CREATING & USING CONFIGMAPS IN KUBERNETES

In programming, we use env files or separate configuration files to store settings, configurations, or variables that are required to execute the program. In Kubernetes, we can use ConfigMaps to achieve the same functionality.

To understand ConfigMap properly, you should have some knowledge of Kubernetes, pods, and basic Kubernetes cluster management.

(This article is part of our Kubernetes Guide. Use the right-hand menu to navigate.)

What is a ConfigMap?

A ConfigMap is a Kubernetes API object that can be used to store data as key-value pairs. Kubernetes pods can use the created ConfigMaps as a:

- Configuration file
- Environment variable
- Command-line argument

ConfigMaps provides the ability to make applications portable by decoupling environment-specific configurations from the containers.

Importantly, ConfigMaps are not suitable for storing confidential data. They do not provide any kind of encryption, and all the data in them are visible to anyone who has access to the file. (Kubernetes provides secrets that can be used to store sensitive information.)

Another consideration of ConfigMaps is the size of the file, as we are trying to store application

configuration ConfigMap files limited to 1MB. For larger data sets, it's better to use separate file mounts, databases, or file services.

ConfigMap example

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: example-configmap
  namespace: default
data:
  # Configuration Values are stored as key-value pairs
  system.data.name: "app-name"
  system.data.url: "https://app-name.com"
  system.data.type_one: "app-type-xxx"
  system.data.value: "3"
  # File like Keys
  system.interface.properties: |
    ui.type=2
    ui.color1=red
    ui.color2=green
```

In a ConfigMap, the required information can be stored in the data field. We can store values in two ways:

- As individual key pair properties
- In a granular format where they are fragments of a configuration format. (File Like Keys)

How to create ConfigMaps

ConfigMaps and pods go hand in hand as ConfigMaps can be used as environment variables and configuration information in a Kubernetes pod.

In this section, we will have a look at how to create ConfigMaps. Here are some notes before we get started:

- We will be using a windows environment with the windows subsystem for Linux (Ubuntu) as the terminal environment.
- The Docker desktop will be configured to facilitate a Kubernetes environment.
- We will be using the official sample files provided by Kubernetes to demonstrate the functionality of ConfigMap.

Creating ConfigMaps from directories

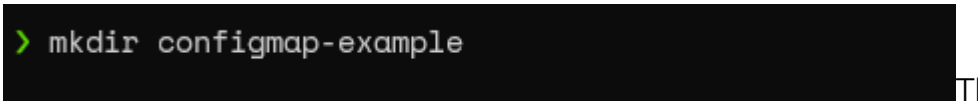
We can use the following command to create ConfigMap directories.

```
kubectl create configmap
```

It will look for appropriate files (regular files) within a specific directory that can be used to create a ConfigMap while ignoring any other file types (hidden files, subdirectories, symlinks, etc.)

First, let's create a directory using this command:

```
mkdir configmap-example
```



Then we'll download the required sample files to the directory. These files will be used to generate the ConfigMap.

```
wget https://kubernetes.io/examples/configmap/game.properties -O configmap-example/game.properties
```

```
wget https://kubernetes.io/examples/configmap/ui.properties -O configmap-example/ui.properties
```

```
❯ wget https://kubernetes.io/examples/configmap/game.properties -O configmap-example/game.properties
--2021-04-24 18:28:29--  https://kubernetes.io/examples/configmap/game.properties
Resolving kubernetes.io (kubernetes.io)... 147.75.40.148
Connecting to kubernetes.io (kubernetes.io)|147.75.40.148|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 157 [application/octet-stream]
Saving to: 'configmap-example/game.properties'

configmap-example/game.proper 100%[=============================================>]     157  --.-KB/s    in 0s

2021-04-24 18:28:29 (14.7 MB/s) - 'configmap-example/game.properties' saved [157/157]

❯ wget https://kubernetes.io/examples/configmap/ui.properties -O configmap-example/ui.properties
--2021-04-24 18:28:51--  https://kubernetes.io/examples/configmap/ui.properties
Resolving kubernetes.io (kubernetes.io)... 147.75.40.148
Connecting to kubernetes.io (kubernetes.io)|147.75.40.148|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 83 [application/octet-stream]
Saving to: 'configmap-example/ui.properties'

configmap-example/ui.properti 100%[=============================================>]      83  --.-KB/s    in 0s

2021-04-24 18:28:52 (5.61 MB/s) - 'configmap-example/ui.properties' saved [83/83]
```

Now let's have a look at the file contents using the following commands.

