**Getting Started with ConfigMaps**

Explore a bit about the ConfigMap Volume type.

**We'll cover the following**

* [The Need of the Hour](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7AvVGAY0q18#The-Need-of-the-Hour)
* [The ConfigMap](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7AvVGAY0q18#The-ConfigMap)

**The Need of the Hour**

ConfigMaps allow us to keep configurations separate from application images. Such separation is useful when other alternatives are not a good fit.

Almost every application can be fine-tuned through configuration. Traditional software deployment methods fostered the use of configuration files. However, we are not discussing traditional, but advanced, distributed, and immutable deployments through Kubernetes schedulers.

Usage of fundamentally new technology often requires new processes and different architecture, if we are to leverage its potential to its maximum. On the other hand, we cannot just throw away everything we have and start new.

We’ll have to try to balance new principles and legacy needs.

If we were to start developing a new application today, it would be, among other things, distributed, scalable, stateless, and fault tolerant. Those are some of today’s needs. While we might question how many of us know how to design an application with those quality attributes in mind, hardly anyone would argue against having any of them. What is often forgotten is the configuration. Which mechanism should your new application use to configure itself? *How about environment variables?*

Environment variables fit well into distributed systems. They are easy to define, and they are portable. They are the ideal choice for configuration mechanism of new applications.

However, in some cases, the configuration might be too complex for environment variables. In such situations, we might need to fall back to files (hopefully YAML). When those cases are combined with legacy applications which are almost exclusively using file-based configuration, it is evident that we cannot rely only on environment variables.

When a configuration is based on files, the best approach we can take is to bake the configuration into a Docker image. That way, we are going down the fully-immutable road. Still, that might not be possible when our application needs different configuration options for various clusters (e.g., testing and production).

We don’t want to convert this into a discussion that ends with “you do NOT need a different configuration for different environments”. Rather just assume that you might have an excellent reason for something like that. In such a case, baking config files into images will not do the trick. That’s where ConfigMaps comes into play.

**The ConfigMap**

ConfigMap allows us to “inject” configuration into containers. The source of the configs can be files, directories, or literal values. The destination can be files or environment variables.

**ConfigMap** takes a configuration from a source and mounts it into running containers as a *volume*.

That’s all the theory you’ll get up-front. Instead of a lengthy explanation, we’ll run some examples, and comment on the features we experience. We’ll be learning by doing, instead of learning by memorizing theory.

For setting ConfigMap you will use the following command after creating the cluster.

kubectl config current-context

However for practising the code on the platform context is already configured.

# Injecting Configuration from a Single File

Learn to inject configuration from a single file using the ConfigMap Volume.

**We'll cover the following**

* [Creating a ConfigMap](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JYRnmn7xom2#Creating-a-ConfigMap)
  + [Looking into the description](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JYRnmn7xom2#Looking-into-the-description)
* [Mounting the ConfigMap](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JYRnmn7xom2#Mounting-the-ConfigMap)
  + [Pod with mounted ConfigMap](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JYRnmn7xom2#Pod-with-mounted-ConfigMap)
  + [Creating the pod](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JYRnmn7xom2#Creating-the-pod)
  + [Verification](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JYRnmn7xom2#Verification)
* [Deleting the objects](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JYRnmn7xom2#Deleting-the-objects)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JYRnmn7xom2#Try-it-yourself)

## Creating a ConfigMap

In its purest, and probably the most common form, a ConfigMap takes a single file. For example, we can create one from the prometheus-conf.yml file.

kubectl create cm my-config --from-file=prometheus-conf.yml

We created a ConfigMap (cm) called my-config. The data of the map is the content of the prometheus-conf.yml file.

### Looking into the description

Let’s describe it, and see what we’ll get.

kubectl describe cm my-config

The **output** is as follows.

