ConfigMap

***What is ConfigMap and when is it used?*** 🤔  
Think of it as a *properties file for your application*. For example depending on your application environment (dev, int, prod) you will have a different database url or logging level. So for these kind of things you can use configMap.

The ***biggest advantage*** is that, with properties file, every time you modify it you have to rebuild and redeploy your application, whereas if you change configuration in configMap, ***you just need to restart the application pod/container***.

ConfigMap can be used by the application as a set of environmental variable values or as an actual configuration file.

Example ConfigMap with database connection configuration:

apiVersion: v1

kind: ConfigMap

metadata:

name: my-config

data:

db-host: cluster-mysql.database

db-port: 3306

db-name: my-db

apiVersion: v1

kind: Pod

metadata:

name: my-app

spec:

containers:

- name: my-app

image: my-app-image

env:

- name: DB\_HOST

valueFrom:

configMapKeyRef:

name: my-config

key: db-host

- name: DB\_PORT

valueFrom:

configMapKeyRef:

name: my-config

key: db-port

- name me: DB\_NAME

valueFrom:

configMapKeyRef:

name: my-config

key: db-name

Secrets 🔐 are also used in these 2 ways. Either as a value for env variables or as a secret file with credentials or a certificate etc mounted into a pod.

So for a better comparison, think of secrets as encrypted configMaps.

Example secret with key-value pairs:

apiVersion: v1

kind: Secret

metadata:

name: my-secret

type: Opaque

data:

db-user: dXNlcg==

db-password: cGFzc3dvcmQ

And you can use it the same way as ConfigMap in your application's configuration file:

apiVersion: v1

kind: Pod

metadata:

name: my-app

spec:

containers:

- name: my-app

image: my-app-image

env:

- name: DB\_USER

valueFrom:

secretKeyRef:

name: my-secret

key: db-user

- name: DB\_PASSWORD

valueFrom:

secretKeyRef:

name: my-secret

key: db-password

SECRETS:

**$ kubectl create secret generic** my-secret \**--from-file=**service\_account\_key=key.json \**--from-literal=**webhook\_token=sdfdgerww4dhgsf643 \**--from-literal=**slack\_token=sffrt64t7uk

If you want this secret to be added to a specific namespace or context add the --namespace or use-context arguments to this command. (Otherwise it will be added to the default namespace)

Kubernetes encodes all it’s secrets in **base64.**So if you want to create a secret from a .yaml file, you need to manually encode all the values in base64 — **or it won’t accept the values!**

$ **echo -n** some\_text\_to\_encode **| base64**c29tZV90ZXh0X3RvX2VuY29kZQ==

Similarly you can **decode a base64 encoded tex**t as well:

$ **echo** c29tZV90ZXh0X3RvX2VuY29kZQ== **| base64 -d**some\_text\_to\_encode

**my-secret.yaml**

apiVersion: v1kind: Secret  
metadata:  
 name: my-secret  
 namespace: defaulttype: Opaquedata:  
 service\_account\_key: eyBoZWxsbzogInNkZmFzZCIsCnBhc3N3b3JkOiAid2hhdCIgfQo=  
 slack\_token: c2ZmcnQ2NHQ3dWs=  
 webhook\_token: c2RmZGdlcnd3NGRoZ3NmNjQz

Now create this Secret by running:

$ **kubectl apply -f** my-secret.yamlsecret "my-secret" created

apiVersion: extensions/v1beta1  
kind: Deployment  
metadata:  
name: sample-app  
namespace: default  
spec:  
replicas: 1  
template:  
metadata:  
labels:  
app: sample-app  
spec  
containers:  
- name: sample-app  
image: gcr.io/google\_containers/defaultbackend:1.0  
ports:  
- containerPort: 8080  
**volumeMounts:  
- name: service-key  
mountPath: /root/key.json  
subPath: key.json  
volumes:  
- name: service-key  
secret:  
secretName: my-secret  
items:  
- key: service-account-key  
path: key.json**

To create the Secret containing the **MYSQL\_ROOT\_PASSWORD**, choose a password and convert it to base64:

***# The root password will be "KubernetesRocks!"*  
$ echo -n 'KubernetesRocks!' | base64  
S3ViZXJuZXRlc1JvY2tzIQ==**

Make a note of the encoded string. You need it to create the YAML file for the Secret:

