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

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.

> **Note:**
> A ConfigMap is not designed to hold large chunks of data. The data stored in a ConfigMap cannot exceed 1 MiB. If you need to store settings that are larger than this limit, you may want to consider mounting a volume or use a separate database or file service.

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 has `data` and `binaryData` fields. These fields accept key-value pairs as their values. Both the `data` field and the `binaryData` are optional. The `data` field is designed to contain UTF-8 strings while the `binaryData` field is designed to contain binary data as base64-encoded strings.

The name of a ConfigMap must be a valid DNS subdomain name.

Each key under the `data` or the `binaryData` field must consist of alphanumeric characters, `-`, `_` or `.`. The keys stored in `data` must not overlap with the keys in the `binaryData` field.

Starting from v1.19, you can add an `immutable` field to a ConfigMap definition to create an immutable ConfigMap.

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.

> **Note:**
> The `spec` of a static Pod cannot refer to a ConfigMap or any other API objects.

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.

```yaml
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. Inside a container command and args
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 ConfigMap when it launches container(s) for a Pod.

The fourth method means you have to write code to read the ConfigMap 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:

configmap/configure-pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: configmap-demo-pod
spec:
  containers:
    - name: demo
      image: alpine
      command: ["sleep", "3600"]
      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
      # An array of keys from the ConfigMap to create as files
      items:
      - key: "game.properties"
        path: "game.properties"
      - key: "user-interface.properties"
        path: "user-interface.properties"
```

A ConfigMap doesn't differentiate between single line property values and multi-line file-like values. What matters is how Pods and other objects consume those values.

For this example, defining a volume and mounting it inside the `demo` container as `/config` creates two files, `/config/game.properties` and `/config/user-interface.properties`, even though there are four keys in the ConfigMap. This is because the Pod definition specifies an `items` array in the `volumes` section. If you omit the `items` array entirely, every key in the ConfigMap becomes a file with the same name as the key, and you get 4 files.

Using ConfigMaps

ConfigMaps can be mounted as data volumes. ConfigMaps can also be used by other parts of the system, without being directly exposed to the Pod. For example, ConfigMaps can hold data that other parts of the system should use for configuration.

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.

Using ConfigMaps as files from a Pod

To consume a ConfigMap in a volume in a Pod:

1. Create a ConfigMap or use an existing one. Multiple Pods can reference the same ConfigMap.

2. Modify your Pod definition to add a volume under `.spec.volumes[]`. Name the volume anything, and have a `.spec.volumes[].configMap.name` field set to reference your ConfigMap object.
3. Add a `.spec.containers[].volumeMounts[]` to each container that needs the ConfigMap. Specify `.spec.containers[].volumeMounts[].readOnly = true` and `.spec.containers[].volumeMounts[].mountPath` to an unused directory name where you would like the ConfigMap to appear.
4. Modify your image or command line so that the program looks for files in that directory. Each key in the ConfigMap `data` map becomes the filename under `mountPath`.

This is an example of a Pod that mounts a ConfigMap in a volume:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
      readOnly: true
  volumes:
  - name: foo
    configMap:
      name: myconfigmap
```

Each ConfigMap you want to use needs to be referred to in `.spec.volumes`.

If there are multiple containers in the Pod, then each container needs its own `volumeMounts` block, but only one `.spec.volumes` is needed per ConfigMap.

Mounted ConfigMaps are updated automatically

When a ConfigMap currently consumed in a volume is updated, projected keys are eventually updated as well. The kubelet checks whether the mounted ConfigMap is fresh on every periodic sync. However, the kubelet uses its local cache for getting the current value of the ConfigMap. The type of the cache is configurable using the `configMapAndSecretChangeDetectionStrategy` field in the KubeletConfiguration struct. A ConfigMap can be either propagated by watch (default), ttl-based, or by redirecting all requests directly to the API server. As a result, the total delay from the moment when the ConfigMap is updated to the moment when new keys are projected to the Pod can be as long as the kubelet sync period + cache propagation delay, where the cache propagation delay depends on the chosen cache type (it equals to watch propagation delay, ttl of cache, or zero correspondingly).

ConfigMaps consumed as environment variables are not updated automatically and require a pod restart.

> **Note:**
> A container using a ConfigMap as a subPath volume mount will not receive ConfigMap updates.

Using Configmaps as environment variables

To use a Configmap in an environment variable in a Pod:

1. For each container in your Pod specification, add an environment variable for each Configmap key that you want to use to the `env[].valueFrom.configMapKeyRef` field.
2. Modify your image and/or command line so that the program looks for values in the specified environment variables.

This is an example of defining a ConfigMap as a pod environment variable:

The following ConfigMap (myconfigmap.yaml) stores two properties: username and access_level:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: myconfigmap
data:
  username: k8s-admin
  access_level: "1"
```

