasks:
    - name: welcome-service-app
      taskRef:
        name: welcome-service-app
      params:
        - name: contextDir
          value: welcome-message
        - name: imageName
          value: "registry:5000/k8ssecrets/welcome-message:${(para
  resources:
    inputs:
      - name: source
        resource: appSource
```

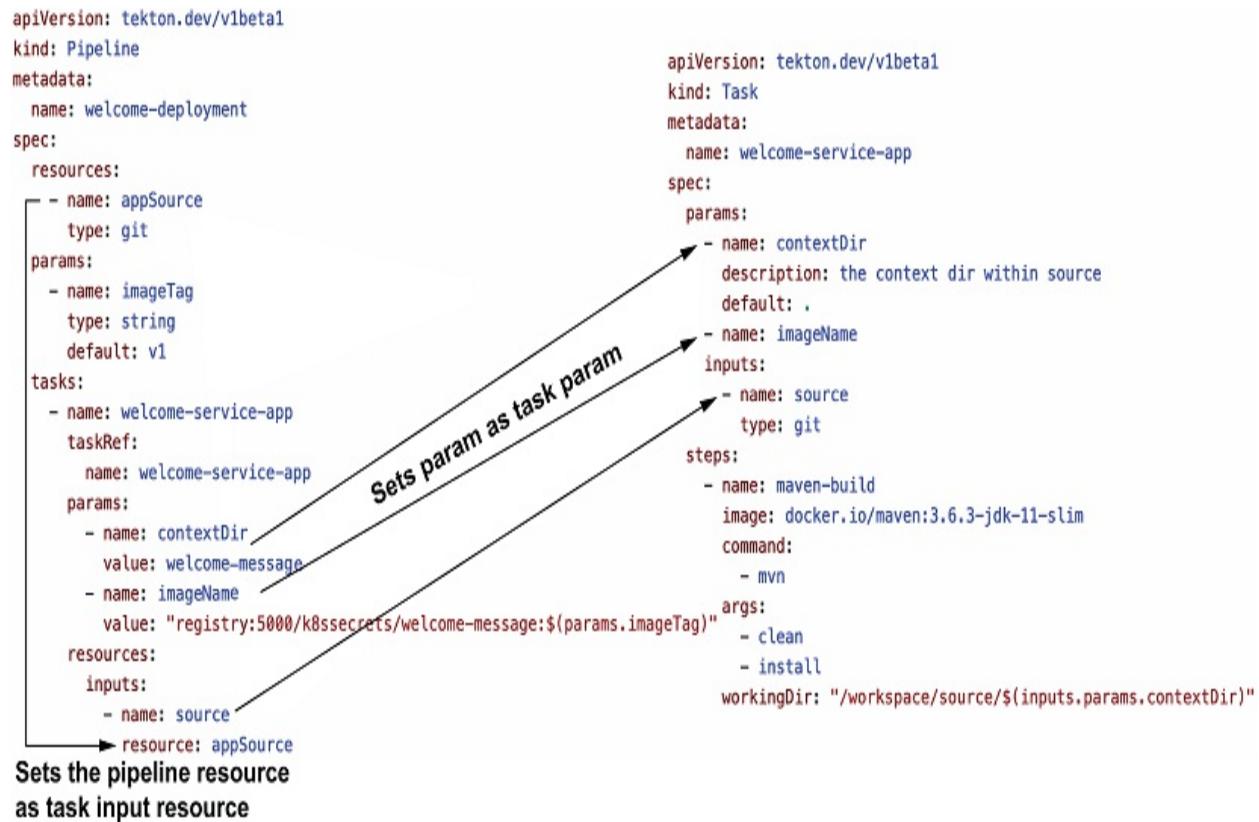
Execute the following command to register the Pipeline into the Kubernetes cluster:

Listing 7.26. Register pipeline

```
kubectl apply -f welcome-service-pipeline.yaml #1
```

Figure [7.10](#) shows the relationship between Pipeline and Task parameters are shown:

Figure 7.10. Relationship between Pipeline and Task parameters



7.3.5 Pipeline Run

We create a Pipeline Run to trigger the `welcome-deployment` pipeline defined in the previous step. In this pipeline run, apart from setting the Git repository location, the **container image tag** is also provided.

Listing 7.27. welcome-service-pipeline-run.yaml

```
apiVersion: tekton.dev/v1beta1
kind: PipelineRun
metadata:
  name: welcome-pipeline-run
spec:
```

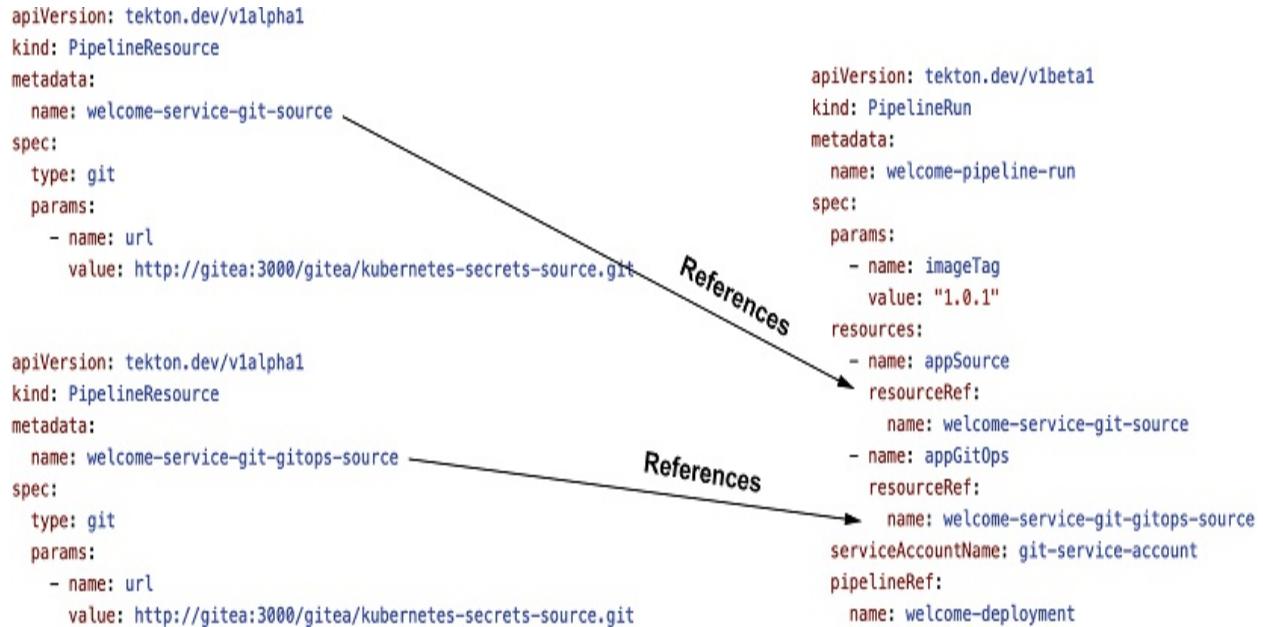
```

params:
  - name: imageTag
    value: "1.0.0" #1
resources:
  - name: appSource
    resourceRef:
      name: welcome-service-git-source
pipelineRef:
  name: welcome-deployment

```

Figure 7.11 shows the relationship between PipelineRun and PipelineResource parameters are shown:

Figure 7.11. Relationship between PipelineRun and PipelineResource



Execute the following command to trigger the Pipeline into the Kubernetes cluster:

Listing 7.28. Register pipeline run

```
kubectl apply -f welcome-service-pipeline-run.yaml #1
```

At this point, a TaskRun is automatically created and executed for each task defined in the tasks section of the Pipeline object. To list them, run the following command in a terminal window:

Listing 7.29. List all task runs

```
tkn tr list #1
```

And the output provides a list of all TaskRuns executed in the Kubernetes cluster with its status.

NAME	STARTED	DURAT
welcome-pipeline-run-welcome-service-app-12zns	1 minute ago	---

tkn let us inspect the logs of a TaskRun and in case of a failure, find the error cause.

In a terminal run the following command, the `-f` option is used to stream live logs of the current execution:

Listing 7.30. Stream logs from pipeline run

```
tkn tr logs welcome-pipeline-run-welcome-service-app-12zns -f #1
```

And we'll see the pipeline logs in the console:

```
#1
[maven-build] Downloaded from central: https://repo.maven.apache.org/maven2/io/
[maven-build] Downloading from central: https://repo.maven.apache.org/maven2/io/
[maven-build] Downloading from central: https://repo.maven.apache.org/maven2/io/
[...]
#2
[build-and-push-image] STEP 11: ENTRYPOINT [ "/deployments/run-jar"]
[build-and-push-image] STEP 12: COMMIT test.org/k8ssecrets/welcome-service
[build-and-push-image] --> abab5f4192b
[build-and-push-image] abab5f4192b3a5d9317419d61553d91baf0dfc4df1

#3
[build-and-push-image] Getting image source signatures
[build-and-push-image] Copying blob sha256:f0b7ce40f8b0d5a8e10eec
[build-and-push-image] Copying blob sha256:ba89bf93365092f038be15
[build-and-push-image] Copying blob sha256:04a05557bbadc648beca5c
[build-and-push-image] Copying blob sha256:821b0c400fe643d0a9f146
[build-and-push-image] Copying blob sha256:7a6b87549e30f9dd8d2502
[build-and-push-image] Copying config sha256:abab5f4192b3a5d93174
[build-and-push-image] Writing manifest to image destination
```

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

Remember that Task is executed as a Pod, and each of the steps is executed inside a container within that Pod. This can be seen when running the following command:

Listing 7.31. Get all Pods

