Kubernetes Learning Series

ConfigMap Implementation

Volume Mounts & Live Updates

Lighting up ConfigMaps: Live Updates Without Restarts!

Table of Contents

Introduction

In real-world Kubernetes environments, application configurations often change like switching log levels or maintenance modes. Instead of rebuilding and redeploying applications every time, we can use ConfigMaps to manage these changes easily and dynamically.

In this exercise, we'll learn how to dynamically update application configurations in Kubernetes using **ConfigMaps mounted as volumes**. This method allows us to change configuration values without rebuilding or restarting the Pod.

How ConfigMap Updates Work

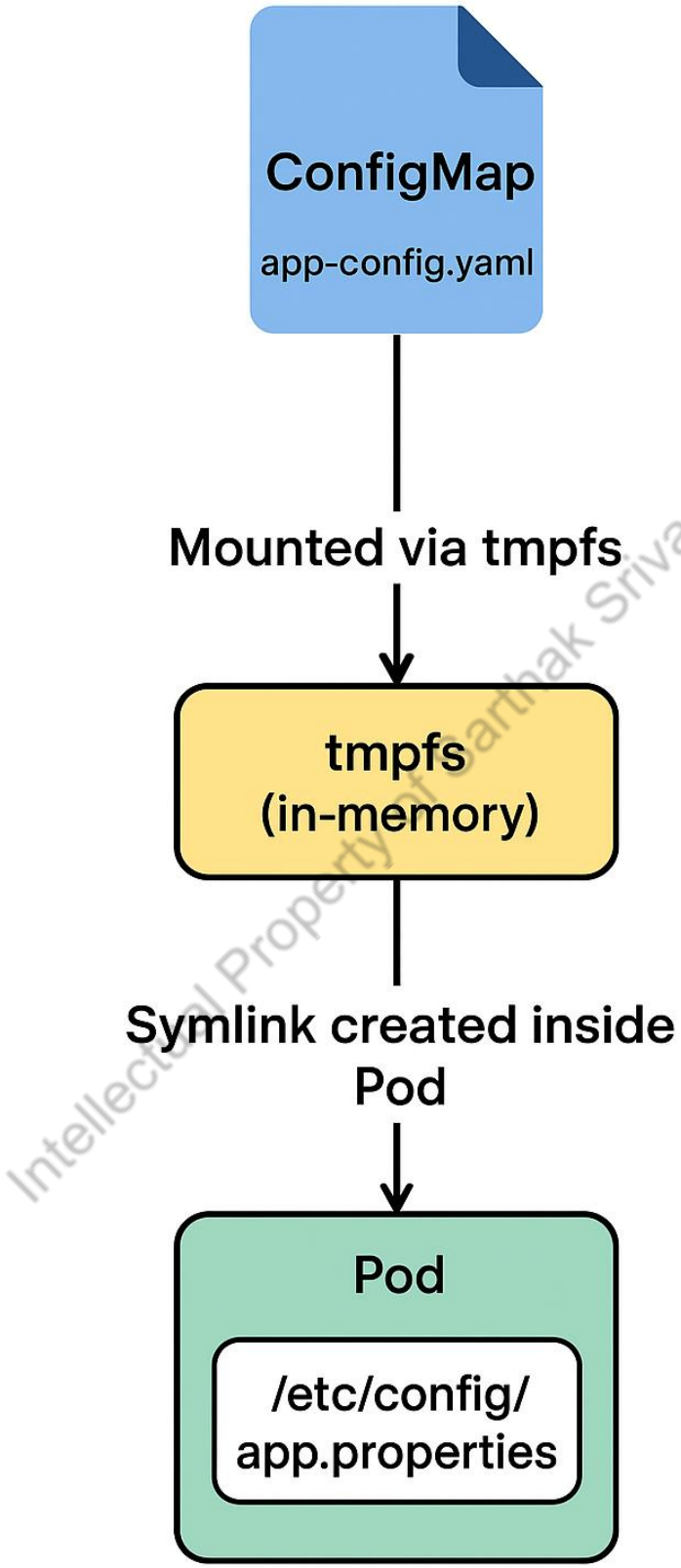
When you use a **ConfigMap** in Kubernetes, you can make your application read configuration values (like mode, log level, etc.) from it.
One common and efficient way to do this in Production is by **mounting the ConfigMap as a volume** inside your Pod.