Name: my-config

Namespace: default

Labels: <none>

Annotations: <none>

Data

====

prometheus-conf.yml:

----

global:

scrape\_interval: 15s

scrape\_configs:

- job\_name: prometheus

metrics\_path: /prometheus/metrics

static\_configs:

- targets:

- localhost:9090

Events: <none>

The important part is located below “Data”. We can see the key which, in this case, is the name of the file (“prometheus-conf.yml”). Further down you can see the content of the file. If you look at the definition of “prometheus-conf.yml” in the playground at the end of this lesson, you’ll see that it is the same as what we saw from the ConfigMap’s description.

## Mounting the ConfigMap

ConfigMap is useless by itself. It is yet another Volume that, like all the others, needs a mount.

### Pod with mounted ConfigMap

Let’s take a look at a Pod specification defined in alpine.yml.

apiVersion: v1

kind: Pod

metadata:

name: alpine

spec:

containers:

- name: alpine

image: alpine

command: ["sleep"]

args: ["100000"]

volumeMounts:

- name: config-vol

mountPath: /etc/config

volumes:

- name: config-vol

configMap:

name: my-config

The essential sections are volumeMounts and volumes. Since volumeMounts are the same no matter the type of the Volume, there’s nothing special about it. We defined that it should be based on the volume called config-vol and that it should mount the path /etc/config. The volumes section uses configMap as the type and, in this case, has a single item name, that coincides with the name of the ConfigMap we created earlier.

### Creating the pod

Let’s create the Pod and see what happens.

kubectl create -f alpine.yml

kubectl get pods

Please confirm that the Pod is indeed running before moving on.

### Verification

Let’s see the content of the /etc/config directory inside the Pod’s only container.

kubectl exec -it alpine -- ls /etc/config

The **output** is as follows.

prometheus-conf.yml

The /etc/config now has a single file that coincides with the file we stored in the ConfigMap.

Let’s add -l to the ls command we executed a moment ago.

kubectl exec -it alpine -- ls -l /etc/config

The **output** is as follows.

total 0

lrwxrwxrwx 1 root root 26 Jun 20 06:04 prometheus-conf.yml -> ..data/prometheus-conf.yml

You’ll see that “prometheus-conf.yml” is a link to “…data/prometheus-conf.yml”.

If you dig deeper, you’ll see that “…data” is also a link to the directory named from a timestamp. And so on, and so forth. For now, the exact logic behind all the links and the actual files is not of great importance. From the functional point of view, there is prometheus-conf.yml, and our application can do whatever it needs to do with it.

Let’s confirm that the content of the file inside the container is indeed the same as the source file we used to create the ConfigMap.

kubectl exec -it alpine -- cat /etc/config/prometheus-conf.yml

The **output** should be the same as the contents of the prometheus-conf.yml file.

We saw one combination of ConfigMap. Let’s see what else we can do with it.

## Deleting the objects

We’ll remove the objects we created thus far and start over.

kubectl delete -f alpine.yml

The command to delete the ConfigMap is as follows.

kubectl delete cm my-config

## Try it yourself

A list of all the commands used in the lesson is given below.

kubectl create cm my-config --from-file=prometheus-conf.yml

kubectl describe cm my-config

kubectl create -f alpine.yml

kubectl get pods

kubectl exec -it alpine -- ls /etc/config

kubectl exec -it alpine -- ls -l /etc/config

kubectl exec -it alpine -- cat /etc/config/prometheus-conf.yml

kubectl delete -f alpine.yml

kubectl delete cm my-config

# Injecting Configurations from Multiple Files

Learn how to inject configuration from multiple files and from a directory.