```
kubectl get pods #1
```

NAME	READY
welcome-pipeline-run-welcome-service-app-12zns-pod-98b21	0

Since welcome-service-app task is composed of three steps (git clone, Maven build, Docker build/push), three containers were created during the task execution as seen in the READY column.

The Continuous Integration pipeline cycle finishes when the container image is published to the container registry. But the service is not deployed nor released yet to the Kubernetes cluster.

In the following chapter, we'll see how to use Continuous Deployment and GitOps methodology to deploy and release the service to the cluster.

7.4 Summary

- Kubernetes Secrets are used either in the application code (username, passwords, API keys) and in the CI pipelines (username, passwords of external services).
- Continuous Integration secrets need to be protected as any other secret. Use SealSecrets in Tekton and Argo CD to store encrypted secrets in Git. Enable Kubernetes Data Encryption at Rest to store encrypted secrets inside Kubernetes.
- Tekton is the Kubernetes-Native platform to implement the Continuous Integration pipeline.
- Git is used as a single source of truth not only for the source code but also for the pipeline scripts.

8 Kubernetes-Native Continuous Delivery and Secrets

This chapter covers

- Introducing Continuous Delivery and Deployment methodology.
- Implementing a Kubernetes-Native Continuous Deployment pipeline using GitOps methodology.
- Showing ArgoCD as a Kubernetes-Native solution for implementing GitOps.

In the previous chapter, we've seen how to manage secrets during the Continuous Integration phase, we showed how to build the application, create a container image, and publish it to the container registry. But the service is not deployed nor released yet to the Kubernetes cluster, let's see in this chapter how to deliver the application securely.

In this chapter, we'll see how to use Continuous Deployment and GitOps methodology to deploy and release services to a Kubernetes cluster using Argo CD to deliver quality applications rapidly while managing the secrets correctly during the whole pipeline secrets-leak occurs in this phase of the development.

8.1 Introduction to Continuous Delivery/Deployment

Continuous Delivery is a methodology that revolves around releasing software faster and more frequently. This methodology helps reduce the cost, time, and risk of delivering changes that potentially affect the user experience. Because delivery of the application is performed continuously and with incremental updates, it's easier to capture feedback from the end-user and react accordingly.

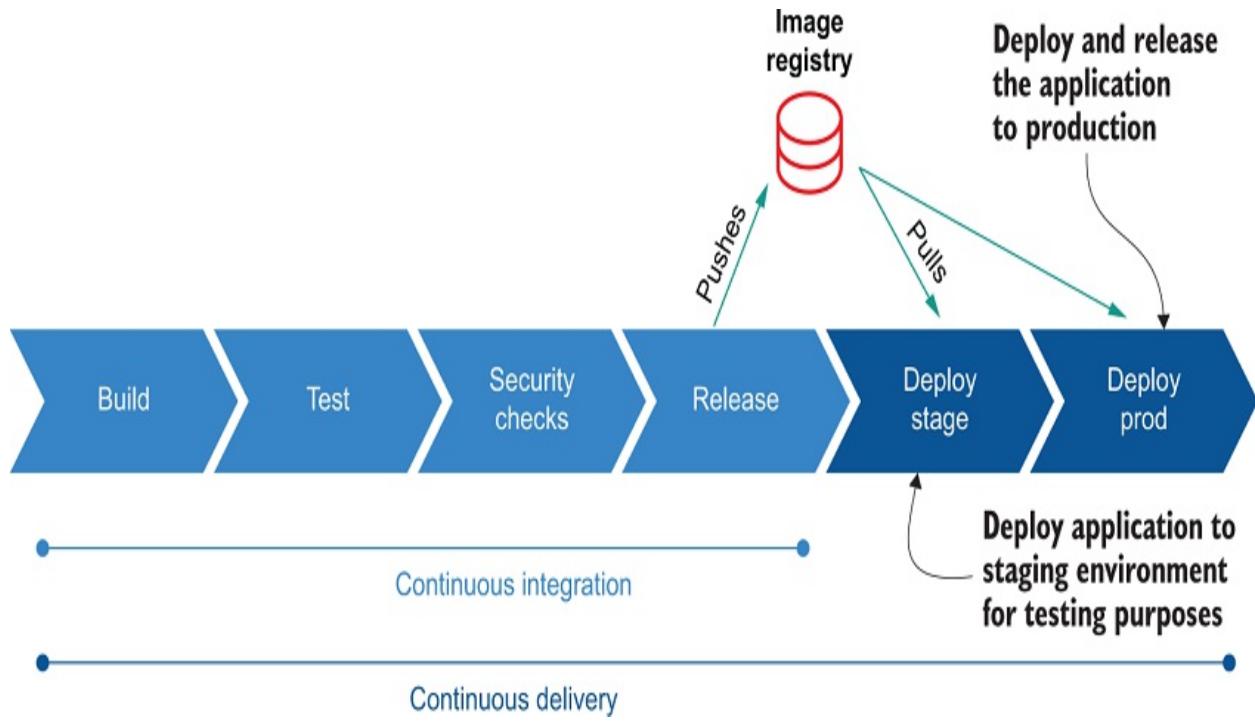
The central concept of CD is the *deployment pipeline*. As the name suggests, it's a set of steps or procedures through which the application must pass to be released for production. The deployment pipeline may be changed depending on the process you choose to follow when releasing the application, but one typical pipeline is composed of the following stages:

- *Commit stage*: The first part of the release process, triggered after a team member commits something to the SCM server. This stage is the continuous integration phase shown in the previous chapter.
- *Acceptance tests*: Tests to make sure the application meets the expectations from the point of view of the business. Some of these tests might be automatic, but others not, such as exploratory testing.
- *Release*: Based on all feedback from each stage, key users decide to release to not production or drop the version.

Notice that a release process in *Continuous Delivery* implies a manual decision to perform the actual release. On the other hand, *Continuous Deployment* automatically releases every change to production on a successful build.

Figure 8.1 shows the steps implied in the Continuous Integration and Continuous Delivery pipelines.

Figure 8.1. Continuous Integration stages vs Continouous Delivery stages



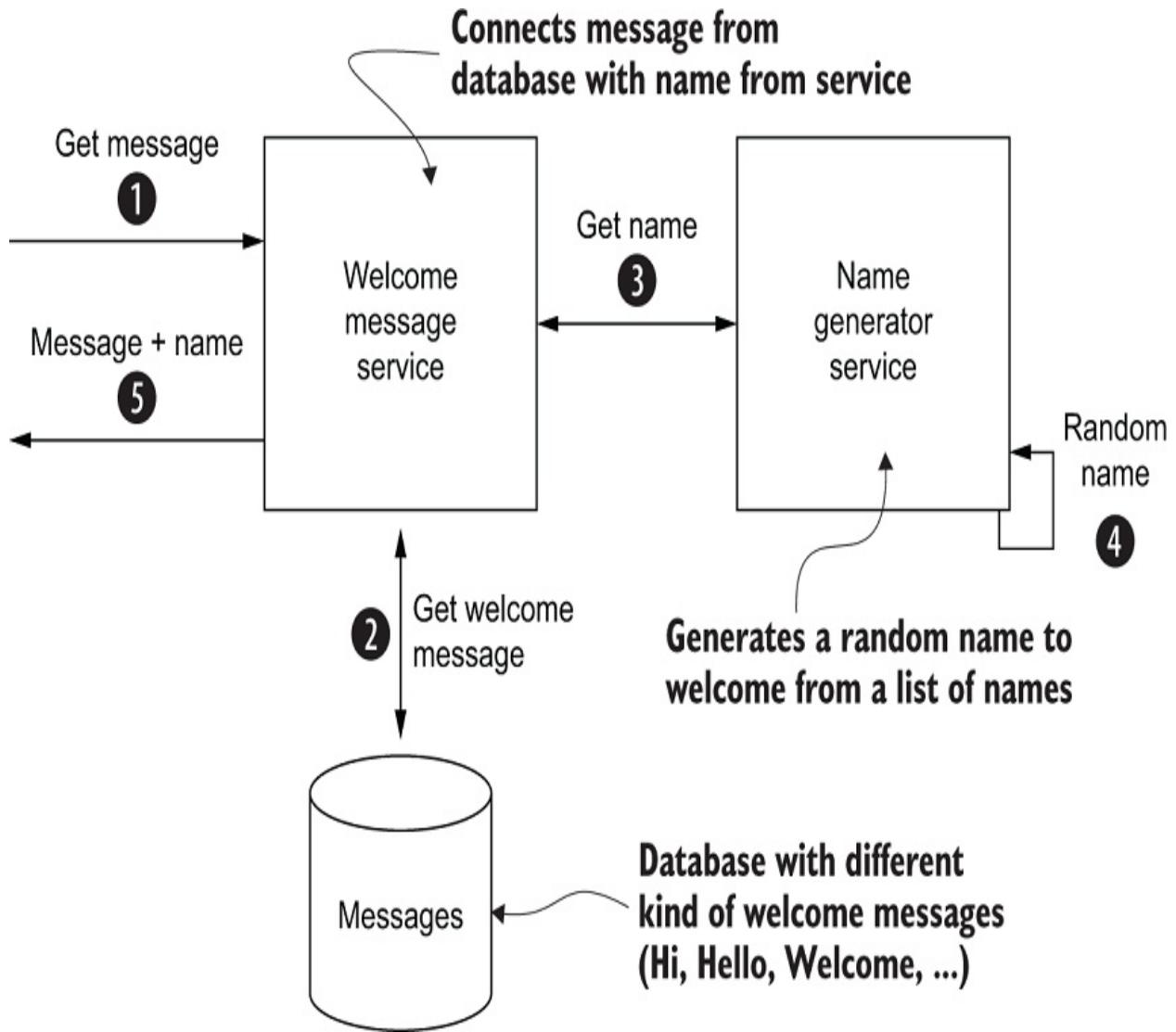
In the following sections, we will focus on the service's release process, deploying it automatically to the Kubernetes cluster, keeping the involved secrets protected using the DevOps methodology.

8.2 Continuous Delivery for Welcome Message

This chapter will deploy the same application used in the previous chapter to a Kubernetes cluster. It takes where the CI phase left (*Welcome Message* container pushed to container registry) and delivers it to the production environment.

As a reminder, figure 8.2, we can see an overview of the application and the parts that are composed of:

Figure 8.2. Overview of the interactions between Welcome and Name services



Now that we have an overview of the application to deploy, we show the Kubernetes resources files to deploy it.

8.2.1 Deploying Name Generator Service

This chapter focuses on applying Continuous Delivery (CD) principles in the *Welcome Message* service to deploy it in the Kubernetes cluster using GitOps methodology automatically. Although this is an example for one specific service, the same approach can be applied similarly to any other service like a payment service, stock service, or user management service.

To keep things simple, we'll deploy the *Name Generator* service manually

using the following deployment file [8.1](#):

Listing 8.1. src/main/kubernetes/deployment.yml

```
---  
apiVersion: v1  
kind: Service  
metadata:  
  labels:  
    app.kubernetes.io/name: name-generator  
    app.kubernetes.io/version: 1.0.0  
  name: name-generator #1  
spec:  
  ports:  
    - name: http  
      port: 8080  
      targetPort: 8080  
  selector:  
    app.kubernetes.io/name: name-generator  
    app.kubernetes.io/version: 1.0.0  
  type: ClusterIP  
---  
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  labels:  
    app.kubernetes.io/version: 1.0.0  
    app.kubernetes.io/name: name-generator  
  name: name-generator  
spec:  
  replicas: 1  
  selector:  
    matchLabels:  
      app.kubernetes.io/version: 1.0.0  
      app.kubernetes.io/name: name-generator  
  template:  
    metadata:  
      labels:  
        app.kubernetes.io/version: 1.0.0  
        app.kubernetes.io/name: name-generator  
    spec:  
      containers:  
        - env:  
          - name: KUBERNETES_NAMESPACE  
            valueFrom:  
              fieldRef:
```

```
        fieldPath: metadata.namespace
image: quay.io/lordofthejars/name-generator:1.0.0 #2
imagePullPolicy: Always
name: name-generator
ports:
- containerPort: 8080
  name: http
  protocol: TCP
```

With the deployment file created, run the following command [8.2](#):

Listing 8.2. Deploy name generator service