**apiVersion: v1  
kind: Secret  
metadata:  
  name: mariadb-root-password  
type: Opaque  
data:  
  password: S3ViZXJuZXRlc1JvY2tzIQ==**

Save that file as **mysql-secret.yaml** and create the Secret in Kubernetes with the **kubectl apply** command:

**$ kubectl apply -f mysql-secret.yaml  
secret/mariadb-root-password created**

### View the newly created Secret

Now that you've created the Secret, use **kubectl describe** to see it:

**$ kubectl describe secret mariadb-root-password  
Name:         mariadb-root-password  
Namespace:    secrets-and-configmaps  
Labels:       <none>  
Annotations:  
Type:         Opaque  
  
Data  
====  
password:  16 bytes**

Note that the **Data** field contains the key you set in the YAML: **password**. The value assigned to that key is the password you created, but it is not shown in the output. Instead, the value's size is shown in its place, in this case, 16 bytes.

You can also use the **kubectl edit secret <secretname>** command to view and edit the Secret. If you edit the Secret, you'll see something like this:

***# Please edit the object below. Lines beginning with a '#' will be ignored,*  
*# and an empty file will abort the edit. If an error occurs while saving this file will be*  
*# reopened with the relevant failures.*  
*#*  
apiVersion: v1  
data:  
  password: S3ViZXJuZXRlc1JvY2tzIQ==  
kind: Secret  
metadata:  
  annotations:  
    kubectl.kubernetes.io/last-applied-configuration: |  
      {"apiVersion":"v1","data":{"password":"S3ViZXJuZXRlc1JvY2tzIQ=="},"kind":"Secret","metadata":{"annotations":{},"name":"mariadb-root-password","namespace":"secrets-and-configmaps"},"type":"Opaque"}  
  creationTimestamp: 2019-05-29T12:06:09Z  
  name: mariadb-root-password  
  namespace: secrets-and-configmaps  
  resourceVersion: "85154772"  
  selfLink: /api/v1/namespaces/secrets-and-configmaps/secrets/mariadb-root-password  
  uid: 2542dadb-820a-11e9-ae24-005056a1db05  
type: Opaque**

Again, the **data** field with the **password** key is visible, and this time you can see the base64-encoded Secret.

### Decode the Secret

Let's say you need to view the Secret in plain text, for example, to verify that the Secret was created with the correct content. You can do this by decoding it.

It is easy to decode the Secret by extracting the value and piping it to base64. In this case, you will use the output format **-o jsonpath=<path>** to extract only the Secret value using a JSONPath template.

***# Returns the base64 encoded secret string*  
$ kubectl get secret mariadb-root-password -o jsonpath='{.data.password}'  
S3ViZXJuZXRlc1JvY2tzIQ==  
  
*# Pipe it to `base64 --decode -` to decode:*  
$ kubectl get secret mariadb-root-password -o jsonpath='{.data.password}' | base64 --decode -  
KubernetesRocks!**

### Another way to create Secrets

You can also create Secrets directly using the **kubectl create secret** command. The MariaDB image permits setting up a regular database user with a password by setting the **MYSQL\_USER** and **MYSQL\_PASSWORD** environment variables. A Secret can hold more than one key/value pair, so you can create a single Secret to hold both strings. As a bonus, by using **kubectl create secret**, you can let Kubernetes mess with base64 so that you don't have to.

**$ kubectl create secret generic mariadb-user-creds \  
      --from-literal=MYSQL\_USER=kubeuser\  
      --from-literal=MYSQL\_PASSWORD=kube-still-rocks  
secret/mariadb-user-creds created**

Note the **--from-literal**, which sets the key name and the value all in one. You can pass as many **--from-literal** arguments as you need to create one or more key/value pairs in the Secret.

Validate that the username and password were created and stored correctly with the **kubectl get secrets** command:

***# Get the username*  
$ kubectl get secret mariadb-user-creds -o jsonpath='{.data.MYSQL\_USER}' | base64 --decode -  
kubeuser  
  
*# Get the password*  
$ kubectl get secret mariadb-user-creds -o jsonpath='{.data.MYSQL\_PASSWORD}' | base64 --decode -  
kube-still-rocks**