**We'll cover the following**

* [Creating a ConfigMap from multiple files](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g2OvZOBENwZ#Creating-a-ConfigMap-from-multiple-files)
  + [Deleting the objects](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g2OvZOBENwZ#Deleting-the-objects)
* [Creating a ConfigMap from a directory](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g2OvZOBENwZ#Creating-a-ConfigMap-from-a-directory)
  + [Deleting the objects](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g2OvZOBENwZ#Deleting-the-objects)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g2OvZOBENwZ#Try-it-yourself)

## Creating a ConfigMap from multiple files

Let's see what happens when we execute the commands that follow.

kubectl create cm my-config --from-file=cm/prometheus-conf.yml --from-file=cm/prometheus.yml

kubectl create -f cm/alpine.yml

#Run the following command separately kubectl exec -it alpine -- ls /etc/config

We created a ConfigMap with two files, and we created the same Pod based on the alpine.yml definition. Finally, we output the list of files from the /etc/config directory inside the Pod’s only container. The **output** of the latter command is as follows.

prometheus-conf.yml prometheus.yml

We can see that both files are present in the container. That leads us to the conclusion that a ConfigMap can **contain multiple files**, and all will be created inside containers that mount it.

### Deleting the objects

Let’s delete the objects (again), and explore one more option behind the --from-file argument.

kubectl delete -f alpine.yml

#Run the following command separately to delete the configmap

kubectl delete cm my-config

### Deleting the objects

Let’s delete the objects (again), and explore one more option behind the --from-file argument.

kubectl delete -f alpine.yml

#Run the following command separately to delete the configmap

kubectl delete cm my-config

## Creating a ConfigMap from a directory

The --from-file argument might lead you to the conclusion that you can specify only a file path as its value. It works with directories as well. We can, for example, add all files from the cm directory to a ConfigMap.

kubectl create cm my-config --from-file=cm

We created my-config ConfigMap with the directory cm. Let’s describe it, and see what’s inside.

kubectl describe cm my-config

The **output** is as follows (content of the files is removed for brevity).

Name: my-config

Namespace: default

Labels: <none>

Annotations: <none>

Data

====

alpine-env-all.yml:

----

...

alpine-env.yml:

----

...

alpine.yml:

----

...

my-env-file.yml:

----

...

prometheus-conf.yml:

----

...

prometheus.yml:

----

...

Events: <none>

We can see that all six files from the cm directory are now inside the “my-config” ConfigMap.

We’re sure you already know what will happen if we create a Pod that mounts that ConfigMap. We’ll check it out anyways.

kubectl create -f cm/alpine.yml

#Run the below command separately after the "alpine" container is created

kubectl exec -it alpine -- ls /etc/config

The **output** of the latter command is as follows.

alpine-env-all.yml alpine.yml prometheus-conf.yml

alpine-env.yml my-env-file.yml prometheus.yml

All the files are there, and the time has come to move away from files and directories.

### Deleting the objects[**#**](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g2OvZOBENwZ#Deleting-the-objects)

So, let’s remove the objects first, and discuss the other sources.

kubectl delete -f alpine.yml

#Run the following command separately to delete the configmap

kubectl delete cm my-config

## Try it yourself[#](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g2OvZOBENwZ#Try-it-yourself)

A list of all the commands used in the lesson is given below.

kubectl create cm my-config --from-file=cm/prometheus-conf.yml --from-file=cm/prometheus.yml

kubectl create -f cm/alpine.yml

#Run the following command separately

kubectl exec -it alpine -- ls /etc/config

kubectl delete -f cm/alpine.yml

#Run the following command separately to delete the configmap

kubectl delete cm my-config

kubectl create cm my-config --from-file=cm

kubectl describe cm my-config

kubectl create -f cm/alpine.yml

#Run the below command separately after the "alpine" container is created

kubectl exec -it alpine -- ls /etc/config

kubectl delete -f cm/alpine.yml

#Run the following command separately to delete the configmap

kubectl delete cm my-config

# Injecting Configurations from Key/Value Literals

Learn how to inject configurations from key/value literals.

**We'll cover the following**

* [Creating ConfigMap using literals](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/B8R0nw9ngrN#Creating-ConfigMap-using-literals)
  + [Creating a Pod](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/B8R0nw9ngrN#Creating-a-Pod)
  + [Verification](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/B8R0nw9ngrN#Verification)
  + [Deleting the objects](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/B8R0nw9ngrN#Deleting-the-objects)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/B8R0nw9ngrN#Try-it-yourself)

## Creating ConfigMap using literals

Hopefully, even when our applications need different configs to work in distinct clusters, the differences are limited. Often, they should be limited to only a few key/value entries. In such cases, it might be easier to create ConfigMaps using --from-literal.