The following command will create the ConfigMap object:

```
kubectl apply -f myconfigmap.yaml
```

The following Pod consumes the content of the ConfigMap as environment variables:

configmap/env-configmap.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: env-configmap
spec:
  containers:
    - name: app
      command: ["/bin/sh", "-c", "printenv"]
      image: busybox:latest
      envFrom:
        - configMapRef:
            name: myconfigmap
```

The `envFrom` field instructs Kubernetes to create environment variables from the sources nested within it. The inner `configMapRef` refers to a ConfigMap by its name and selects all its key-value pairs. Add the Pod to your cluster, then retrieve its logs to see the output from the printenv command. This should confirm that the two key-value pairs from the ConfigMap have been set as environment variables:

```
kubectl apply -f env-configmap.yaml
```

```
kubectl logs pod/env-configmap
```

The output is similar to this:

```
...
username: "k8s-admin"
access_level: "1"
...
```

Sometimes a Pod won't require access to all the values in a ConfigMap. For example, you could have another Pod which only uses the username value from the ConfigMap. For this use case, you can use the `env.valueFrom` syntax instead, which lets you select individual keys in a ConfigMap. The name of the environment variable can also be different from the key within the ConfigMap. For example:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: env-configmap
spec:
  containers:
  - name: envars-test-container
    image: nginx
    env:
    - name: CONFIGMAP_USERNAME
      valueFrom:
        configMapKeyRef:
          name: myconfigmap
          key: username
```

In the Pod created from this manifest, you will see that the environment variable `CONFIGMAP_USERNAME` is set to the value of the `username` value from the ConfigMap. Other keys from the ConfigMap data are not copied into the environment.

It's important to note that the range of characters allowed for environment variable names in pods is restricted. If any keys do not meet the rules, those keys are not made available to your container, though the Pod is allowed to start.

Immutable ConfigMaps

ⓘ **FEATURE STATE:** Kubernetes v1.21 [stable]

The Kubernetes feature *Immutable Secrets and ConfigMaps* provides an option to set individual Secrets and ConfigMaps as immutable. For clusters that extensively use ConfigMaps (at least tens of thousands of unique ConfigMap to Pod mounts), preventing changes to their data has the following advantages:

- protects you from accidental (or unwanted) updates that could cause applications outages
- improves performance of your cluster by significantly reducing load on kube-apiserver, by closing watches for ConfigMaps marked as immutable.

You can create an immutable ConfigMap by setting the `immutable` field to `true`. For example:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  ...
data:
  ...
immutable: true
```

Once a ConfigMap is marked as immutable, it is *not* possible to revert this change nor to mutate the contents of the `data` or the `binaryData` field. You can only delete and recreate the ConfigMap. Because existing Pods maintain a mount point to the deleted ConfigMap, it is recommended to recreate these pods.

What's next

- Read about Secrets.
- Read Configure a Pod to Use a ConfigMap.
- Read about changing a ConfigMap (or any other Kubernetes object)
- Read The Twelve-Factor App to understand the motivation for separating code from configuration.

Feedback

Was this page helpful?

Yes No

Last modified November 21, 2025 at 2:18 PM PST: Fix formatting of kubectl logs command (69fb346f79)