



Concepts

[HOME](#) [GETTING STARTED](#) [CONCEPTS](#) [TASKS](#) [TUTORIALS](#) [REFERENCE](#) [CONTRIBUTE](#)



ConfigMaps



A ConfigMap is an API object used to store non-confidential data in key-value pairs. Pods can consume ConfigMaps as environment variables, command-line arguments, or as configuration files in a volume.

A ConfigMap allows you to decouple environment-specific configuration from your container images, so that your applications are easily portable.

Caution: ConfigMap does not provide secrecy or encryption.

If the data you want to store are confidential, use a [Secret](#) rather than a ConfigMap, or use additional (third party) tools to keep your data private.

- [Motivation](#)
- [ConfigMap object](#)
- [ConfigMaps and Pods](#)
- [What's next](#)


Motivation

Use a ConfigMap for setting configuration data separately from application code.

For example, imagine that you are developing an application that you can run on your own computer (for development) and in the cloud (to handle real traffic). You write the code to look in an environment variable named `DATABASE_HOST`. Locally, you set that variable to `localhost`. In the cloud, you set it to refer to a Kubernetes [Service](#) that exposes the database component to your cluster.

This lets you fetch a container image running in the cloud and debug the exact same code locally if needed.

ConfigMap object

A ConfigMap is an API [object](#) that lets you store configuration for other objects to use. Unlike most Kubernetes objects that have a `spec`, a ConfigMap  `data` section to store items (keys) and their values.

The name of a ConfigMap must be a valid [DNS subdomain name](#).

ConfigMaps and Pods

You can write a Pod `spec` that refers to a ConfigMap and configures the container(s) in that Pod based on the data in the ConfigMap. The Pod and the ConfigMap must be in the same namespace.

Here's an example ConfigMap that has some keys with single values, and other keys where the value looks like a fragment of a configuration format.

```
apiVersion: v1
kind: ConfigMap
metadata:
  Name: game-demo
data:
  # property-like keys; each key maps to a simple value
  player_initial_lives: 3
  ui_properties_file_name: "user-interface.properties"
  #
  # file-like keys
  game.properties: |
    enemy.types=aliens,monsters
    player.maximum-lives=5
  user-interface.properties: |
    color.good=purple
    color.bad=yellow
    allow.textmode=true
```

There are four different ways that you can use a ConfigMap to configure a container inside a Pod:

1. Command line arguments to the entrypoint of a container
2. Environment variables for a container
3. Add a file in read-only volume, for the application to read
4. Write code to run inside the Pod that uses the Kubernetes API to read a ConfigMap

These different methods lend themselves to different ways of modeling the data being consumed. For the first three methods, the kubelet uses the data from the Secret when it launches container(s) for a Pod.

The fourth method means you have to write code to read the Secret and its data. However, because you're using the Kubernetes API directly, your application can subscribe to get updates whenever the ConfigMap

changes, and react when that happens. By accessing the Kubernetes API directly, this technique also lets you access a ConfigMap in a different namespace.




Here's an example Pod that uses values from `game-demo` to configure a Pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-demo-pod
spec:
  containers:
    - name: demo
      image: game.example/demo-game
      env:
        # Define the environment variable
        - name: PLAYER_INITIAL_LIVES # Notice that the case is different here
                                          # from the key name in the ConfigMap.
          valueFrom:
            configMapKeyRef:
              name: game-demo # The ConfigMap this value comes from.
              key: player_initial_lives # The key to fetch.
        - name: UI_PROPERTIES_FILE_NAME
          valueFrom:
            configMapKeyRef:
              name: game-demo
              key: ui_properties_file_name
      volumeMounts:
        - name: config
          mountPath: "/config"
          readOnly: true
  volumes:
    # You set volumes at the Pod level, then mount them into containers inside that Pod
    - name: config
      configMap:
        # Provide the name of the ConfigMap you want to mount.
        name: game-demo
```

A ConfigMap doesn't differentiate between single line property values and multi-line file-like values. What matters how Pods and other objects consume those values. For this example, defining a volume and mounting it inside the `demo` container as `/config` creates four files:

- `/config/player_initial_lives`
- `/config/ui_properties_file_name`
- `/config/game.properties`
- `/config/user-interface.properties`

If you want to make sure that `/config` only contains files with a `.properties` extension, use two different ConfigMaps, and refer to both ConfigMaps in the  for a Pod. The first ConfigMap defines `player_initial_lives` and `ui_properties_file_name`. The second ConfigMap defines the files that the kubelet places into `/config`.

Note:

The most common way to use ConfigMaps is to configure settings for containers running in a Pod in the same namespace. You can also use a ConfigMap separately.

For example, you might encounter addons or operators that adjust their behavior based on a ConfigMap.

What's next

- Read about [Secrets](#).
- Read [Configure a Pod to Use a ConfigMap](#).
- Read [The Twelve-Factor App](#) to understand the motivation for separating code from configuration.

Feedback

Was this page helpful?

Yes

No

Create an Issue

Edit This Page

Page last modified on April 05, 2020 at 11:39 AM PST by [Add ConfigMap concept page \(Page History\)](#)