Chapter 13

ConfigMaps and Secrets

Introduction

- While deploying an application, we may need to pass such runtime parameters like configuration details, passwords, etc. For example, let's assume we need to deploy ten different applications for our customers, and, for each customer, we just need to change the name of the company in the UI. Then, instead of creating ten different Docker images for each customer, we may just use the template image and pass the customers' names as a runtime parameter. In such cases, we can use the **ConfigMap API** resource. Similarly, when we want to pass sensitive information, we can use the **Secret API** resource. In this chapter, we will explore ConfigMaps and Secrets.

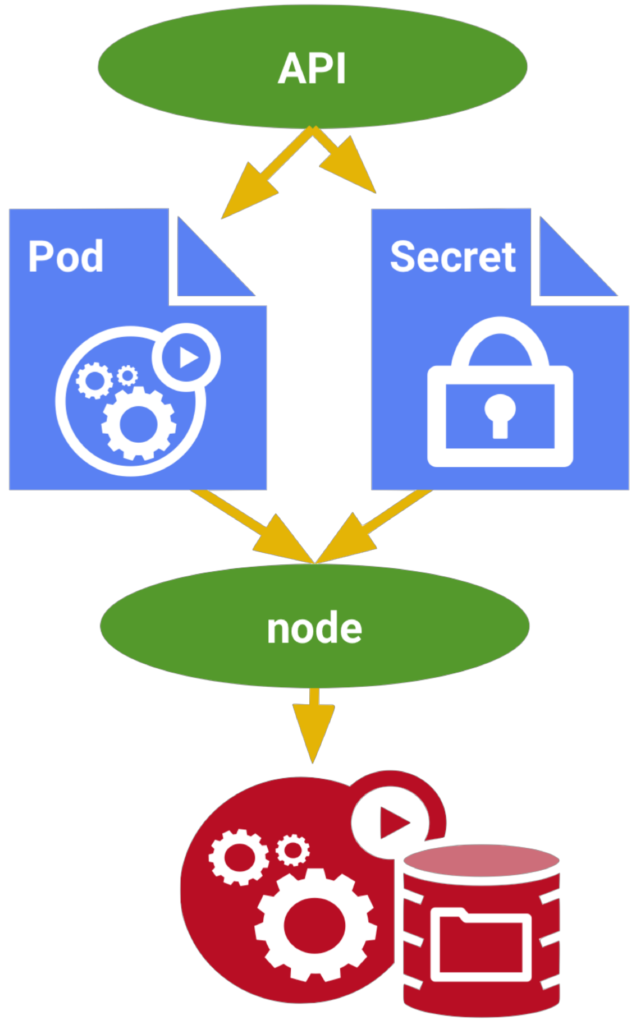
Learning Objectives

By the end of this chapter, you should be able to:

- Discuss configuration management for applications in Kubernetes using ConfigMaps.
- Share sensitive data (such as passwords) using Secrets.

What is a Secret?

Secrets are secure objects which store sensitive data, such as passwords, OAuth tokens, and SSH keys, in your clusters. Storing sensitive data in Secrets is more secure than plaintext ConfigMaps or in Pod specifications. Using Secrets gives you control over how sensitive data is used, and reduces the risk of exposing the data to unauthorized users.



ConfigMaps

หลุดพ้น

- ConfigMaps allow us to decouple the configuration details from the container image. Using ConfigMaps, we can pass configuration details as key-value pairs, which can be later consumed by Pods, or any other system components, such as controllers. We can create ConfigMaps in two ways:
    - From literal values
    - From files.

Create a ConfigMap from Literal Values and Get Its Details

A ConfigMap can be created with the **`kubectl create`** command, and we can get the values using the **`kubectl get`** command.

Create the ConfigMap

```
$ kubectl create configmap my-config --from-literal=key1=value1 --from-literal=key2=value2
configmap "my-config" created
```

Get the ConfigMap Details for `my-config`

```
$ kubectl get configmaps my-config -o yaml
apiVersion: v1
data:
  key1: value1
  key2: value2
kind: ConfigMap
metadata:
  creationTimestamp: 2017-05-31T07:21:55Z
  name: my-config
  namespace: default
  resourceVersion: "241345"
  selfLink: /api/v1/namespaces/default/configmaps/my-config
  uid: d35f0a3d-45d1-11e7-9e62-080027a46057
```

With the **`-o yaml`** option, we are requesting the **`kubectl`** command to spit the output in the YAML format. As we can see, the object has the **`ConfigMap kind`**, and it has the key-value pairs inside the data field. The name of **`ConfigMap`** and other details are part of the **`metadata`**field.