```
cat game.properties
cat ui.properties
```

```
❯ cat game.properties
enemies=aliens
lives=3
enemies.cheat=true
enemies.cheat.level=noGoodRotten
secret.code.passphrase=UUDDLRLRBABAS
secret.code.allowed=true
secret.code.lives=30%
❯ cat ui.properties
color.good=purple
color.bad=yellow
allow.textmode=true
how.nice.to.look=fairlyNice
```

When creating ConfigMaps using directories, the most important factor is that you have to correctly define the key-value pairs within each file.

After that, let's create the ConfigMap using the create configmap command.

`kubectl create configmap game-config-example --from-file=configmap-example/`

```
❯ kubectl create configmap game-config-example --from-file=configmap-example/
configmap/game-config-example created
```

This command will package the files within the specified directory and create a ConfigMap file. We can use the kubectl describe command to view the ConfigMap file.

`kubectl describe configmaps game-config-example`

```
❯ kubectl describe configmaps game-config-example
Name:         game-config-example
Namespace:    default
Labels:       <none>
Annotations:  <none>

Data
====
game.properties:
----
enemies=aliens
lives=3
enemies.cheat=true
enemies.cheat.level=noGoodRotten
secret.code.passphrase=UUDDLRLRBABAS
secret.code.allowed=true
secret.code.lives=30
ui.properties:
----
color.good=purple
color.bad=yellow
allow.textmode=true
how.nice.to.look=fairlyNice

Events:  <none>
```

We can get the ConfigMap in YAML format using the following command.

```
kubectl get configmaps game-config-example -o yaml
```

```
> kubectl get configmaps game-config-example -o yaml
apiVersion: v1
data:
  game.properties: |-
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
  ui.properties: |
    color.good=purple
    color.bad=yellow
    allow.textmode=true
    how.nice.to.look=fairlyNice
kind: ConfigMap
metadata:
  creationTimestamp: "2021-04-24T13:53:24Z"
  managedFields:
  - apiVersion: v1
    fieldsType: FieldsV1
    fieldsV1:
      f:data:
        .: {}
        f:game.properties: {}
        f:ui.properties: {}
    manager: kubectl-create
    operation: Update
    time: "2021-04-24T13:53:24Z"
  name: game-config-example
  namespace: default
  resourceVersion: "8324"
  selfLink: /api/v1/namespaces/default/configmaps/game-config-example
  uid: ca1fc37e-20aa-4650-a1c6-8c7c87f4387e
```

Creating ConfigMaps from files

In the same way we created ConfigMaps using directories, we can also create ConfigMaps using files by using the --from-file parameter to point to a single file in the kubectl create configmap command. So, let's create a ConfigMap using the game.properties file as shown below.

`kubectl create configmap game-config-example-2 --from-file=configmap-example/game.properties`

```
> kubectl create configmap game-config-example-2 --from-file=configmap-example/game.properties
configmap/game-config-example-2 created
```

`kubectl describe configmap game-config-example-2`

```
> kubectl describe configmap game-config-example-2
Name:         game-config-example-2
Namespace:    default
Labels:       <none>
Annotations:  <none>

Data
====
game.properties:
----
enemies=aliens
lives=3
enemies.cheat=true
enemies.cheat.level=noGoodRotten
secret.code.passphrase=UUDDLRLRBABAS
secret.code.allowed=true
secret.code.lives=30
Events:  <none>
```

We can define multiple --from-file arguments multiple times to create a single ConfigMap file using several different files.

`kubectl create configmap game-config-example-2 --from-file=c`

Creating ConfigMaps from an environment file

Kubernetes allows users to create ConfigMaps using env files. We can use the --from-env-file argument when defining an env file. This argument can also be used multiple times to define multiple env files.

When using env files, each line should adhere to the <name>=<value> format. Empty lines and comments will be ignored, while quotation marks will be a part of ConfigMap.

`cat configmap-example/game-env-file.properties`