```
kubectl apply -f src/main/kubernetes/deployment.yml #1
service/name-generator created
deployment.apps/name-generator created
```

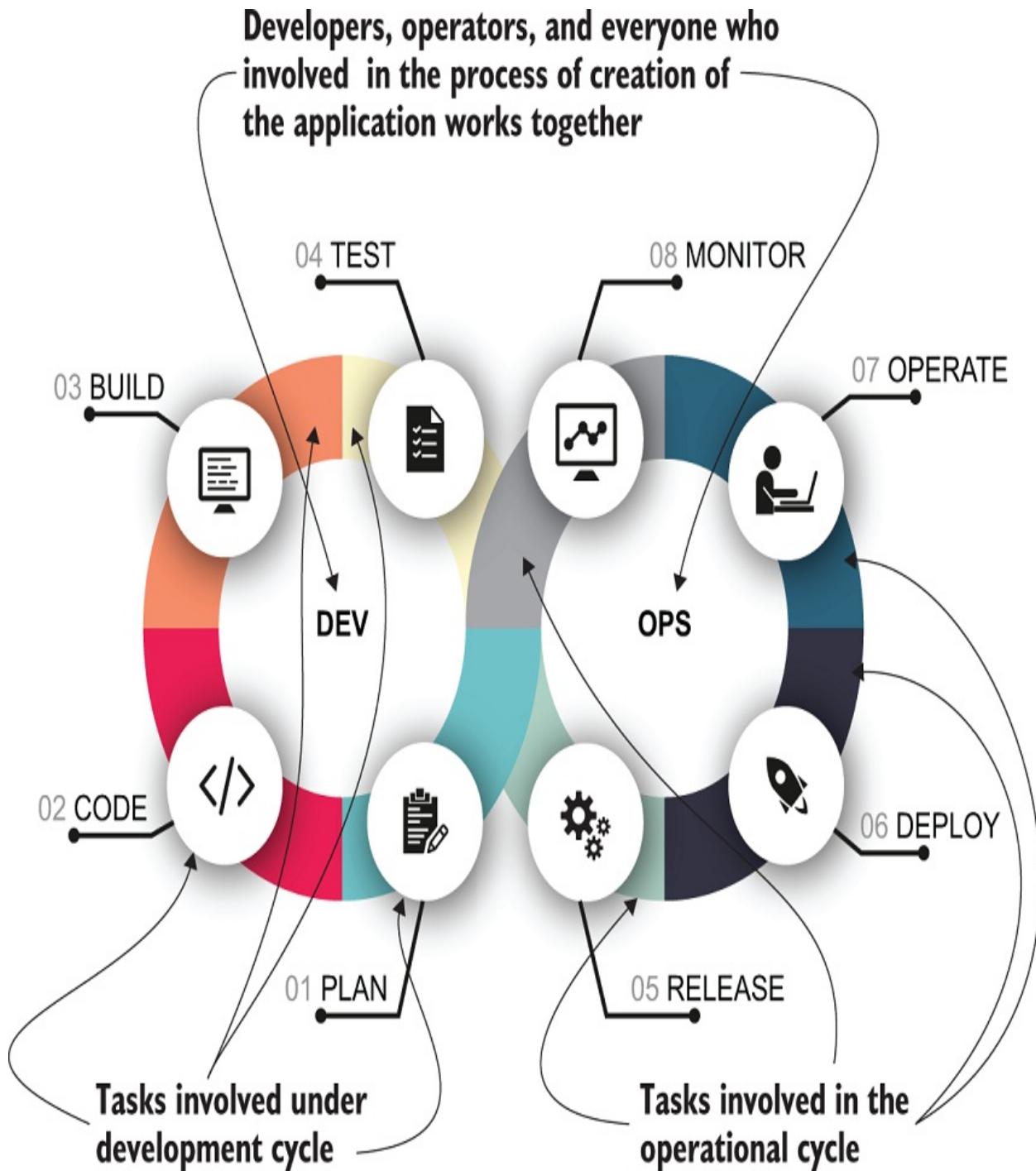
With the *Name Generator* service up and running, let's deploy and release *Welcome Message* service using *GitOps* methodology.

8.2.2 DevOps & GitOps

DevOps is a set of practices that automates and helps integrate the processes between Software Development and IT teams, so applications are built, tested, and released faster and more reliably.

DevOps isn't about developers and operators team only; it involves the whole organization. Every team should be a part of the software lifecycle from the early beginning, from the planning phase until the application is released in the production environment is monitored.

Figure 8.3. DevOps lifecycle



GitOps is a way to implement *DevOps* methodology and it's based on the assumption that Git is the one and only source of truth. Not only the source code of the application is stored in Git, but also the scripts/pipeline definitions to build the application and the infrastructure code to release and update the application. This implies that all parts of the application are versioned, branched and of course, they can be audited.

In summary GitOps principles are:

- Git is the single source of truth
- Treat everything as code
- Operations through Git workflows

One of the important aspects of GitOps is that any update on the Git repository related to the infrastructure part must trigger an update to the environment to meet the desired state of the application.

When there is a divergence between the desired state (set in the Git repo) and the observed state (the real state in the cluster), the convergence mechanism is executed to drive the observed state towards the desired state defined in the version control.

There are two ways to cause a divergence.

1. If a human operator manually updates the Kubernetes cluster, the desired and observed state will be different, and the convergence mechanism updates the Kubernetes cluster to the desired state defined in Git.
2. If a file is updated in Git (for example a new container image needs to be released), the desired and observed state will be different, and the Kubernetes cluster state is updated to the new state defined in Git.

Let's see how we can update a Kubernetes cluster via Git.

8.3 Argo CD

It's time to deploy *Welcome message* service to Kubernetes cluster using Argo CD and following *GitOps* methodology.

GitOps is a way to implement *DevOps* methodology, and it's based on the assumption that Git is the only source of truth. The source code of the application is stored in Git, but also the scripts/pipeline definitions to build the application and the infrastructure code to release and update the application. This implies that all parts of the application are versioned,

branched, and of course, they can be audited.

In summary, GitOps principles are:

- Git is the single source of truth
- Treat everything as code
- Operations through Git workflows

One of the essential aspects of GitOps is that any update on the Git repository related to the infrastructure part must trigger an update to the environment to meet the desired state of the application.

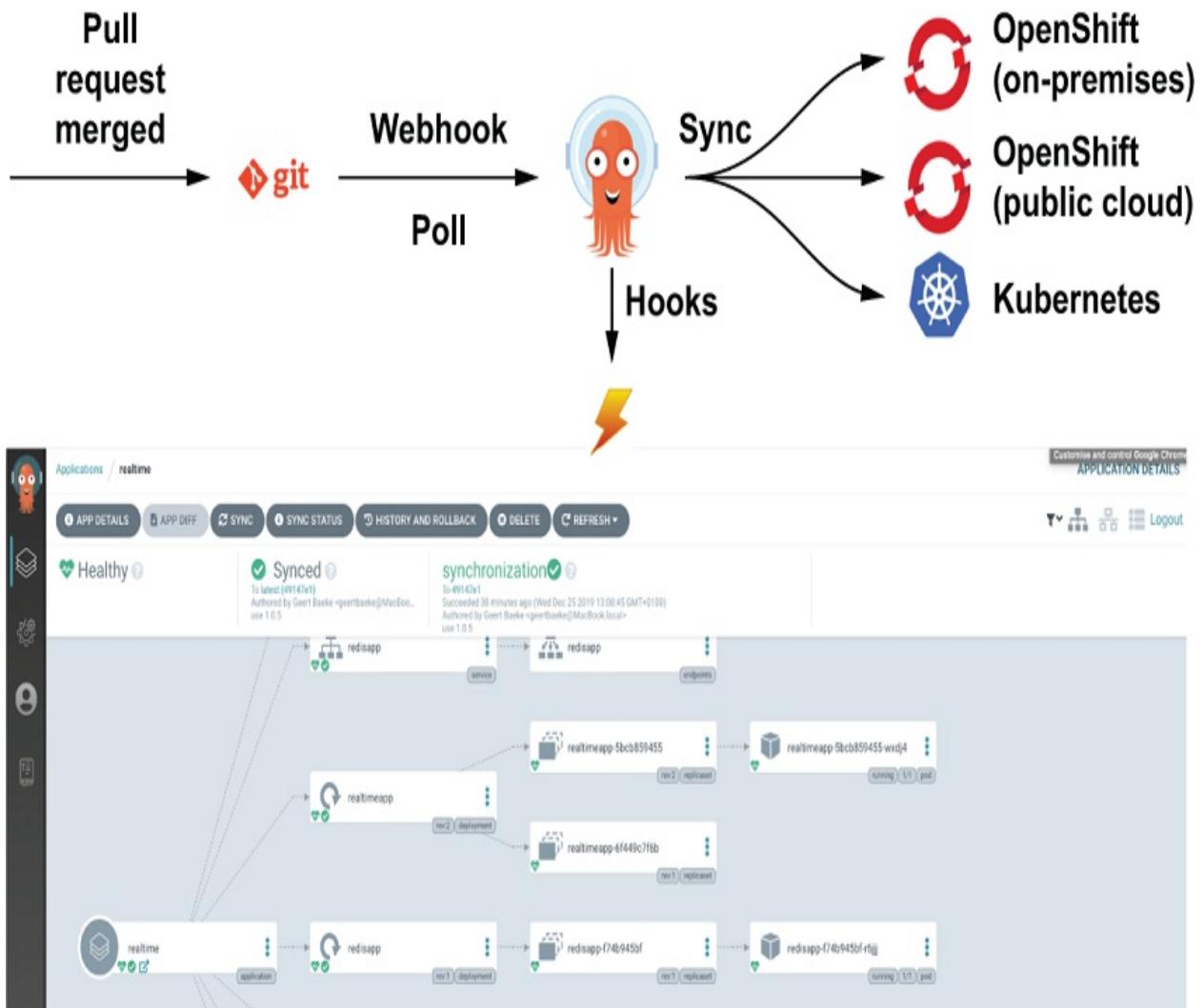
When there is a divergence between the desired state (set in the Git repo) and the observed state (the actual state in the cluster), the convergence mechanism is executed to drive the observed state towards the desired state defined in the version control.

There are two ways to cause a divergence.

1. If a human operator manually updates the Kubernetes cluster, the desired and observed state will be different. The convergence mechanism updates the Kubernetes cluster to the desired state defined in Git.
2. If a file is updated in Git (for example, a new container image needs to be released), the desired and observed state will be different, and the Kubernetes cluster state is updated to the new state defined in Git.

Let's see how we can update a Kubernetes cluster via Git.

Figure 8.4. Argo CD synchronization



Argo CD has three major components:

API Server: The ArgoCD backend exposes the API to Web UI, CLI, or any other system. The main responsibilities of this part are application management, security concerns, and manage Git webhook events.

- **Repository Server**

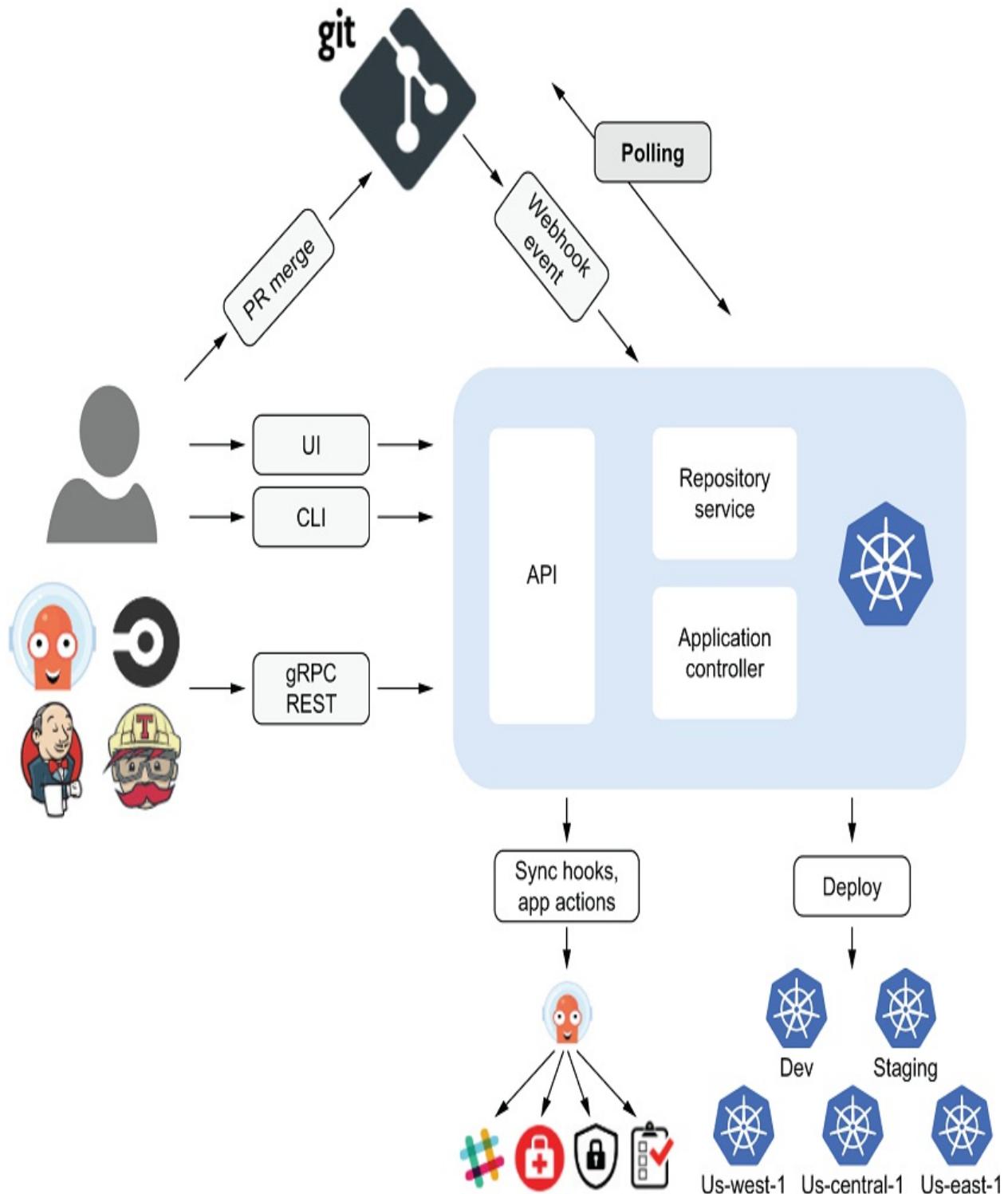
It's an internal service to maintain the local cache of the Git repositories.

- **Application Controller**

It's implemented as a Kubernetes controller that continuously monitors the manifests placed at the Git repository and compares them with the live state on the cluster. Optionally takes corrective actions.

Figure 8.5 show all these parts and how they are related together:

Figure 8.5. Interactions between ArgoCD elements and Kubernetes cluster to deploy an application



8.3.1 Installation of ArgoCD

To install ArgoCD 1.8.7, create a new Kubernetes namespace and apply the Argo CD resource to the Kubernetes cluster as shown in listing 8.3:

Listing 8.3. Install ArgoCD