Let’s take a look at an example.

kubectl create cm my-config \

--from-literal=something=else \

--from-literal=weather=sunny

kubectl get cm my-config -o yaml

The **output** of the latter command is as follows (metadata is removed for brevity).

apiVersion: v1

data:

something: else

weather: sunny

kind: ConfigMap

...

We can see that two entries were added, one for each literal.

### Creating a Pod

Let’s create a Pod with the ConfigMap mounted.

kubectl create -f alpine.yml

#Wait a few seconds before executing the following command

kubectl exec -it alpine -- \

ls /etc/config

The **output** of the latter command is as follows.

something weather

Both files are there.

### Verification

Finally, let’s confirm that the content of one of the files is correct.

kubectl exec -it alpine -- \

cat /etc/config/something

The **output** is as follows.

Else

The --from-literal argument is useful when we’re in need to set a relatively small set of configuration entries in different clusters. It makes more sense to specify only the things that change, than all the configuration options.

The problem is that most of the existing applications are not designed to read separate configuration entries from different files.

On the other hand, if you’re sketching a new application, you might not choose this option either since you’d be able to develop it in a way that it reads environment variables. When faced with a choice between ConfigMap and environment variables, the **latter wins** most of the time.

All in all, we’re not sure how often you’ll be using the --from-literal argument. Maybe a lot, more likely not at all.

### Deleting the objects

There’s one more config source left to explore, so let’s delete the objects we’re currently running, and move on.

kubectl delete -f alpine.yml

#Run the below command separately to the configMap

kubectl delete cm my-config

That was it for injecting configuration from literals.

## Try it yourself

A list of all the commands used in the lesson is given below.

kubectl create cm my-config \

--from-literal=something=else \

--from-literal=weather=sunny

kubectl get cm my-config -o yaml

kubectl create -f alpine.yml

#Wait a few seconds before executing the following command

kubectl exec -it alpine -- \

ls /etc/config

kubectl exec -it alpine -- \

cat /etc/config/something

kubectl delete -f alpine.yml

#Run the below command separately to the configMap

kubectl delete cm my-config

# Injecting Configurations from Environment Files

Learn how to use environment files to inject configurations.

**We'll cover the following**

* [Looking into the definition](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/xlBAypRr4Dn#Looking-into-the-definition)
* [Creating the ConfigMap](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/xlBAypRr4Dn#Creating-the-ConfigMap)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/xlBAypRr4Dn#Try-it-yourself)

## Looking into the definition

Let’s take a look at the my-env-file.yml file.

something=else

weather=sunny

The file has the same key/value pairs as those we used in the example with --from-literal.

## Creating the ConfigMap

Let’s see what happens if we create a ConfigMap using that file as the source.

kubectl create cm my-config \

--from-env-file=my-env-file.yml

kubectl get cm my-config -o yaml

We created the ConfigMap using the --from-env-file argument, and we retrieved the ConfigMap in yaml format.

The **output** of the latter command is as follows (metadata is removed for brevity).

apiVersion: v1

data:

something: else

weather: sunny

kind: ConfigMap

...

We can see that there are two entries, each corresponding to key/value pairs from the file. The result is the same as when we created a ConfigMap using --from-literal arguments. Two different sources produced the same outcome.

If we used --from-file argument, the result would be as follows.

apiVersion: v1

data:

my-env-file.yml: |

something=else

weather=sunny

kind: ConfigMap

...

For --from-file, first delete the previous config using kubectl delete cm my-config.

All in all, --from-file reads the content of one or more files, and stores it using file names as keys. --from-env-file, assumes that content of a file is in key/value format, and stores each as a separate entry.