```
> cat configmap-example/game-env-file.properties
enemies=aliens
lives=3
allowed="true"

# This comment and the empty line above it are ignored
```

`kubectl create configmap game-config-env-file-example --from-env-file=configmap-example/game-env-file.properties`

```
> kubectl create configmap game-config-env-file-example --from-env-file=configmap-example/game-env-file.properties
configmap/game-config-env-file-example created
```

`kubectl get configmap game-config-env-file-example -o yaml`

```
> kubectl get configmap game-config-env-file-example -o yaml
apiVersion: v1
data:
  allowed: '"true"'
  enemies: aliens
  lives: "3"
kind: ConfigMap
metadata:
  creationTimestamp: "2021-04-24T21:25:36Z"
  managedFields:
  - apiVersion: v1
    fieldsType: FieldsV1
    fieldsV1:
      f:data:
        .: {}
        f:allowed: {}
        f:enemies: {}
        f:lives: {}
    manager: kubectl-create
    operation: Update
    time: "2021-04-24T21:25:36Z"
  name: game-config-env-file-example
  namespace: default
  resourceVersion: "16396"
  selfLink: /api/v1/namespaces/default/configmaps/game-config-env-file-example
  uid: e90b2d34-458f-41d8-86a6-b1a1fd104bf6
```

Creating ConfigMap from a file with a predefined key

When creating a ConfigMap, we can use the following format in --from-file argument to define a key name that will overwrite the file name used in the data section.

`--from-file=<Key-Name>=<File-Path>` The following example demonstrates how to define a key while creating a ConfigMap.

`kubectl create configmap game-config-key-example --from-file=game-key-example-data=configmap-example/game.properties`

```
> kubectl create configmap game-config-key-example --from-file=game-key-example-data=configmap-example/game.
properties
configmap/game-config-key-example created
```

`kubectl get configmap game-config-key-example -o yaml`

```
> kubectl get configmap game-config-key-example -o yaml
apiVersion: v1
data:
  game-key-example-data: |-
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
kind: ConfigMap
metadata:
  creationTimestamp: "2021-04-24T21:48:18Z"
  managedFields:
  - apiVersion: v1
    fieldsType: FieldsV1
    fieldsV1:
      f:data:
        .: {}
        f:game-key-example-data: {}
    manager: kubectl-create
    operation: Update
    time: "2021-04-24T21:48:18Z"
  name: game-config-key-example
  namespace: default
  resourceVersion: "18755"
  selfLink: /api/v1/namespaces/default/configmaps/game-config-key-example
  uid: 8a27d189-6ddc-431d-a53f-11a1690a5ba6
```

Creating ConfigMaps from values

Another way to create ConfigMaps is to provide literal values as parameters in the create configmap command. For this, we can use the --from-literal argument to pass each key pair. This is especially handy when we need to create ConfigMaps on the fly.

`kubectl create configmap config-example-values --from-literal=example.value=one --from-literal=example-type=2 --from-literal=example.url="http://example.com"`

```
> kubectl create configmap config-example-values --from-literal=example.value=one --from-literal=example-type=2 --from-literal=example.url="http://example.com"
configmap/config-example-values created
```

`kubectl get configmap config-example-values -o yaml`

```
> kubectl get configmap config-example-values -o yaml
apiVersion: v1
data:
  example-type: "2"
  example.url: http://example.com
  example.value: one
kind: ConfigMap
metadata:
  creationTimestamp: "2021-04-24T21:56:54Z"
  managedFields:
  - apiVersion: v1
    fieldsType: FieldsV1
    fieldsV1:
      f:data:
        .: {}
        f:example-type: {}
        f:example.url: {}
        f:example.value: {}
    manager: kubectl-create
    operation: Update
    time: "2021-04-24T21:56:54Z"
  name: config-example-values
  namespace: default
  resourceVersion: "19646"
  selfLink: /api/v1/namespaces/default/configmaps/config-example-values
  uid: efac70b4-07b4-49fe-ad0e-9e1c493fb376
```

Utilizing ConfigMaps in pods

Now we have a basic understanding of how to create ConfigMaps. The next step is to use the created ConfigMaps for creating a Pod. In this section, we will create a simple ConfigMap and use it when creating a pod in Kubernetes.

As the first step, let's create a file named "app-basic.properties" and include two key-value pairs.

app-basic.properties

```
system.type="TESTING CONFIGMAP"
system.number=12345
```

We will create a ConfigMap named "app-basic-configmap" using the above file and the --from-file option.