```
kubectl create namespace argocd #1
kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/
customresourcedefinition.apiextensions.k8s.io/applications.argoproj/
serviceaccount/argocd-application-controller created
serviceaccount/argocd-dex-server created
serviceaccount/argocd-redis created
serviceaccount/argocd-server created
role.rbac.authorization.k8s.io/argocd-application-controller crea
role.rbac.authorization.k8s.io/argocd-dex-server created
role.rbac.authorization.k8s.io/argocd-redis created
role.rbac.authorization.k8s.io/argocd-server created
clusterrole.rbac.authorization.k8s.io/argocd-application-controll
clusterrole.rbac.authorization.k8s.io/argocd-server created
rolebinding.rbac.authorization.k8s.io/argocd-application-controll
rolebinding.rbac.authorization.k8s.io/argocd-dex-server created
rolebinding.rbac.authorization.k8s.io/argocd-redis created
rolebinding.rbac.authorization.k8s.io/argocd-server created
clusterrolebinding.rbac.authorization.k8s.io/argocd-application-c
clusterrolebinding.rbac.authorization.k8s.io/argocd-server create
configmap/argocd-cm created
configmap/argocd-gpg-keys-cm created
configmap/argocd-rbac-cm created
configmap/argocd-ssh-known-hosts-cm created
configmap/argocd-tls-certs-cm created
secret/argocd-secret created
service/argocd-dex-server created
service/argocd-metrics created
service/argocd-redis created
service/argocd-repo-server created
service/argocd-server created
service/argocd-server-metrics created
deployment.apps/argocd-dex-server created
deployment.apps/argocd-redis created
deployment.apps/argocd-repo-server created
deployment.apps/argocd-server created
statefulset.apps/argocd-application-controller created
```



Tip

argocd CLI is command-line utility used to interact with the Argo CD part.

To install it just visit <https://github.com/argoproj/argo-cd/releases/tag/v1.7.14>, download the package for your platform, uncompress the archive, and copy the argocd file into a PATH directory so it's accessible from any directory.

Then expose the Argo CD server by changing the Argo CD Kubernetes Service to LoadBalancer type using patch command as shown in listing [8.4](#).

Listing 8.4. Expose ArgoCD server

```
kubectl patch svc argocd-server -n argocd -p '{"spec": {"type": "LoadBalancer", "externalIPs": [{"ip": "192.168.1.10"}]}}
```

To use argocd CLI tool, we need the external IP and the Argo CD server exposed port. We can get them by running the commands shown in listing [8.5](#):

Listing 8.5. Argo CD access IP and port

```
IP=$(minikube ip -p argo) #1
PORT=$(kubectl get service/argocd-server -n argocd -o jsonpath='{
```

The last step before configuring Argo CD is login to the Argo CD server using the CLI tool:

By default, the username is admin and the initial password is autogenerated to be the Pod name of the Argo CD API server. The password is retrieved with the command shown in listing [8.6](#):

Listing 8.6. Gets Argo CD password

```
kubectl get pods -n argocd -l app.kubernetes.io/name=argocd-server
```

Run login command as shown in listing [8.7](#) to login into Argo CD server using admin username and the password retrieved in the last step:

Listing 8.7. ArgoCD login

```
argocd login $IP:$PORT #1
```

8.3.2 Welcome Service and GitOps

When we installed Gitea at [7.2.2](#) section, a Git repository with the sources required in this chapter were migrated inside. This repo contains two source directories, one for each service, and a directory named `welcome-message-gitops` where all Kubernetes YAML files related to the deployment of the *Welcome Message* service and GitOps definitions are placed.

The repository layout is shown in listing [8.8](#):

Listing 8.8. Repository layout

```
name-generator #1
├── README.md
├── mvnw
├── mvnw.cmd
├── pipelines
├── pom.xml
└── src
└── target
welcome-message #2
├── Dockerfile
├── README.md
├── mvnw
├── mvnw.cmd
├── pom.xml
└── src
└── target
└── vault-init
welcome-message-gitops #3
└── apps #4
└── cluster #5
└── gitops #6
```

`welcome-message-gitops` directory is composed of three directories:

- `apps`
Deployment YAML file to release the *Welcome Message* service to Kubernetes.
- `cluster`

YAML files to deploy external dependencies required by *Welcome Message* service (*PostgreSQL* and *Vault*).

- **gitops**

YAML files required to register the application in Argo CD.

```
welcome-message-gitops
└── apps
└── cluster
└── gitops
```

Apps

The `apps` folder contains the deployment files required to deploy the service into a Kubernetes cluster. In this case, there are two standard Kubernetes Deployment and Service resources.

```
welcome-message-gitops
└── apps
    └── app.yaml
    └── service.yaml
```

To deploy the *Welcome Message* service, create the `apps.yaml` file with the content shown in listing 8.9:

Listing 8.9. `apps/apps.yaml`

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app.kubernetes.io/name: welcome-message
  name: welcome-message
spec:
  replicas: 1
  selector:
    matchLabels:
      app.kubernetes.io/name: welcome-message
  template:
    metadata:
      labels:
        app.kubernetes.io/name: welcome-message
  spec:
```

```

containers:
- env:
  - name: KUBERNETES_NAMESPACE
    valueFrom:
      fieldRef:
        fieldPath: metadata.namespace
image: quay.io/lordofthejars/welcome-message:1.0.0 #1
imagePullPolicy: Always
name: welcome-message
ports:
- containerPort: 8080
  name: http
  protocol: TCP

```

Create service.yml file to make the *Welcome Message* accessible with the content shown in listing [8.10](#):

Listing 8.10. apps/service.yml

```

---
apiVersion: v1
kind: Service
metadata:
  labels:
    app.kubernetes.io/name: welcome-message
  name: welcome-message
spec:
  ports:
  - name: http
    port: 8080
    targetPort: 8080
  selector:
    app.kubernetes.io/name: welcome-message
  type: LoadBalancer

```

Cluster

The cluster folder contains all YAML files required to deploy and configure the external dependencies required by the service, in this case *PostgreSQL* as the database, and *Hashicorp Vault* as secret management system.

```

welcome-message-gitops
└── cluster
    └── postgresql.yaml

```

```
|   └── vault-job.yaml  
|   └── vault-secrets.yaml  
|   └── vault.yaml
```

`postgresql.yaml` and `vault.yaml` files are standard deployment files for deploying both of the services to the Kubernetes cluster, but two more files are deserving an explanation.

The `vault-secrets.yaml` file is a Kubernetes Secret object containing secrets required to be stored into Hashicorp Vault and consumed by the application. In this case, the token to log in to Hashicorp Vault and the API token used by the *Welcome Service* to authenticate to *Name Generator Service*. The file is partially shown in listing 8.11:

Listing 8.11. `cluster/vault-secrets.yml`

```
apiVersion: v1  
kind: Secret  
metadata:  
  name: vault-secrets  
type: Opaque  
data:  
  VAULT_LOGIN: cm9vdA== #1  
  NAME_SERVICE_TOKEN: ZX1KcmFXUWlPa..... #2
```

The `vault-job.yaml` file is a Kubernetes Job object configuring the Hashicorp Vault instance. It's applied after the deployment of Hashicorp Vault and enables Kubernetes auth mode and database dynamic secrets, configures policies, and adds the API token into the key-value secret store.

The file is shown partially in listing 8.12:

Listing 8.12. `cluster/vault-job.yml`

```
apiVersion: batch/v1  
kind: Job  
metadata:  
  name: init-vault  
  annotations:  
    argocd.argoproj.io/hook: PostSync #1  
    argocd.argoproj.io/hook-delete-policy: HookSucceeded  
spec:
```

```

template:
spec:
  volumes:
    - name: vault-scripts-volume
      configMap:
        name: vault-scripts
        defaultMode: 0777
  containers:
    - name: init-vault
      image: vault:1.6.2
      envFrom:
        - secretRef: #2
          name: vault-secrets
      volumeMounts:
        - mountPath: /vault-scripts #3
          name: vault-scripts-volume
      command:
        - /bin/ash #4
        - -c
        - |
          export VAULT_ADDR=http://vault:8200
          vault login $VAULT_LOGIN
          vault auth enable kubernetes
          vault secrets enable database
          vault write database/config/mydb plugin_name=postgres
          vault write database/roles/mydbrole db_name=mydb ...
          vault policy write vault-secrets-policy/vault-scripts
          vault kv put secret/myapps/welcome/config name-service
      restartPolicy: Never
  backoffLimit: 2

```



Important

To keep things simple `vault-secrets.yaml` is of kind `Secrets` but it should be protected in any of the ways explained in Chapter 3.

Gitops

The `gitops` folder contains all files used to register the previous folders as Argo CD applications.

```

welcome-message-gitops
└── gitops

```

```
└── apps-ops.yaml #1
    cluster-ops.yaml #2
```

In the following section, we'll see these files in more detail.



Tip

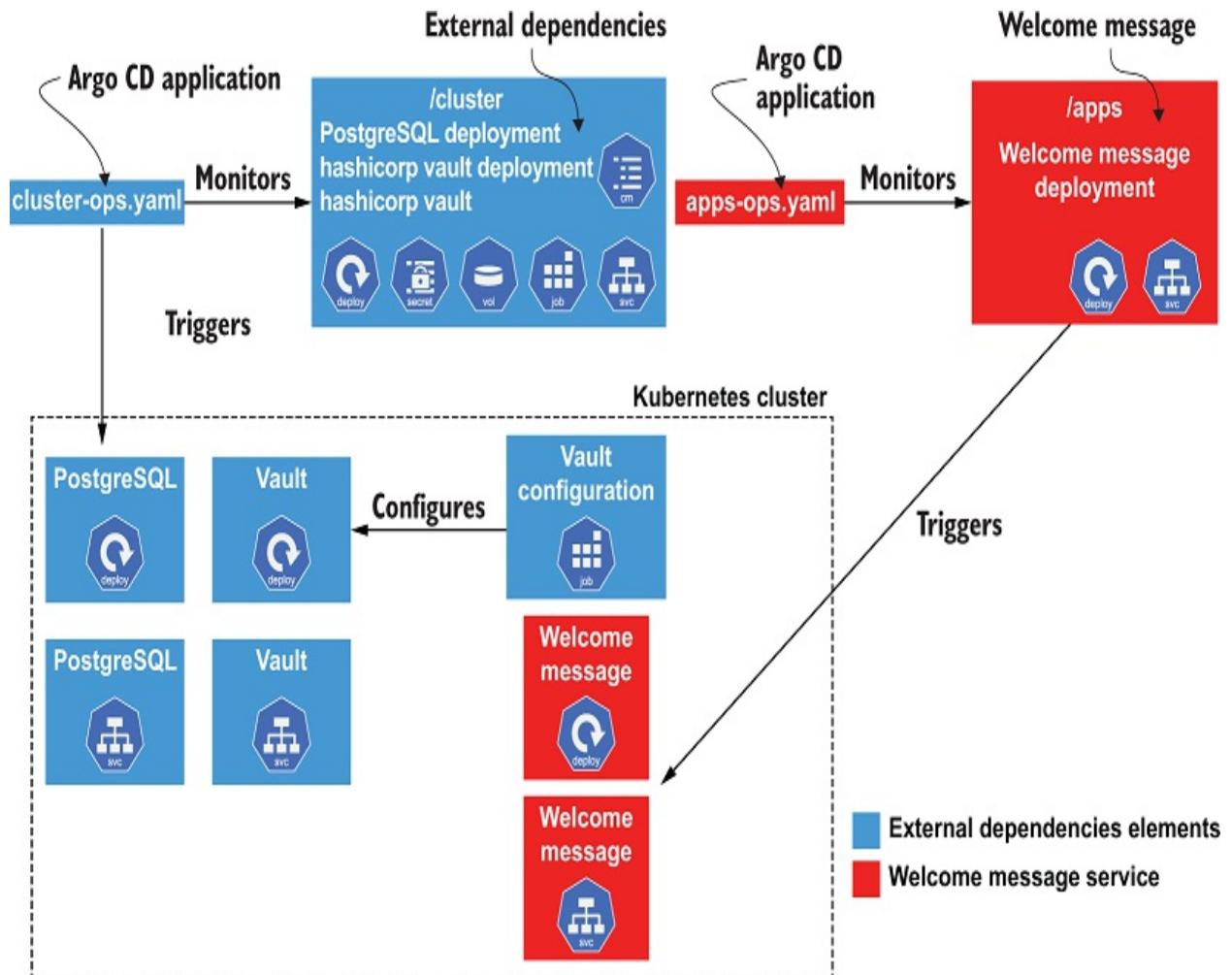
You could add the application and the external dependencies deployment files into the same directory. But our advice is to keep the files that usually change separated from those that rarely do.

8.3.3 Creating Welcome Message Service from Git Repository

An Argo CD *Application* is a group of Kubernetes resources used to deploy the application to the target environment and keep it in the desired state. An Application is defined in an Argo CD Custom Resource Definition (CRD) file where we specify parameters like Git repository, the path of Kubernetes resources, or the target where the application is deployed.

Figure [8.6](#) shows the schema of what is deployed by `cluster-ops` and `apps-ops` Argo CD applications.

Figure 8.6. Argo CD application deployments



In the listing 8.13 we see how to define an Argo CD application that clones the Git repo defined in Gitea, listens to any change in the welcome-message-gitops/apps directory, and apply these changes to meet the desired state. As we said before, the apps directory contains the resources to deploy the *Welcome Message* service to the Kubernetes cluster.