## Try it yourself

A list of all the commands used in the lesson is given below.

kubectl create cm my-config \

--from-env-file=my-env-file.yml

kubectl get cm my-config -o yaml

# Converting ConfigMap Output into Environment Variables

Explore the conversion of ConfigMap into environment variables.

**We'll cover the following**

* [Altering the perpetual process](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g738zv5QRwZ#Altering-the-perpetual-process)
  + [Looking into first definition](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g738zv5QRwZ#Looking-into-first-definition)
  + [Creating the pod](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g738zv5QRwZ#Creating-the-pod)
  + [Looking Into Second Definition](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g738zv5QRwZ#Looking-Into-Second-Definition)
  + [Creating the Pod](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g738zv5QRwZ#Creating-the-Pod)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g738zv5QRwZ#Try-it-yourself)

## Altering the perpetual process

All the examples we’ve seen so far are differing only in the source. The destination is always the same. No matter whether ConfigMap is created from a file, from a directory, from literal values, or from an environment file, it perpetually resulted in one or more files being injected into a container.

This time we’ll try something different. We’ll see how we can convert a ConfigMap into environment variables.

### Looking into first definition

Let’s take a look at a sample definition of alpine-env.yml file.

apiVersion: v1

kind: Pod

metadata:

name: alpine-env

spec:

containers:

- name: alpine

image: alpine

command: ["sleep"]

args: ["100000"]

env:

- name: something

valueFrom:

configMapKeyRef:

name: my-config

key: something

- name: weather

valueFrom:

configMapKeyRef:

name: my-config

key: weather

The major difference, when compared with alpine.yml, is that volumeMounts and volumes sections are gone. This time we have an env section.

Instead of a value field, we have valueFrom. Further on, we declared that it should get values from a ConfigMap (configMapKeyRef) named my-config. Since that ConfigMap has multiple values, we specified the key as well.

### Creating the pod

Let’s create the Pod.

kubectl create \

-f alpine-env.yml

#Wait for a few seconds before executing the below command

kubectl exec -it alpine-env – env

We created the Pod and executed the env command inside its only container. The output of the latter command, limited to the relevant parts, is as follows.

...

weather=sunny

something=else

...

There’s another, often more useful way to specify environment variables from a ConfigMap. Before we try it, we’ll remove the currently running Pod.

kubectl delete \

-f alpine-env.yml

### Looking Into Second Definition

Let’s take a look at yet another definition alpine-env-all.yml.

apiVersion: v1

kind: Pod

metadata:

name: alpine-env

spec:

containers:

- name: alpine

image: alpine

command: ["sleep"]

args: ["100000"]

envFrom:

- configMapRef:

name: my-config

The difference is only in the way environment variables are defined.

This time, the syntax is much shorter. We have envFrom, instead of the env section. It can be either configMapRef or secretRef. Since we did not yet explore Secrets, we’ll stick with the prior. Inside configMapRef is the name reference to the my-config ConfigMap.

### Creating the Pod

Let’s see it in action.

We created the Pod and retrieved all the environment variables from inside its only container. The output of the latter command, limited to the relevant parts, is as follows.

...

something=else

weather=sunny

...

The result is the **same** as before. The difference is only in the way we define environment variables.

With env.valueFrom.configMapKeyRef syntax, we need to specify each ConfigMap key separately. That gives us control over the scope and the relation with the names of container variables.

The envFrom.configMapRef converts all ConfigMap’s data into environment variables. That is often a better and simpler option if you don’t need to use different names between ConfigMap and environment variable keys. The syntax is short, and we don’t need to worry whether we forgot to include one of the ConfigMap’s keys.

## Try it yourself

A list of all the commands used in the lesson is given below.

kubectl create -f alpine-env.yml

#Wait for a few seconds before executing the below command

kubectl exec -it alpine-env -- env

kubectl delete \

-f alpine-env.yml

kubectl create \

-f alpine-env-all.yml

#Wait for a few seconds before executing the below command

kubectl exec -it alpine-env – env

# Defining ConfigMaps as YAML

Explore defining ConfigMaps as YAML files.