What Actually Happens Behind the Scenes

1. ### Temporary File System (tmpfs)
   Kubernetes stores ConfigMap data in a special in-memory storage area called **tmpfs**.
   This is a temporary file system that exists only while the Pod is running fast and doesn't write to disk.
   Using tmpfs ensures that ConfigMap data is stored in memory, which makes updates lightweight and fast without writing to the node's disk.
2. ### Symlinks (Shortcut Files)
   Inside your Pod, Kubernetes doesn't copy the actual ConfigMap files directly.
   Instead, it creates **symlinks,** which are like *shortcuts* pointing to the real files stored in tmpfs.
   This design helps Kubernetes refresh configurations without restarting or rewriting containers, the symlink simply points to the new version of the ConfigMap data.

Flow Diagram

The following diagram shows how Kubernetes keeps ConfigMap data in sync inside running Pods, smart, lightweight, and automatic.



⚙️ How Updates Work

- When you **edit the ConfigMap**, Kubernetes updates the files stored in tmpfs.
- Because the Pod's files are symlinks (shortcuts) to that tmpfs data, the change **automatically appears** inside the Pod.
- This update usually happens within a few seconds and no Pod restart is needed.

This behavior allows you to make configuration changes in production safely, without downtime or redeploying your services.

🚫 Environment Variables Don't Auto-Update

If you use a ConfigMap to set **environment variables**, those values are loaded **only when the Pod starts**.
So if you edit the ConfigMap later, the environment variables **won't change automatically**, you'll need to **restart the Pod**.

This distinction is important because many beginners assume both methods behave the same way, knowing this helps you choose the right approach based on whether live updates are needed.

⚠️ One More Thing: App Reloading

Even though the file inside the Pod updates automatically, some applications **cache** their configuration when they start.
In that case, the app might not notice the new changes until you restart or reload it.

This means if your application doesn't automatically reload files, you may need to build a reload mechanism or restart the app to pick up new configurations.

Step 1: Create the namespace

A namespace helps keep your demo resources separate from other Kubernetes workloads. It's like a folder that organizes your deployments and ConfigMaps.

Command:

kubectl create namespace config-demo

```
controlplane ~ ➜ kubectl create namespace config-demo
```

Expected Output:

```
namespace config-demo created
```

```
mkdir ~/k8s-config-demo && cd ~/k8s-config-demo
namespace/config-demo created
```

Step 2: Create Deployment

The Deployment ensures your Pod runs continuously and automatically restarts if it fails.

We'll mount our ConfigMap here so that configuration files appear under /etc/config inside the Pod.

```
controlplane ~ ✖ cat >deploy.yaml <<EOF
> apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-deployment
  namespace: config-demo
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: app-container
          image: nginx:latest   # Replace with your app image
          volumeMounts:
            - name: config-volume
              mountPath: /etc/config
              readOnly: true
      volumes:
        - name: config-volume
          configMap:
            name: app-config
EOF
```

Created a Deployment (app-deployment) mounting the ConfigMap as a volume at /etc/config.

Step 3: Create the ConfigMap

A ConfigMap holds configuration data separately from your container image, allowing you to change it without touching the application code.

```
controlplane ~ ➜ cat >configm.yaml <<EOF
> apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
  namespace: config-demo
data:
  app.properties: |
    APP_MODE=maintenance
    LOG_LEVEL=debug
EOF
```

Created a ConfigMap (app-config) with app.properties containing.

Step 4: Apply the Deployment and ConfigMap

Applying both ensures that the Deployment (Pod) and the ConfigMap exist together, the Pod can then successfully mount the ConfigMap as a volume.

Command:

kubectl apply -f deploy.yaml

```
controlplane ~ ➜ kubectl apply -f deploy.yaml
```

kubectl apply -f configm.yaml

```
controlplane ~ ➜ kubectl apply -f configm.yaml
```

Expected Output:

```
deployment/app-deployment created

configmap/app-config created
```


Step 5: Check the Pod

Let's verify if our Pod is up and running in the config-demo namespace.