Create the `apps-ops.yaml` file in the `welcome-message-gitops/gitops` folder.

Listing 8.13. `welcome-message-gitops/gitops/apps-ops.yaml`

```

apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: welcome-cluster-apps
  namespace: argocd

```

```

spec:
  project: default
  syncPolicy: #1
    automated:
      prune: true #2
      selfHeal: true #3
  source:
    repoURL: http://gitea.default.svc:3000/gitea/kubernetes-secrets
    targetRevision: HEAD
    path: welcome-message-gitops/apps #5
  destination: #6
    server: https://kubernetes.default.svc
    namespace: default

```

In a similar way, cluster directory is added as an Argo CD application.

Create the `cluster-ops.yaml` file in the `welcome-message-gitops/gitops` folder.

Listing 8.14. welcome-message-gitops/gitops/cluster-ops.yaml

```

apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: welcome-cluster-ops
  namespace: argocd
spec:
  project: default
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
  source:
    repoURL: http://gitea.default.svc:3000/gitea/kubernetes-secrets
    targetRevision: HEAD
    path: welcome-message-gitops/cluster #1
  destination:
    server: https://kubernetes.default.svc
    namespace: default%

```

Let's apply the `cluster-ops.yaml` resource to install and configure the external dependencies required by the *Welcome Message* service. In a terminal window run the command shown in listing [8.15](#):

Listing 8.15. Register cluster application

```
kubectl apply -f welcome-message-gitops/gitops/cluster-ops.yaml
```

The previous execution registers the `cluster` directory as the Argo CD application. Since it's the first time, and the `syncPolicy` parameter is set to `automated`, Argo CD automatically applies the resources defined there.

Validate that PostgreSQL and Vault are deployed by getting Pods of default namespace using the command shown in listing [8.16](#):

Listing 8.16. Get all pods

```
kubectl get pods -n default
```

NAME	READY	STATUS	RESTARTS
gitea-deployment-7fbff9c8b-bbcjq	1/1	Running	1
name-generator-579ccdc5d5-mhzft	1/1	Running	2
postgresql-59ddd57cb6-tjrg2	1/1	Running	0
registry-deployment-64d49ff847-hljjg9	1/1	Running	2
vault-0	1/1	Running	0
welcome-message-55474d6b78-g915w	1/1	Running	0

The `argocd` CLI tool also lets us review the status of the deployment using the command shown in listing [8.17](#):

Listing 8.17. List ArgoCD applications

```
argocd app list #1
```

NAME	CLUSTER	NAMESPACE
welcome-cluster-ops	https://kubernetes.default.svc	default
STATUS	HEALTH	SYNCPOLICY
Synced	Healthy	Auto-Prune
REPO		
https://github.com/lordofthejars/kubernetes-secrets-source.git		
PATH	TARGET	
welcome-message-gitops/cluster	HEAD	

status field shows the current status of the resources. When it's set to

Synced, the cluster is aligned with the state specified with the Git repository.

To deploy the *Welcome Message* service, apply the `apps-ops.yaml` file created in the previous step using the command shown in listing [8.18](#):

Listing 8.18. Register service application

```
kubectl apply -f welcome-message-gitops/gitops/apps-ops.yaml #1
```

Welcome Message service is deployed when its Pod is in the running state using the command shown in listing [8.19](#):

Listing 8.19. Get all pods

```
kubectl get pods -n default
```

NAME	READY	STATUS	RESTARTS
gitea-deployment-7fbbf9c8b-bbcjq	1/1	Running	1
name-generator-579ccdc5d5-mhzft	1/1	Running	2
postgresql-59ddd57cb6-tjrg2	1/1	Running	0
registry-deployment-64d49ff847-hljjg9	1/1	Running	2
vault-0	1/1	Running	0
welcome-message-55474d6b78-g915w	1/1	Running	0

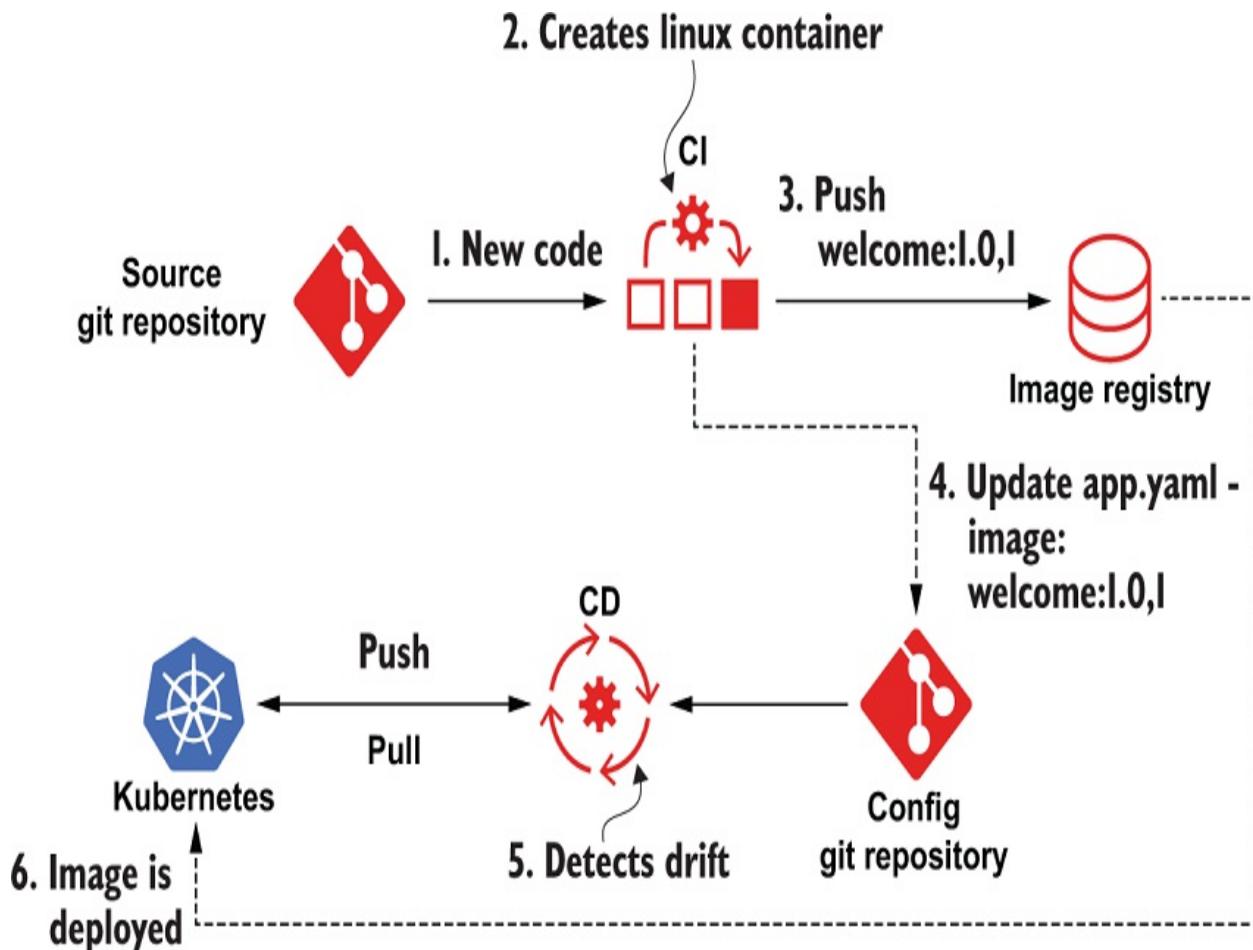
8.3.4 Updating Welcome Service

So far, we've learned how to build the *Welcome Message* service using Tekton and how to deploy it the first time automatically using the Argo CD project. But what's happen when the service is already deployed, and a new version needs to be released?

Apart from packaging the new version of the service, building a container, and pushing it to the container registry, as it's explained in [7.3](#) section, now the CI pipeline needs to update the *Welcome Message* service Kubernetes deployment file with the new container tag and push the update to Git repository to start the rolling update of the service.

Figure [8.7](#) shows how Tekton (CI part) and Argo CD (CD part) interrelate to implement a Continuous Delivery pipeline.

Figure 8.7. Inter-relationship between Tekton and Argo CD part



To implement these two remaining steps, some changes need to be done in Tekton resources.

PipelineResource for GitOps repository

The first thing to register is a new Pipeline Resource registering the GitOps repository location. This repository is the place where Argo CD is listening for any change. The definition is shown in listing 8.20:

Listing 8.20. welcome-service-gitops-resource.yaml

```
apiVersion: tekton.dev/v1alpha1
kind: PipelineResource
metadata:
```

```
name: welcome-service-git-gitops-source
spec:
  type: git
  params:
    - name: url
      value: http://gitea.default.svc:3000/gitea/kubernetes-secre
```

And in a terminal window apply the resource using the command shown in listing [8.21](#):

Listing 8.21. Register GitOps repository

```
kubectl apply -f welcome-service-gitops-resource.yaml #1
```

We need to set the username and password to push the changes done at the deployment file to the previous repository. This is done by creating a Kubernetes Secret and a Service Account with the content shown in listing [8.22](#):

Listing 8.22. git-secret.yaml

```
---
apiVersion: v1
kind: Secret
metadata:
  name: git-auth
  annotations:
    tekton.dev/git-0: http://gitea.default.svc:3000 #1
type: kubernetes.io/basic-auth #2
stringData: #3
  username: gitea
  password: gitea1234
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: git-service-account
secrets:
  - name: git-auth #4
```



Important

Remember to manage these secrets correctly using any of the techniques shown in Chapter 3.

And in a terminal window apply the resource using the command shown in listing [8.23](#):

Listing 8.23. Register Gitea secret

```
kubectl apply -f git-secret.yaml #1
```

Updating Tekton Task to update the container image

About yq

yq is a lightweight and portable command-line YAML processor. It can be used to query YAML documents or to modify them.

yq tool uses the following structure:

```
yq eval [flag] [expression] [yaml_file]
```

Given the following YAML document:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: myboot
    name: myboot
spec:
  replicas: 1
  selector:
    matchLabels:
      app: myboot
  template:
    metadata:
      labels:
        app: myboot
  spec:
    containers:
      - image: quay.io/rhdevelopers/myboot:v1
```

We can refer to the `image` field by using the following expression:

```
.spec.template.spec.containers[0].image.
```

To update the `image` field to a new value using `yq`:

```
yq eval -i '.spec.template.spec.containers[0].image = "quay.io/rh
```

Before configuring the pipeline, we need to modify the task defined previously with two new additions:

- Define a new resource for the GitOps repository, so it's automatically cloned.
- Define a step that updates the deployment definition with the container image tag created in the previous step.