**We'll cover the following**

* [Looking into the YAML](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7DG1mOj5p4A#Looking-into-the-YAML)
* [Deploying Prometheus](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7DG1mOj5p4A#Deploying-Prometheus)
  + [Looking into the definition](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7DG1mOj5p4A#Looking-into-the-definition)
  + [Creating the application](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7DG1mOj5p4A#Creating-the-application)
* [Destroying Everything](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7DG1mOj5p4A#Destroying-Everything)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7DG1mOj5p4A#Try-it-yourself)
  + [Troubleshooting Tips for minikube](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/7DG1mOj5p4A#Troubleshooting-Tips-for-minikube)

All ConfigMaps we created so far were done through kubectl create cm commands. Everything in Kubernetes can be defined as YAML, and that includes ConfigMaps as well.

## Looking into the YAML

Even though we have not yet specified ConfigMaps as YAML, we have seen the format quite a few times throughout this chapter. Since we cannot be sure whether you can create a ConfigMap YAML file from memory, let’s make things easy on ourselves and use kubectl to output our existing my-config ConfigMap in YAML format.

kubectl get cm my-config -o yaml

The **output** is as follows.

apiVersion: v1

data:

something: else

weather: sunny

kind: ConfigMap

metadata:

name: my-config

...

Just as with any other Kubernetes object, ConfigMap has apiVersion, kind, and metadata. The data is where the maps are defined. Each must have a key and a value. In this example, there’s the key weather with the value sunny.

## Deploying Prometheus

Let’s try to translate that knowledge into the objects we’d need to deploy Prometheus.

### Looking into the definition

The **definition**, of prometheus.yml limited to the relevant parts, is as follows.

apiVersion: apps/v1

kind: Deployment

metadata:

name: prometheus

spec:

...

template:

...

spec:

containers:

...

volumeMounts:

- mountPath: /etc/prometheus

name: prom-conf

volumes:

- name: prom-conf

configMap:

name: prom-conf

...

apiVersion: v1

kind: ConfigMap

metadata:

name: prom-conf

data:

prometheus.yml: |

global:

scrape\_interval: 15s

scrape\_configs:

- job\_name: prometheus

metrics\_path: /prometheus/metrics

static\_configs:

- targets:

- localhost:9090

**Line 12-14:** The Deployment object defines the volumeMounts that references the prom-conf Volume, which is a configMap. We saw quite a few similar examples before.

**Line 25:** The ConfigMap object’s data section has only one key (prometheus.yml). Once this ConfigMap is mounted as a volume, the name of the file will be the same as the key (prometheus.yml).

The value has a bit of “special” syntax. Unlike the previous example where the value was a single word written directly after the colon, the structure of the value is now a bit more complex. To be more precise, it contains multiple lines.

When working with a large value, we can start with the pipe sign (|). Kubernetes will interpret the value as “everything that follows, as long as it is indented.” You’ll notice that all the lines of the value are at least two spaces to the right of the beginning of the key (prometheus.yml). If you’d like to insert an additional key, all you’d need to do is to add it on the same level (indentation), as the other prometheus.yml.

### Creating the application

Let’s create the application and confirm that everything works as expected.

kubectl get cm my-config -o yaml

kubectl create -f prometheus.yml

kubectl rollout status deploy prometheus

# please wait for deployments to successfully roll out before running the

# following command:

kubectl port-forward service/prometheus 3000:9090 --address 0.0.0.0

#click the link beside run button

## Destroying Everything

For now, we’ll destroy the cluster we used in this chapter.

k3d cluster delete mycluster –all

## Try it yourself

A list of all the commands used in the lesson is given below.