```
kubectl create configmap app-basic-configmap --from-file=configmap-example/app-basic.properties
```

```
❯ kubectl create configmap app-basic-configmap --from-file=configmap-example/app-basic.properties
configmap/app-basic-configmap created
```

`kubectl get configmap app-basic-configmap -o yaml`

```
❯ kubectl get configmap app-basic-configmap -o yaml
apiVersion: v1
data:
  app-basic.properties: |
    system.type="TESTING CONFIGMAP"
    system.number=12345
kind: ConfigMap
metadata:
  creationTimestamp: "2021-04-24T22:11:36Z"
  managedFields:
  - apiVersion: v1
    fieldsType: FieldsV1
    fieldsV1:
      f:data:
        .: {}
        f:app-basic.properties: {}
    manager: kubectl-create
    operation: Update
    time: "2021-04-24T22:11:36Z"
  name: app-basic-configmap
  namespace: default
  resourceVersion: "21173"
  selfLink: /api/v1/namespaces/default/configmaps/app-basic-configmap
  uid: acb51647-1c3d-4db1-9a5d-d8d8427b8286
```

Finally, let's create a Pod referencing the newly created ConfigMap. We will be using the following YAML file to create the Pod.

example-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-example-pod
spec:
  containers:
    - name: configmap-example-busybox
      image: k8s.gcr.io/busybox
      command:
      envFrom:
        # Load the Complete ConfigMap
        - configMapRef:
```

```
            name: app-basic-configmap
  restartPolicy: Never
```

As you can see from the above example, We are going to load the complete ConfigMap we created to the Kubernetes Pod.

```
kubectl create -f example-pod.yaml
kubectl get pods
```

```
> kubectl create -f example-pod.yaml
pod/configmap-example-pod created
> kubectl get pods
NAME                    READY   STATUS      RESTARTS   AGE
configmap-example-pod   0/1     Completed   0          109s
```

```
kubectl logs configmap-example-pod | grep system.number
```

```
> kubectl logs configmap-example-pod | grep system.number
system.number=12345
```

The above result indicates that the ConfigMap "app-basic-configmap" was successfully loaded when creating the Kubernetes Pod.

Mapping keys from ConfigMaps to pods

Another way we can use ConfigMaps is to directly map values from ConfigMaps to the specific environmental variables in the Pod.

In this section, we will create two simple configmap files manually and load and map the values directly to the Kubernetes Pod. There, we will define the ConfigMaps as YAML files and then use the kubectl create command to generate the ConfigMaps.

application-defaults.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: application-configs
  namespace: default
data:
  app.value: "45000"
  app.type: test-application
  app.ui: web
```

application-logs.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: application-log-configs
```

```
  namespace: default
data:
  log_level: WARNING
  log_type: TEXT
```

```
kubectl create -f application-defaults.yaml
kubectl create -f application-logs.yaml
```

```
> kubectl create -f application-defaults.yaml
configmap/application-configs created
> kubectl create -f application-logs.yaml
configmap/application-log-configs created
```

example-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-example-pod
spec:
  containers:
    - name: configmap-example-busybox
      image: k8s.gcr.io/busybox
      command:
      env:
        - name: APPLICATION_TYPE
          valueFrom:
            configMapKeyRef:
              name: application-configs
              key: app.type
        - name: APPLICATION_UI_TYPE
          valueFrom:
            configMapKeyRef:
              name: application-configs
              key: app.ui
        - name: LOG_LEVEL
          valueFrom:
            configMapKeyRef:
              name: application-log-configs
              key: log_level
  restartPolicy: Never
```

In this configuration, we are mapping environmental variables to values within each ConfigMap.

The following is the basic structure for mapping a value. In the environment section in the YAML file, we define a variable name and reference the ConfigMap via the "configMapKeyRef" element using the "valueFrom." Here we will provide:

- The ConfigMap name
- The key where the value should be mapped from

```
env:
 - name: <<VARIABLE NAME>>
   valueFrom:
     configMapKeyRef:
       name: <<CONFIGMAP NAME>>
       key: <<CONFIGMAP KEY>>
```

Next, we will create the Pod using the kubectl create command as shown below.