```
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: welcome-service-app
spec:
  params:
    ...
  resources:
    inputs:
      - name: source
        type: git
      - name: gitops #1
        type: git
  steps:
    - name: maven-build
      ...
    - name: build-and-push-image
      ...
    - name: update-deployment-file
      image: quay.io/lordofthejars/image-updater:1.0.0 #2
      script: |
        #!/usr/bin/env ash

        cd /workspace/gitops #3

        git checkout -b newver #4
```

```

git config --global user.email "alex@example.com" #5
git config --global user.name "Alex"

yq eval -i '.spec.template.spec.containers[0].image = env

git add . #7
git commit -m "Update to $DESTINATION_IMAGE"
git push origin newver:master
env:
  - name: DESTINATION_IMAGE
    value: "$(inputs.params.imageName)"
volumes:
  - name: varlibc
    emptyDir: {}

```

Listing [8.24](#) shows the full version of the Task:

Listing 8.24. welcome-service-task.yaml

```

apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: welcome-service-app
spec:
  params:
    - name: contextDir
      description: the context dir within source
      default: .
    - name: imageName
      description: the container name url-group-artifact-tag
  resources:
    inputs:
      - name: source
        type: git
      - name: gitops
        type: git
  steps:
    - name: maven-build #1
      image: docker.io/maven:3.6.3-jdk-11-slim
      command:
        - mvn
      args:
        - clean
        - install
      workingDir: "/workspace/source/$(inputs.params.contextDir)"

```

```

- name: build-and-push-image
  image: quay.io/buildah/stable
  script: |
    #!/usr/bin/env bash
    buildah bud --layers -t $DESTINATION_IMAGE $CONTEXT_DIR
    buildah push --tls-verify=false $DESTINATION_IMAGE docker
  env:
    - name: DESTINATION_IMAGE
      value: "$(inputs.params.imageName)"
    - name: CONTEXT_DIR
      value: "/workspace/source/$(inputs.params.contextDir)"
  securityContext:
    runAsUser: 0
    privileged: true
  volumeMounts:
    - name: varlibc
      mountPath: /var/lib/containers
- name: update-deployment-file
  image: quay.io/lordofthejars/image-updater:1.0.0
  script: |
    #!/usr/bin/env ash
    cd /workspace/gitops #4
    git checkout -b newver #5
    git config --global user.email "alex@example.com"
    git config --global user.name "Alex"
    yq eval -i '.spec.template.spec.containers[0].image = "$D'
    git add .
    git commit -m "Update to $DESTINATION_IMAGE"
    git push origin newver:master #7
  env:
    - name: DESTINATION_IMAGE
      value: "$(inputs.params.imageName)"
  volumes:
    - name: varlibc
      emptyDir: {}

```

And in a terminal window apply the resource executing the command shown in listing [8.25](#):

Listing 8.25. Updates welcome service task

```
kubectl replace -f welcome-service-task.yaml #1
```



Important

Substitute replace with apply if you didn't apply the Tekton Task in the previous section.

Update Pipeline definition

The Pipeline definition requires an update on the resources part as shown in listing [8.26](#) to register the GitOps repository.

Listing 8.26. welcome-service-pipeline.yaml

```
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: welcome-deployment
spec:
  resources:
    - name: appSource
      type: git
    - name: appGitOps #1
      type: git
  ...
  tasks:
    - name: welcome-service-app
      ...
      resources:
        inputs:
          - name: source
            resource: appSource
          - name: gitops
            resource: appGitOps #2
```

And in a terminal window apply the resource executing the command shown in listing [8.27](#):

Listing 8.27. Update welcome service pipeline

```
kubectl replace -f welcome-service-pipeline.yaml
```



Important

Substitute replace with apply if you didn't apply the Tekton Task in the

previous section.

Update Pipeline Run

Finally, create a new Pipeline Run with the following changes:

- Increase the image tag number
- Sets the reference of the new Pipeline Resource.
- Configure the Service Account under the Tekton Pods will run.

The new PipelineRun is shown in listing [8.28](#):

Listing 8.28. welcome-service-gitops-resource-2.yaml

```
apiVersion: tekton.dev/v1beta1
kind: PipelineRun
metadata:
  name: welcome-pipeline-run-2 #1
spec:
  params:
    - name: imageTag
      value: "1.0.1" #2
  resources:
    ...
    - name: appGitOps
      resourceRef:
        name: welcome-service-git-gitops-source #3
  serviceAccountName: git-service-account #4
  pipelineRef:
    name: welcome-deployment
```

And in a terminal window apply the resource executing the command shown in listing [8.29](#):

Listing 8.29. Register pipeline run

```
kubectl apply -f welcome-service-gitops-resource-2.yaml #1
```

Inspecting Output

After applying the previous Pipeline Run, a new Pipeline instance is started

executing the following steps in Tekton:

- Clone service Git repository and GitOps repository from gitea.
- Build the service.
- Create a container image and push it to the container registry instance.
- Update the deployment file with the new image.
- Push the deployment file to the GitOps repository.

To view the log lines of current execution, run the command shown in listing [8.30](#):

Listing 8.30. List pipeline run

```
tkn pr logs -f #1
```

And the steps are shown in the log lines:

```
[welcome-service-app : git-source-source-8xjvv] { "level": "info",
[welcome-service-app : git-source-source-8xjvv] { "level": "info",
[welcome-service-app : git-source-gitops-v6f2q] { "level": "info",
[welcome-service-app : git-source-gitops-v6f2q] { "level": "info",
[welcome-service-app : maven-build] [INFO] Scanning for projects.

.....

[welcome-service-app : maven-build] [INFO] BUILD SUCCESS
[welcome-service-app : maven-build] [INFO] -----
[welcome-service-app : maven-build] [INFO] Total time:  01:12 min
[welcome-service-app : maven-build] [INFO] Finished at: 2021-04-1
[welcome-service-app : maven-build] [INFO] -----


[welcome-service-app : build-and-push-image] STEP 1: FROM registr
[welcome-service-app : build-and-push-image] Getting image source

.....


[welcome-service-app : build-and-push-image] Writing manifest to
[welcome-service-app : build-and-push-image] Storing signatures

[welcome-service-app : update-deployment-file] Switched to a new

....
```

```
[welcome-service-app : update-deployment-file] To http://gitea:30  
[welcome-service-app : update-deployment-file] 841cd10..472d777
```

After these steps are executed, Argo CD will detect the change on the `welcome-message-gitops/apps/app.yaml` deployment file. It will apply these changes triggering a rolling update of rolling update the service to the new version.



Important

The Argo CD controller can detect and sync the new manifests using a webhook or polling every 3 minutes. The polling strategy is the default one.

After waiting at most 3 minutes, Argo CD will detect the change and start the synchronization process applying the new deployment file.

Suppose we continuously monitor the status of the Pods. In that case, we'll see how the old *Welcome Message* Pod is terminated, and a new one is started automatically with the container created in the Tekton process. Execute the command shown in listing [8.31](#) continuously to inspect the change:

Listing 8.31. Get all pods

```
kubectl get pods
```

NAME	READY	STATUS	RESTARTS
gitea-deployment-7fbbf9c8b-xc27w	1/1	Running	0
name-generator-579ccdc5d5-mhzft	1/1	Running	2
postgresql-59ddd57cb6-tjrg2	1/1	Running	0
registry-deployment-64d49ff847-hljjg9	1/1	Running	2
vault-0	1/1	Running	0
welcome-message-55799f7dc9-c5j24	1/1	Running	0
welcome-message-6778c7978b-tqxkv	1/1	Terminating	0
welcome-pipeline-run-welcome-service-app-mj5z6-pod-mkkpx	0/5	Completed	0

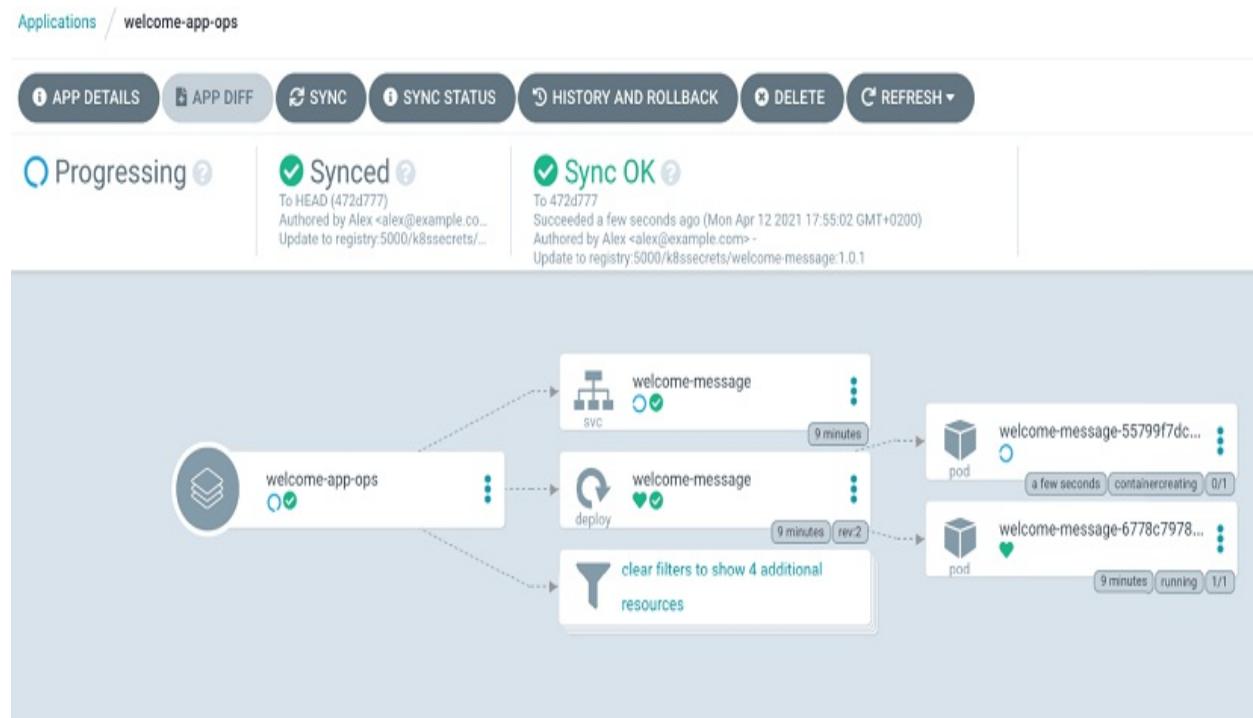
Describing the new deployed Pod like listing [8.32](#) shows that the new container is used:

Listing 8.32. Describe new welcome service pod