kubectl get cm my-config -o yaml

kubectl create -f prometheus.yml

kubectl rollout status deploy prometheus

# please wait for deployments to successfully roll out before running the

# following command:

kubectl port-forward service/prometheus 3000:9090 --address 0.0.0.0

#click the link beside run button

k3d cluster delete mycluster –all

### Troubleshooting Tips for minikube

If you were to create the deployment in minikube locally, you will need the following commands to set the appropriate IP:

cat prometheus.yml | sed -e \

"s/192.168.99.100/$(minikube ip)/g" \

| kubectl create -f -

kubectl rollout status deploy prometheus

open http://$(minikube ip)/prometheus/targets

We created the objects (with the help of sed transformations), we waited until the Deployment rolled out, and, finally, we opened the Prometheus targets screen in a browser. The result should be a green target towards Prometheus’ internal metrics.

# A Plea NOT to Use ConfigMaps!

Learn the best way to configure our applications.

**We'll cover the following**

* [Why No ConfigMaps?](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g76L8ljKD3Y#Why-No-ConfigMaps?)
* [Finding out the Best Way](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g76L8ljKD3Y#Finding-out-the-Best-Way)
* [When to Use ConfigMaps?](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/g76L8ljKD3Y#When-to-Use-ConfigMaps?)

## Why No ConfigMaps?

ConfigMaps, in our experience, are overused.

If you have a configuration that is the same across multiple clusters, or if you have only one cluster, all you should do is include it in your Dockerfile and forget it ever existed. When there are no variations of a config, there’s no need to have a configuration file. At least, not outside an immutable image.

Unfortunately, that is not always the case. To be more precise, it’s almost never the case. We tend to make things more complicated than they should be. That, among other things, often means an endless list of configuration options hardly anyone ever uses. Still, some things usually do change, from one cluster to another, and we might need to look into alternatives to configurations baked into images.

## Finding out the Best Way

Design your new applications to use a combination of configuration files and environment variables. Make sure that the default values in a configuration file are sensible and applicable in most use-cases. Bake it into the image. When running a container, declare only the environment variables that represent the differences of a specific cluster. That way, your configuration will be portable and simple at the same time.

What if your application is not new and does not support configuration through environment variables? Refactor it so that it does. It shouldn’t be hard to add the ability to read a few environment variables. Keep in mind that you don’t need all the settings, but only those that differ from one cluster to another. It would be hard to imagine that such a trivial request would be complex or time-consuming. If it is, you might have more significant issues to fix before even thinking about putting your application into a container.

## When to Use ConfigMaps?

Still, configuration files will not disappear. No matter which strategy we choose, each image should have a copy of them with sensible default values. Maybe, we can put in an extra effort and change the application, so that configuration entries are loaded from two locations. That way, we can load the default values from one, and only the differences from the other. That would, at least, reduce the need to have to specify more than the minimum required for each cluster. In such a case, ConfigMap’s --from-literal and --from-env-file sources are an excellent choice.

When everything else fails, the --from-file source is your friend. Just make sure that ConfigMap is not defined in the same file as the objects that mount it. If it is, it would mean that they could be used only inside one cluster. Otherwise, we’d be deploying the same config, and we should go back to the initial idea of having it baked into the image together with the application.

Do not let this pessimism discourage you from using ConfigMaps. They are very handy, and you should adopt them. Our intent to discourage you from doing so had the intention of making you think of alternatives, not to tell you never to use ConfigMaps.

## The Similarities

The mechanisms behind Kubernetes ConfigMaps and Docker Swarm Configs are almost the same. At least, from the functional perspective.

* Both allow us to store some literal texts in the scheduler’s internal data store, and both enable us to add them to containers.
* The syntax is equally simple and straightforward in both cases. Still, there are a few differences.

Docker swarm vs Kubernetes

## The Differences

### Docker

Docker is good at preventing people from doing silly things (the politically correct version of the word stupid). An example would be an attempt to delete a configuration. It cannot be deleted if there are Docker services that reference the configuration. Only after all the services that reference it are removed, are we allowed to remove the configuration source. Kubernetes, on the other hand, will let us delete a ConfigMap object without even a hint about the consequences.