```
kubectl create -f example-pod.yaml
kubectl get pods
```

```
> kubectl create -f example-pod.yaml
pod/econfigmap-example-pod created
> kubectl get pods
NAME                    READY   STATUS      RESTARTS   AGE
econfigmap-example-pod   0/1     Completed   0          11s
```

After successfully creating the Pod, we can explore the environment variables as shown below.

```
kubectl logs configmap-example-pod | grep APPLICATION_TYPE
kubectl logs configmap-example-pod | grep APPLICATION_UI_TYPE
kubectl logs configmap-example-pod | grep LOG_LEVEL
```

```
> kubectl logs configmap-example-pod | grep APPLICATION_TYPE

APPLICATION_TYPE=test-application
> kubectl logs configmap-example-pod | grep APPLICATION_UI_TYPE

APPLICATION_UI_TYPE=web
> kubectl logs configmap-example-pod | grep LOG_LEVEL

LOG_LEVEL=WARNING
```

The above results indicate that the values were correctly mapped to environment variables with custom names within the Kubernetes pod.

ConfigMap defined environment variables in pod commands

Another way we can utilize ConfigMap defined environmental variables is by using them in Pod Commands. This can be done for both the command and args elements in a YAML file using the $(VARIABLE_NAME) Kubernetes substitution syntax.

The following code block demonstrates how to use these environment variables in the command element using example-pod.yaml as the base.

```
apiVersion: v1
kind: Pod
metadata:
```

```
  name: configmap-example-pod
spec:
  containers:
    - name: configmap-example-busybox
      image: k8s.gcr.io/busybox
      command:
      env:
        - name: APPLICATION_TYPE
          valueFrom:
            configMapKeyRef:
              name: application-configs
              key: app.type
        - name: APPLICATION_UI_TYPE
          valueFrom:
            configMapKeyRef:
              name: application-configs
              key: app.ui
        - name: LOG_LEVEL
          valueFrom:
            configMapKeyRef:
              name: application-log-configs
              key: log_level
  restartPolicy: Never
```

In this instance, the environmental variables are identified at the execution of the command (at the container start), and they will be directly displayed in the terminal.

Adding ConfigMap data to a volume

Users can consume ConfigMaps by mounting the ConfigMap data into a Kubernetes volume. In the following example, we are mounting the “application-log-config” ConfigMap data to a volume called “config-volume” mounted in “/etc/config” in the container. Then we have configured a command that would list all the files within the /etc/config directory.

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-example-volume-pod
spec:
  containers:
    - name: configmap-volume-example-busybox
      image: k8s.gcr.io/busybox
      command:
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
  volumes:
    - name: config-volume
```

```
    configMap:
      name: application-log-configs
restartPolicy: Never
```

Mounted ConfigMaps are automatically updated. Kubelt will periodically check if the mounted ConfigMap is up to date and update the mount accordingly. However, this auto-update mechanism does not apply to volumes mapped as a SubPath volume.

That concludes this tutorial. Explore more Kubernetes topics with the right-hand menu.

ConfigMaps are essential to K8s clusters

In this article, we have learned about Kubernetes ConfigMaps, including multiple ways that can be used to create ConfigMaps and how to utilize ConfigMaps in a Kubernetes Pod. ConfigMaps are an essential part of any Kubernetes cluster, providing a robust method to store simple and frequently accessed application or container data.

Related reading

- BMC DevOps Blog
- Kubernetes Guide, with 20+ tutorials
- The State of Kubernetes Today
- Kubernetes Certifications: How & Why to Get Certified
- The 12-Factor App Methodology Explained
- How & Why To Become a Software Factory