Command:

kubectl get pods -n config-demo

```
controlplane ~ ➜ kubectl get pods -n config-demo
```

Expected Output:

```
NAME                              READY   STATUS    RESTARTS   AGE
app-deployment-7fb7d7c97c-9b94n   1/1     Running   0          1m
```

```
NAME                              READY   STATUS    RESTARTS   AGE
app-deployment-7fb7d7c97c-9b94n   1/1     Running   0          5m24s
```

Step 6: Verify the mounted ConfigMap

Let's look inside the Pod to confirm that our ConfigMap file has been successfully mounted at /etc/config.

Command:

kubectl exec -it <pod-name> -n config-demo -- cat /etc/config/app.properties

```
controlplane ~ ➜ kubectl exec -it app-deployment-7fb7d7c97c-9b94n -n config-
demo -- cat /etc/config/app.properties
```

Expected Output:

```
APP_MODE=maintenance
LOG_LEVEL=debug
```


```
APP_MODE=maintenance
LOG_LEVEL=debug
```

Editing ConfigMap to check if the changes made in the ConfigMap will reflect in real-time without restarting the deployment.

Now, let's test whether updates made to the ConfigMap appear inside the running Pod without restarting it.

Step 1: Check the ConfigMap exists

Command:

kubectl get cm -n config-demo

```
controlplane ~ ➜ kubectl get cm -n config-demo
```

Expected Output:

```
NAME               DATA   AGE
app-config         1      6m47s
kube-root-ca.crt   1      7m12s
```

Step 2: Edit the ConfigMap

Command:

kubectl edit cm app-config -n config-demo

```
controlplane ~ ➜ kubectl edit cm app-config -n config-demo
```

This opens the ConfigMap in your default editor. You will see something like the following -

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
  namespace: config-demo
data:
  app.properties: |
    APP_MODE=maintenance
    LOG_LEVEL=debug
```

Step 3: Make changes

For example, change APP_MODE to Production:

```
data:
  app.properties: |
    APP_MODE=production
    LOG_LEVEL=debug
```

Once you have made the change, you will have to save and exit the editor (:wq in vi).

Expected Output:

```
configmap/app-config edited
```

✅ The ConfigMap is now updated in Kubernetes.

Step 4: Check the changes

This step demonstrates auto-**update behavior**. You will see both before and after clearly.

Before Editing:

APP_MODE=maintenance

LOG_LEVEL=debug

After Editing:

APP_MODE=Production

LOG_LEVEL=debug

Command:

kubectl get cm app-config -n config-demo -o yaml

```
controlplane ~ ➜ kubectl get cm app-config -n config-demo -o yaml
apiVersion: v1
data:
  app.properties: |
    APP_MODE=Production
    LOG_LEVEL=debug
kind: ConfigMap
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"v1","data":{"app.properties":"APP_MODE=maintenance\nLOG_
LEVEL=debug\n"},"kind":"ConfigMap","metadata":{"annotations":{},"name":"app-c
onfig","namespace":"config-demo"}}
  creationTimestamp: "2025-10-09T11:19:41Z"
  name: app-config
  namespace: config-demo
  resourceVersion: "4237"
  uid: cbf4ad89-3f29-4961-ac1b-7586c69bd9e2
```

Expected Output:

```
data:
  app.properties: |
    APP_MODE=Production
    LOG_LEVEL=debug
```

You should see the updated values in app.properties.

maintenance -> **Production**

Step 5: Verify inside the Pod

Let's check inside the running Pod to see if the new ConfigMap values have appeared.

If your Pod is mounting the ConfigMap as a **volume**, the changes will appear automatically in the mounted file:

Command:

kubectl get pods -n config-demo

```
controlplane ~ ➜ kubectl get pods -n config-demo
```

Expected Output:

```
NAME                                READY   STATUS    RESTARTS   AGE
app-deployment-7fb7d7c97c-9b94n   1/1     Running   0          8m16s
```

Note: Copy the **Pod Name** to paste it in the next command.

Command:

kubectl exec -it app-deployment-7fb7d7c97c-9b94n -n config-demo -- cat /etc/config/app.properties

```
controlplane ~ ➜ kubectl exec -it app-deployment-7fb7d7c97c-9b94n -n config-
demo -- cat /etc/config/app.properties
```

Expected Output:

```
APP_MODE=Production
LOG_LEVEL=debug
```

No Pod restart was needed; the volume mount automatically reflected the changes.

Trying to update the values inside the ConfigMap once again so that the learners can clearly see the live update happening

We'll edit the ConfigMap once again to demonstrate how changes appear live, and also understand the slight delay that may occur before the update reflects inside the Pod.

Step 1: Edit the ConfigMap

Command:

kubectl edit cm app-config -n config-demo