### Kubernetes

Kubernetes ConfigMap, on the other hand, provides a wider variety of options. While a Docker Swarm configuration can be created only from a file or stdin, the Kubernetes equivalent can be generated from a file, from a directory, from a literal value, and from files with environment variables. Each of those sources can be used multiple times. We can even combine them. Further on, Kubernetes ConfigMaps can be converted not only to files, but also to environment variables. Flexibility and extra functionalities are available both at the source and the destination end-points.

### Conclusion

Docker Swarm **wins** a user experience point. Kubernetes **gains a star** for providing more choices. Neither has a significant difference that would warrant a win, so we’re proclaiming it a tie.

There are many more features worthwhile comparing. We’re not yet finished. Stay tuned for more.

# What's Next?

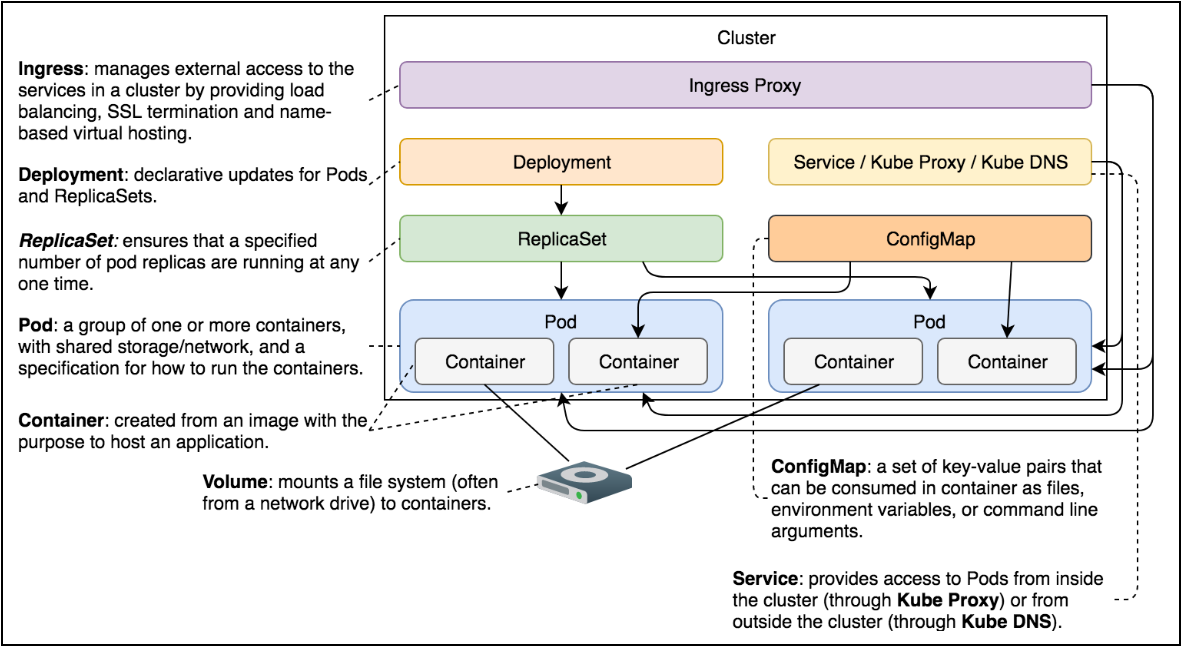
Recap what we have learned so far and what we are going to learn next.

**We'll cover the following**

* [Summary](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/3wWoJg0YLQA#Summary)

## Summary

In this chapter, we explored ConfigMaps and learned to inject configurations from various sort of sources including single files, multiple files, literals, and environment files. We also discussed the ways of configuring your applications.



The above illustration shows and defines Kubernetes components explored so far.

ℹ️ If you’d like to know more about ConfigMaps, please explore [ConfigMap v1 core](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.24/" \l "configmap-v1-core" \t "_blank) API documentation.