```
kubectl describe pod welcome-message-55799f7dc9-c5j24 #1  
...  
Controlled By: ReplicaSet/welcome-message-55799f7dc9  
Containers:  
welcome-message:  
Container ID: docker://fa8b45f46311819adb0fcfc5c8d8a17e4626792a  
Image:          registry:5000/k8ssecrets/welcome-message:1.0.  
...  
...
```

Figure 8.8 shows a screenshot of Argo CD dashboard where *Welcome Message* status is shown and if we look closely, we'll see rev:2 deployment is the current one as the service has been updated.

Figure 8.8. Argo CD dashboard



8.4 Summary

In this chapter, you've learned:

- Kubernetes Secrets are used in the application code (username,

passwords, API keys) and the CI/CD pipelines (username, passwords of external services). Enable Kubernetes Data Encryption at Rest to store encrypted secrets inside Kubernetes.* Continuous Delivery secrets need to be protected as any other secret. Use `SealSecrets` in Argo CD to store encrypted secrets in Git.

- Git is used as a single source of truth for the source code and the deployment scripts.
- Argo CD is a Kubernetes controller that allows you to implement GitOps in Kubernetes.

Appendix A. Tooling

In order deploy and manage a Kubernetes environment on your machine, several tools are required to be installed and configured.

A.1 minikube

`minikube` is a local Kubernetes cluster, focusing on making it easy to learn and develop for Kubernetes in a local environment. `minikube` relies on container/virtualization technology such as Docker, Podman, Hyperkit, Hyper-V, KVM, or VirtualBox to boot up a Linux machine with Kubernetes installed.

The VirtualBox virtualization tool is used for simplicity and a generic installation that works in most used operating systems (Microsoft Windows, Linux, Mac OS).

To install VirtualBox (if you haven't done it yet), first of all, open in a browser the following URL <https://www.virtualbox.org/>. When the webpage is opened, click on Downloads link located at the left menu as shown in the figure [A.1](#):

Figure A.1. VirtualBox Homepage annotating Downloads section



Then select your package based on the operating system to run the examples as shown in the figure [A.2](#):

Figure A.2. VirtualBox Downloads page with different packages



VirtualBox

Download VirtualBox

Here you will find links to VirtualBox binaries and its source code.

VirtualBox binaries

By downloading, you agree to the terms and conditions of the respective license.

If you're looking for the latest VirtualBox 6.0 packages, see [VirtualBox 6.0 builds](#). | until July 2020.

If you're looking for the latest VirtualBox 5.2 packages, see [VirtualBox 5.2 builds](#). | 2020.

VirtualBox 6.1.28 platform packages

- [Windows hosts](#)
- [OS X hosts](#)
- [Linux distributions](#)
- [Solaris hosts](#)
- [Solaris 11 IPS hosts](#)

The binaries are released under the terms of the GPL version 2.

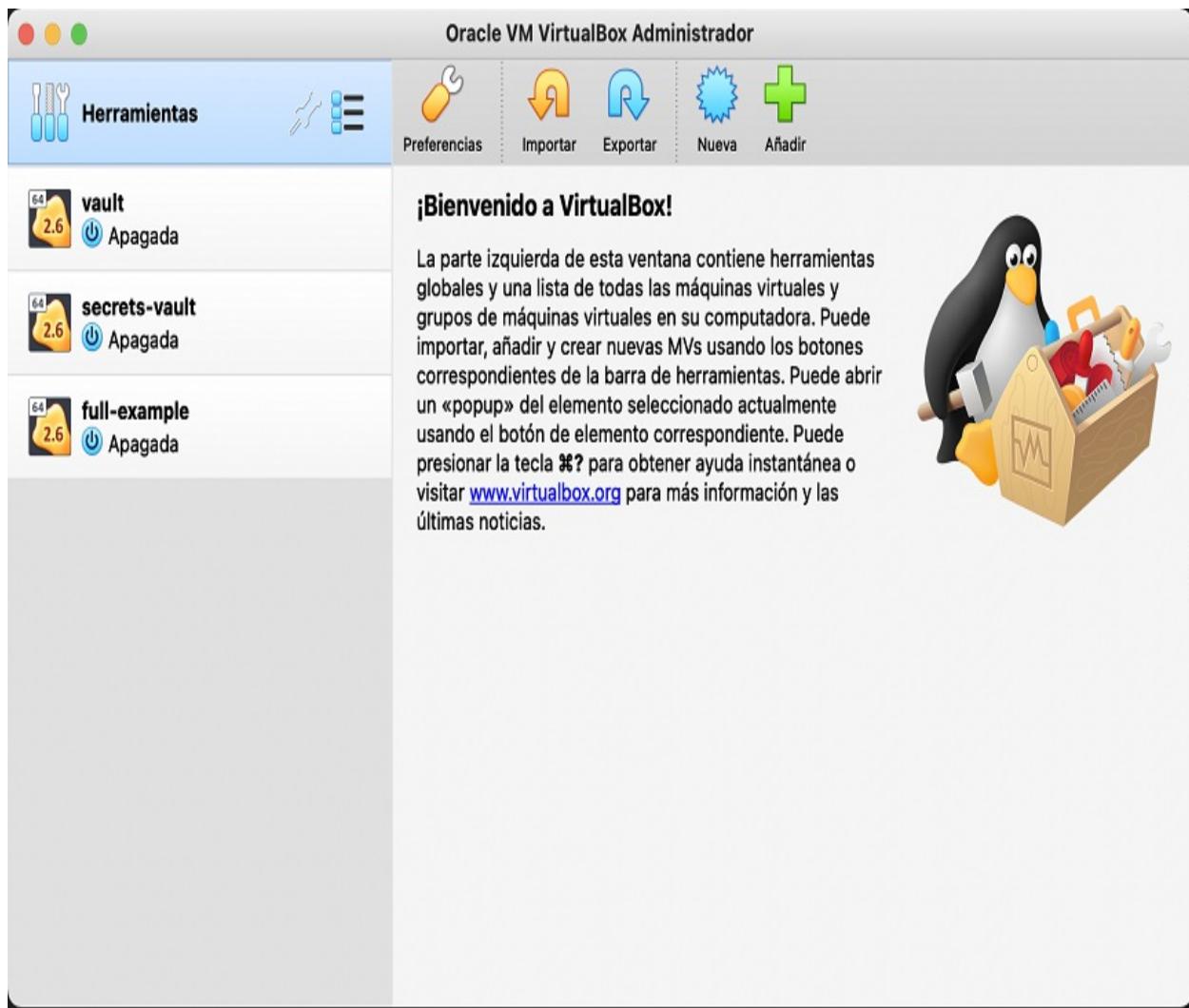
The downloading process will start storing the package on your local disk. When this process is finished, click the downloaded file to start the installation process as shown in the figure [A.3](#):

Figure A.3. VirtualBox installation window



You can use the default VirtualBox configuration values provided by the installation process or adapt to your requirements. After the installation finishes, you can validate the VirtualBox has been installed correctly by opening it. You can see in the figure [A.4](#) an example of the opening screen of VirtualBox with three machines installed.

Figure A.4. VirtualBox status window with 3 instances



A.2 kubectl

To interact with a Kubernetes cluster, we need to install the `kubectl` CLI tool. The best place to download and install `kubectl` is by opening the following URL <https://kubernetes.io/docs/tasks/tools/>. When the webpage is opened, click on the installation link depending on your platform as shown in the figure A.5:

Figure A.5. Kubectl homepage

 Search

Kubernetes Documentation / Tasks / Install Tools

Home

Getting started

Concepts

Tasks

Install Tools

Install and Set
Up kubectl on
Linux

Install and Set
Up kubectl on
macOS

Install and Set
Up kubectl on
Windows

Administer a

Install Tools

kubectl

The Kubernetes command-line tool, [kubectl](#), allows you to run commands against Kubernetes clusters. You can use kubectl to deploy applications, inspect and manage cluster resources, and view logs. For more information including a complete list of kubectl operations, see the [kubectl reference documentation](#).

kubectl is installable on a variety of Linux platforms, macOS and Windows. Find your preferred operating system below.

- [Install kubectl on Linux](#)
- [Install kubectl on macOS](#)
- [Install kubectl on Windows](#)

We will install Kubernetes 1.19.0; for this reason, it's essential to download the kubectl CLI version 1.19.0. To download a specific version, scroll down the page until a **Note** is shown how to install a particular version instead of the latest stable. Figure A.6 shows the part with the explanation to download a specific version:

Figure A.6. Downloading specific kubectl version. Replace v1.22.0 with v1.19.0

Install kubectl binary with curl on macOS

1. Download the latest release:

Intel

Apple Silicon

```
curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/darwin/amd64/kubectl"
```

Note:

To download a specific version, replace the `$(curl -L -s https://dl.k8s.io/release/stable.txt)` portion of the command with the specific version.

For example, to download version v1.22.0 on Intel macOS, type:

```
curl -LO "https://dl.k8s.io/release/v1.22.0/bin/darwin/amd64/kubectl"
```

And for macOS on Apple Silicon, type:

```
curl -LO "https://dl.k8s.io/release/v1.22.0/bin/darwin/arm64/kubectl"
```

With VirtualBox and kubectl installed, we can start the downloading the minikube version 1.17.1 to boot up the Kubernetes cluster.

Open in a browser the following URL

<https://github.com/kubernetes/minikube/releases/tag/v1.17.1>. When the webpage is loaded, unfold the Assets menu to find the minikube release specific to your platform.

Figure A.7 shows the GitHub Release page of minikube 1.17.1:

Figure A.7. Minikube is released as GitHub Release in Assets section

Release Notes

Version 1.17.1 - 2020-01-28

Features:

- Support for macOS/arm64 (Apple M1 Chip)
- Add new flag --user and to log executed commands #10106
- Unhide --schedule flag for scheduled stop #10274
- Make the ssh driver opt-in and not default #10269

Bugs:

- fixing debian and arch concurrent multiarch builds #9998
- configure the crictl yaml file to avoid the warning #10221

Thank you to our contributors for this release!

- Anders F Björklund
- BLasan
- Ilya Zuyev
- Jiefeng He
- Jorropo
- Medya Ghazizadeh
- Niels de Vos
- Priya Wadhwa
- Sharif Elgamal
- Steven Powell
- Thomas Strömberg
- andrzejsydor

Installation

See [Getting Started](#)

ISO Checksum

9fe214053aaff2352f150505f583e3d7f662e522d39db1839ce85f77ab51f1d5

► Assets 36

When the Assets menu is unfolded, click the `minikube` link depending on your platform. Figure A.8 shows the list of releases:

Figure A.8. Minikube is released in several platforms, download the one that fits yours

⋮ minikube-1.17.1-0.aarch64.rpm	21.1 MB
⋮ minikube-1.17.1-0.x86_64.rpm	12.8 MB
⋮ minikube-darwin-amd64	52.6 MB
⋮ minikube-darwin-amd64.sha256	65 Bytes
⋮ minikube-darwin-amd64.tar.gz	26.6 MB
⋮ minikube-darwin-arm64	51.7 MB
⋮ minikube-installer.exe	23.4 MB
⋮ minikube-linux-aarch64	49.9 MB
⋮ minikube-linux-amd64	53.2 MB
⋮ minikube-linux-amd64.sha256	65 Bytes
⋮ minikube-linux-amd64.tar.gz	29.3 MB
⋮ minikube-linux-arm	45.7 MB
⋮ minikube-linux-arm.sha256	65 Bytes
⋮ minikube-linux-arm64	49.9 MB
⋮ minikube-linux-arm64.sha256	65 Bytes
⋮ minikube-linux-ppc64le	50.1 MB
⋮ minikube-linux-ppc64le.sha256	65 Bytes
⋮ minikube-linux-s390x	52.3 MB
⋮ minikube-linux-s390x.sha256	65 Bytes
⋮ minikube-linux-x86_64	53.2 MB
⋮ minikube-windows-amd64	54.2 MB
⋮ minikube-windows-amd64.exe	54.2 MB

When the file is downloaded, rename it to `minikube` as the filename also contains the platform and the architecture. For example, `minikube-linux-amd64` is the `minikube` version for Linux for the 64 bits architectures.

With VirtualBox installed and the `minikube` file renamed, create a Kubernetes cluster by running the following command in a terminal window:

```
minikube start --kubernetes-version='v1.19.0' --vm-driver='virtua
```

And the output lines should be similar:

```
[vault] minikube v1.17.1 en Darwin 11.6
[KUBERNETES] Kubernetes 1.20.2 is now available. If you would like to upgr
[NEW] minikube 1.24.0 is available! Download it: https://github.com
[STAR] Using the virtualbox driver based on existing profile
[LIGHTBULB] To disable this notice, run: 'minikube config set WantUpdateNotice=false'

[THUMBS_UP] Starting control plane node vault in cluster vault
[RESTART] Restarting existing virtualbox VM for "vault" ...
[PREPARING] Preparando Kubernetes v1.19.0 en Docker 20.10.2...
[SEARCH] Verifying Kubernetes components...
[STAR] Enabled addons: storage-provisioner, default-storageclass

[!] /usr/local/bin/kubectl is version 1.21.3, which may have incom
  - Want kubectl v1.19.0? Try 'minikube kubectl -- get pods -A'
[OK] Done! kubectl is now configured to use "" cluster and "defaul
```

Appendix B. Installing and Configuring `yq`

Most operating systems have built in support for manipulating text within the terminal, such as sed and awk. These tools are great for fairly simple text manipulations, but can become cumbersome when working with structured serialization languages, such as YAML. `yq` (<https://mikefarah.gitbook.io/yq>) is a command line tool that provides support for querying and manipulating YAML based content and can be helpful when interacting with Kubernetes environments as the majority of the resources are expressed in YAML format. A similar and popular tool, called `jq`, provides similar capabilities for JSON formatted content, which is described in Chapter 4. This appendix describes the installation of `yq` along with several examples to confirm the successful installation of the tool.

B.1 Installing `yq`

`yq` is supported on multiple Operating Systems and can be installed using a variety of methods including through the use of a package manager or as a direct binary download from the project website (<https://github.com/mikefarah/yq/releases/latest>). The direct binary option is the most straightforward option as there are no external dependencies or prerequisites needed. Be sure to locate version 4 or higher of the tool as there was a significant rewrite from prior versions. The releases page will provide choices to download a compressed archive or the direct binary. Alternatively, you can use the terminal to download the binary to your local machine.



Warning

There is another tool, also called `yq`, which is available as a Python package and performs similar capabilities. Installing the incorrect tool will cause errors as there differences in the syntax and functionality between the two

tools.

The following command can be used to download the `yq` binary and place the binary in a directory on the PATH:

Listing B.1. Downloading `yq`

```
sudo curl -o /usr/bin/yq https://github.com/mikefarah/yq/releases  
sudo chmod +x /usr/bin/yq
```

The installation was successful if the following commands succeeds:

```
yq --version
```

If the command does not succeed, confirm the file was downloaded successfully, placed in a directory on the Operating System PATH and that the binary is executable.

B.2 `yq` By Example

`yq` can perform actions against YAML formatted content such as querying and manipulating values and is useful when working with Kubernetes manifests. `kubectl` does contain functionality to output in a variety of formats, such as JSONPath or golang templates. However, certain advanced features, like piping, are not available and require a more full purpose, dedicated tool.

To showcase some of the ways `yq` can be used to manipulate YAML content, create a file called `book-info.yaml` containing resource that we are familiar with, a Kubernetes secret.

Listing B.2. `book-info.yaml`

```
apiVersion: v1  
metadata:  
  name: book-info  
stringData:  
  title: Securing Kubernetes Secrets  
  publisher: Manning  
type: Opaque
```

```
kind: Secret
```

yq has two primary modes: evaluating a single document (with the `evaluate` or `e` subcommand) or multiple documents (with the `eval-all` or `ea` subcommand). Evaluating a single document is the most common mode so let's use it to query the contents of the `book-info.yaml` file created in listing [B.2](#).

Operations against YAML files use *expressions* to determine the specific actions to take. Since most YAML content uses nested content, these properties can be accessed using dot notation. For example, to extract the contents of the `title` field underneath `stringData`, the expression `.stringData.title` is used. The combination of the type of action to perform, such as `evaluate`, the *expression*, and the location of the YAML content are three components needed when using yq.

Now, use the following command to extract the `title` field as shown below:

```
yq eval '.stringData.title' book-info.yaml
```

More complex expressions can also be used to perform advanced operations. *Pipes* (`|`) can be used to chain expressions together so that the output from one expression becomes the input to another expression. For example, to determine the length of the `publisher` field underneath `stringData`, a *pipe* can be used to take the output of the `.stringData.publisher` expression and feed it into the `length` operator as shown below:

```
yq eval '.stringData.publisher | length' book-info.yaml
```

The result of the command should have returned 7.

In addition to extracting properties, yq can also be used to modify YAML content. Let's add a new property underneath `stringData` called `category` with a value of `Security`. The yq expression to accomplish this task is `'.stringData.category = "Security"'`. Execute the following command to add the `category` field.