Create a ConfigMap from a Configuration File

First, we need to create a configuration file. We can have a configuration file with the content like:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: customer1
data:
  TEXT1: Customer1_Company
  TEXT2: Welcomes You
  COMPANY: Customer1 Company Technology Pct. Ltd.
```

in which we mentioned the **`kind`**, **`metadata`**, and **`data`** fields, which are targeted to connect with the **`v1`** endpoint of the API server.

If we name the file with the configuration above as **`customer1-configmap.yaml`**, we can then create the ConfigMap with the following command:

```
$ kubectl create -f customer1-configmap.yaml
configmap "customer1" created
```

Use ConfigMap Inside Pods

As an Environment Variable

We can get the values of the given key as environment variables inside a Pod. In the following example, while creating the Deployment, we are assigning values for environment variables from the **customer1** ConfigMap:

```
....
 containers:
      - name: rsvp-app
        image: teamcloudyuga/rsvpapp
        env:
        - name: MONGODB_HOST
          value: mongodb
        - name: TEXT1
          valueFrom:
            configMapKeyRef:
              name: customer1
              key: TEXT1
        - name: TEXT2
          valueFrom:
            configMapKeyRef:
              name: customer1
              key: TEXT2
        - name: COMPANY
          valueFrom:
            configMapKeyRef:
              name: customer1
              key: COMPANY
....
```

With the above, we will get the **TEXT1** environment variable set to **Customer1_Company**, **TEXT2** environment variable set to **Welcomes You**, and so on.

Secrets

Let's assume that we have a *Wordpress* blog application, in which our `wordpress` frontend connects to the `MySQL` database backend using a password. While creating the Deployment for `wordpress`, we can put down the `MySQL` password in the Deployment's YAML file, but the password would not be protected. The password would be available to anyone who has access to the configuration file.

In situations such as the one we just mentioned, the Secret object can help. With **Secrets, we can share sensitive information like passwords, tokens, or keys in the form of key-value pairs**, similar to ConfigMaps; thus, we can control how the information in a Secret is used, reducing the risk for accidental exposures. In Deployments or other system components, the Secret object is *referenced*, without exposing its content.

It is important to keep in mind that the Secret data is stored as plain text inside **etcd**. Administrators must limit the access to the API server and **etcd**.

Create the Secret with the 'kubectl create secret' Command

To create a Secret, we can use the **`kubectl create secret`** command:

```
 $ kubectl create secret generic my-password --from-
literal=password=mysqlpassword
```

The above command would create a secret called **`my-password`**, which has the value of the **`password`** key set to **`mysqlpassword`**.

'get' and 'describe' the Secret

Analyzing the **`get`** and **`describe`** examples below, we can see that they do not reveal the content of the Secret. The type is listed as **`Opaque`**.

```
$ kubectl get secret my-password
NAME           TYPE      DATA    AGE
my-password    Opaque    1       8m

$ kubectl describe secret my-password
Name:              my-password
Namespace:         default
Labels:            <none>
Annotations:       <none>
Type   Opaque
Data
====
password.txt:   13 bytes
```

Create a Secret Manually

We can also create a Secret manually, using the YAML configuration file. With Secrets, each object data must be encoded using **`base64`**. If we want to have a configuration file for our Secret, we must first get the **`base64`** encoding for our password:

```
$ echo mysqlpassword | base64
bXlzcWxwYXNzd29yZAo=
```

and then use it in the configuration file:

```
apiVersion: v1
kind: Secret
metadata:
  name: my-password
type: Opaque
data:
  password: bXlzcWxwYXNzd29yZAo=
```

Please note that **`base64`** encoding does not do any encryption, and anyone can easily decode it:

```
$ echo "bXlzcWxwYXNzd29yZAo=" | base64 --decode
```

Therefore, make sure you do not commit a Secret's configuration file in the source code.

Use Secrets Inside Pods

We can get Secrets to be used by containers in a Pod by mounting them as data volumes, or by exposing them as environment variables.

Using Secrets as Environment Variables

As shown in the following example, we can reference a Secret and assign the value of its key as an environment variable (`WORDPRESS_DB_PASSWORD`):

```
. . . . .
   spec:
     containers:
     - image: wordpress:4.7.3-apache
       name: wordpress
       env:
       - name: WORDPRESS_DB_HOST
         value: wordpress-mysql
       - name: WORDPRESS_DB_PASSWORD
         valueFrom:
           secretKeyRef:
             name: my-password
             key: password
. . . . .
```

Using Secrets as Files from a Pod

We can also mount a Secret as a Volume inside a Pod. A file would be created for each key mentioned in the Secret, whose content would be the respective value. For more details, you can study the Kubernetes documentation.

Learning Objectives (Review)

You should now be able to:

- Discuss configuration management for applications in Kubernetes using ConfigMaps.
- Share sensitive data (such as passwords) using Secrets.