```
yq eval '.stringData.category = "Security"' book-info.yaml
```

The following should have been returned as a result of the command:

Listing B.3. The output of updating YAML content using `yq`

```
apiVersion: v1
metadata:
  name: book-info
stringData:
  title: Securing Kubernetes Secrets
  publisher: Manning
  category: Security #1
type: Opaque
kind: Secret
```

Even though the output of the execution resulted in the addition of the new category field, it is important to note that the content of the `book-info.yaml` file was not modified. To update the content of the `book-info.yaml` field, the `-i` option must be used which will instruct `yq` to perform an *in place* update of the file. Execute the following command to perform an in place file update:

```
yq eval -i '.stringData.category = "Security"' book-info.yaml
```

Confirm the changes have been applied to the file.

The capabilities provided by `yq` to extract and manipulate make it a versatile tool and useful to have in your arsenal when working with any YAML formatted content.

Appendix C. Installing and Configuring Pip

Similar to most other programming languages, Python includes support for enabling reusable portions of code, such as statements and definitions, that can be included in other applications. These pieces of reusable code are known as *Modules*. Multiple modules can be organized together and included into a *Package*. The Standard Python library contains of an array of modules and packages which are fundamental to any application. However, the contents of the Standard Python Library does not cover every possible use case imaginable. This is where user defined packages and modules come in. As more and more individuals create customized packages, it becomes important that there be a way to easily distribute and consume these Python packages. The Python Package Index (PyPi) (<https://pypi.org/>) attempts to provide a solution to enable a centralized location for storing and discovering Python packages shared by the Python community. Python packages from both the centralized Python Package Index or a self hosted instance can be managed using `pip`, a package manager that facilitates the discovery, downloading and lifecycle management for Python packages.

C.1 Installing pip

Python packages located in the Python Package Index can be managed using the `pip` executable. Most recent distributions of Python (>=2.9.2 for Python 2 and >=3.4 for Python 3) using binary installations have `pip` preinstalled. If Python was installed through other methods, such as a package manager, `pip` may not be included. You can determine whether `pip` is installed by attempting to execute `pip` if on Python version 2 or `pip3` if on Python version 3. If an error is returned when executing the prior command, `pip` must be installed.

There are several ways that `pip` can be installed:

- The `ensurepip` Python 3 module

- The `get-pip.py` script
- Using a package manager

Let's install pip using the `ensurepip` Python module since it makes use of native Python constructs that are applicable across the majority of platforms.

Execute the following command to install pip.

```
python -m ensurepip --upgrade
```

Confirm that pip was installed by executing the `pip` command.

```
pip
```

If the command returned without error, pip was successfully installed.



Note

On some systems, an alias or symbolic link may be used to link `python3` and `python` executables to provide backwards compatibility or simplify the interaction with Python. The same situation also applies for pip. Specifying the `--version` flag when executing either `python`, `python3`, `pip` or `pip3` will confirm the specific version being used.

C.2 Basic Pip Operations

Before starting to work with pip, it is recommended that it along with a few of the supporting tools are updated to their latest versions. Obtaining the latest updates will ensure appropriate access to any required source archives. Execute the following command to update pip along with the `setuptools` and `wheel` packages to their latest version:

```
python -m pip install --upgrade pip setuptools wheel
```

With the necessary tool up to date, let's start working with pip.

One of the first steps for anyone using a package manager is to determine the software component to install. This may be known upfront, or could be

queried from the list of available components. The best location to search for packages is on the PyPi website (<https://pypi.org/>) which details each available package, their history, along with any dependencies.

There are countless packages available to install from the Python Index, and choosing the correct package can be a challenge. One of the most common use cases for Python developers is a need to making HTTP based requests. While Python does provide modules, such as *http.client*, constructing simple queries can be complex. The *requests* module attempts to simplify the use of HTTP based invocations.

Details related to the *requests* module can be found on the PyPi website, but let's use pip to install the package. Execute the following command to install the *requests* package using pip.

```
pip install requests
```

Information related to the installed package can be viewed using the *info* subcommand:

```
pip show requests
```

The response from the command is shown below:
app-name:

```
Name: requests
Version: 2.26.0
Summary: Python HTTP for Humans.
Home-page: https://requests.readthedocs.io
Author: Kenneth Reitz
Author-email: me@kennethreitz.org
License: Apache 2.0
Location: /usr/local/lib/python3.9/site-packages
Requires: certifi, charset-normalizer, idna, urllib3
Required-by:
```

With the *requests* package installed, a simple Python interactive session can be used to illustrate how it can be used. Execute the following commands to first start a Python interactive session.

```
python
```

Now, import the *requests*, query a remote address, and print the HTTP status.

Listing C.1. Using the requests package to query a remote server

```
import requests #1
response = requests.get("https://google.com") #2
print(response.status_code) #3
```

A 200 response code should be returned indicating a successful query to the remote HTTP server.

Type `exit()` to exit the Python interactive console.

Installed Python packages can also be installed *pip*. The `pip list` command can be used to determine which packages are currently installed. Once the desired package for removal has been identified, it can be removed by using the `pip uninstall` command. To remove the *requests* package previously installed, execute the following command:

Listing C.2. Removing the requests package

```
pip uninstall -y requests #1
```

Once the command completes successfully, the Python package has been removed.

Appendix D. Installing and Configuring Git

Git has become the de facto Version Control System for tracking the changes in files and is used frequently when working with Kubernetes content. It gained popularity due to its simplified branch management capabilities in comparison with other version control systems. Before beginning to use Git on a local machine, there are a series of steps that must first be completed.

D.1 Installing Git

To start working with Git content, the `git` executable must be installed. Support is available for `git` across most operating systems, include Linux, OSX and Windows and the steps are available on the Git website (<https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>).

When installing `git` on Linux, the easiest method is to use a package manager, such as `apt` or `dnf`.

On a Debian based Linux operating system, execute the following command in a terminal to install `git`

```
apt install git-all
```

On a RPM based operating system, such as Fedora or Red Hat Enterprise Linux, execute the following command in a terminal to install `git`

```
dnf install git-all
```

Confirm `git` was installed successfully by checking the version:

```
git --version
```

If the version was returned, `git` was successfully installed. If an error occurs, confirm the steps associated with the installation method used with

completed fully.

D.2 Configuring git

Even though Git can be used immediately after installation, it is recommended that additional steps be undertaken in order to be customize the git environment. Certain capabilities, such as committing code, will not be available without additional actions.

The `git config` subcommand is available to retrieve and set configuration options. These options can be specified at one of three levels:

- At a system level and specified within the `[path]/etc/gitconfig` file.
- At a user profile level in the `~/.gitconfig`. Can be targeted by specifying the `--global` option using the `git config` subcommand.
- At a repository level within the `.git/config` file. Can be targeted by specifying the `--local` option using the `git config` subcommand.

While there is an array of configurable options available in Git, there are two options that should be defined whenever Git is installed:

- Username
- Email address

These values will be associated with any commits that you perform. Failure to configure these values will result in the following error while attempting to perform a commit:

Listing D.1. Error message produced when no Git identity configured

Author identity unknown

*** Please tell me who you are.

Run

```
git config --global user.email "you@example.com" #1  
git config --global user.name "Your Name"
```

```
to set your accounts default identity.  
Omit --global to set the identity only in this repository.  
  
fatal: unable to auto-detect email address (got 'root@machine.(no
```

As the error in listing [D.1](#) describes, both the `user.email` and `user.name` variables should be configured. While these variables can be configured separately within each repository, for simplicity sakes, it is more straightforward to define at a global level and make appropriate modifications within individual repositories.

Execute the following commands to set the `user.email` and `user.name` to configure the required variables substituting your user details as appropriate:

```
git config --global user.name "Your Name"  
git config --global user.email "you@example.com"
```

Confirm the values are set appropriately by listing all global configurations:

```
git config --global -l
```

The following values should be returned:

```
user.name=Your Name  
user.email=you@example.com
```

At this point, your machine is ready to start fully interacting with the Git ecosystem.

Appendix E. Installing gpg

GPG (GNU Privacy Guard) is an open standards implementation of the proprietary PGP (Pretty Good Privacy) encryption scheme and is commonly used to encryption and decryption emails and file system content. The combination of symmetric-key cryptography and public-key cryptography enables a rapid and secure method of exchanging messages. Users must first generate a public/private key pair using GPG tooling to enable the encryption and decryption of messages. The private key is used during the encryption process while the public key is used at decryption time. Public key are shared with anyone who needs to decrypt the encrypted message and can also be hosted on Internet Key Servers to simplify how encrypted content is decrypted by a wider audience.

The creation of the public/private key pair along with encrypting and decrypting message is facilitated through the use of GPG tooling, specifically the gpg Command Line Interface.

E.1 Obtaining the GPG Tools

The GPG tools along with the gpg Command Line Interface is can be installed on most major operating systems and is available either as a direct download or from a package manager, such as apt, dnf, or brew.

On a Debian based Linux operating system, execute the following command in a terminal.

```
apt install gnupg
```

On a RPM based operating system, such as Fedora or Red Hat Enterprise Linux, execute the following command in a terminal:

```
dnf install gnupg
```

On an OSX operating system, execute the following command in a terminal:

```
brew install gpg
```

Confirm the gpg CLI was installed successfully by using the `--version` flag to assess the version of the tool.

Listing E.1. Displaying the GPG Version

```
gpg (GnuPG) 2.2.20
libgcrypt 1.8.5
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <https://gnu.org/licen
This is free software: you are free to change and redistribute it
There is NO WARRANTY, to the extent permitted by law.

Home: /root/.gnupg #1
Supported algorithms:
Pubkey: RSA, ELG, DSA, ECDH, ECDSA, EDDSA
Cipher: IDEA, 3DES, CAST5, BLOWFISH, AES, AES192, AES256, TWOFISH
          CAMELLIA128, CAMELLIA192, CAMELLIA256
Hash: SHA1, RIPEMD160, SHA256, SHA384, SHA512, SHA224
Compression: Uncompressed, ZIP, ZLIB, BZIP2
```

If a response similar to listing [E.1](#) is displayed, the GPG tools were successfully installed.

E.2 Generating a Public/Private Key pair

Prior to being able to encrypt content, a key pair must be generated. The process of generating a key pair was also discussed in Chapter 3, but will be addressed here again for completeness.

GPG keys can be created using the gpg CLI using one of the following flags:

- `--quick-generate-key` - Generates a key pair with the user providing an *USER-ID* along with expiration and algorithm details.
- `--generate-key` - Generates a key pair while prompting for a real name and email details.
- `--full-generate-key` - Dialogs for all of the possible key pair generation options

The option that you choose depends on your own requirements. The `--`

`generate-key` and `--full-generate-key` options support *batch* mode which enables a non-interactive method for key pair creation.

Create a new GPG key pair using the `--generate-key` flag:

```
gpg --generate-key
```

As soon as this command is executed, a home directory for GPG files is created within the `.gnupg` folder in your home directory. This location can be changed by specifying the `GNUPGHOME` environment variable.

Provide your name and email address at the prompts. You will then be asked to confirm the details. Press `o` to confirm the details.

Then, you are prompted to provide a passphrase to protect your key. The exercise in Chapter 3 advised not to create one to promote simplicity integrating each of the security tools that were being used. However, when creating GPG key pairs for other uses, it is recommended that a passphrase is provided.

Once a passphrase has been provided, a new key pair will be generated and details related to the key will be presented, as shown in listing [E.2](#).

Listing E.2. Displaying the GPG Version

```
pub    rsa2048 2020-12-31 [SC] [expires: 2022-12-31]
      53696D1AB6954C043FCBA478A23998F0CBF2A552
uid          [ultimate] John Doe <jdoe@example.com>
sub    rsa2048 2020-12-31 [E] [expires: 2022-12-31]
```

You can confirm that the name and email were added correctly from the values provided along with the algorithm, key size and expiration.

Now that the key pair has been generated, messages can be encrypted using the GPG set of tools.