```
controlplane ~ ➜ kubectl edit cm app-config -n config-demo
configmap/app-config edited
```

This opens the ConfigMap in your editor.

For demonstration, change APP_MODE again, e.g.:

```
data:
  app.properties: |
    APP_MODE=Staging
    LOG_LEVEL=debug
```

Save and exit.

Step 2: Verify the ConfigMap updated

Command:

kubectl get cm app-config -n config-demo -o yaml

```
controlplane ~ ➜ kubectl get cm app-config -n config-demo -o yaml
```

Expected Output:

```
apiVersion: v1
data:
  app.properties: |
    APP_MODE=Staging
    LOG_LEVEL=info
kind: ConfigMap
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"v1","data":{"app.properties":"APP_MODE=maintenance\nLOG_
LEVEL=debug\n"},"kind":"ConfigMap","metadata":{"annotations":{},"name":"app-c
onfig","namespace":"config-demo"}}
  creationTimestamp: "2025-10-09T11:19:41Z"
  name: app-config
  namespace: config-demo
  resourceVersion: "4598"
  uid: cbf4ad89-3f29-4961-ac1b-7586c69bd9e2
```

Step 3: Check inside the running Pod

Command:

kubectl exec -it app-deployment-7fb7d7c97c-9b04n -n config-demo –cat /etc/config/app.properties

First attempt:

```
controlplane ~ ➜ kubectl exec -it app-deployment-7fb7d7c97c-9b94n -n config-
demo -- cat /etc/config/app.properties
```

First Attempt:
APP_MODE=Production
LOG_LEVEL=debug

Second Attempt:
APP_MODE=Staging
LOG_LEVEL=debug

Output:

```
APP_MODE=Production
LOG_LEVEL=debug
```

Second attempt:

```
controlplane ~ ➜ kubectl exec -it app-deployment-7fb7d7c97c-9b94n -n config-
demo -- cat /etc/config/app.properties
```

Output:

```
APP_MODE=Production
LOG_LEVEL=debug
```

Why didn't the change in APP_MODE from "Production" to "Staging" appear inside the Pod after hitting the command to verify the changes inside the pod twice?

Explanation:

Propagation is not instant:

- The kubelet checks for ConfigMap changes **periodically**, usually every few seconds.
- That's why our above 2 cat commands still showed the old value (Production).
- On the 3rd attempt, the symlink had been updated and your Pod saw the new value (Staging).
- **Pod sees updated content immediately once kubelet syncs it**, no restart needed. The slight delay is normal and expected in all Kubernetes clusters.

You can find the updated value (Staging) in the following screenshot:

Third attempt Output:

```
controlplane ~ ➜ kubectl exec -it app-deployment-7fb7d7c97c-9b94n -n config-
demo -- cat /etc/config/app.properties
APP_MODE=Staging
LOG_LEVEL=info
```

Key Takeaway:

- "ConfigMaps as volumes allow live updates."
- "There can be a short propagation delay of a few seconds, depending on kubelet sync."
- "No Pod restart is needed, eventually the file updates automatically inside the running Pod."

This small delay shows that the kubelet periodically checks for ConfigMap changes.
It's a great example of how Kubernetes handles consistency efficiently without constantly reloading resources.

Tip for smoother transition:

In production systems, it's good practice to design your application to detect file changes or reload configuration dynamically, ensuring it always uses the latest settings from the ConfigMap.

Wait **5–10 seconds** after editing the ConfigMap before verifying the file inside the Pod

---

🧩 Common Issues

Issue	Possible Cause	Fix
ConfigMap changes not visible	kubelet hasn't synced yet	Wait a few seconds and check again
Pod shows old values after several minutes	App caches config	Restart the Pod or reload the app
Error "namespace not found"	Namespace doesn't exist	Run kubectl create namespace config-demo

---

Summary

🧠 **What We Learned**

- How to mount a ConfigMap as a volume inside a Pod.
- How ConfigMap updates propagate automatically without restarting the Pod.
- Why there's a slight delay in update reflection.
- The difference between **ConfigMap as Volume** (auto-updates) and **ConfigMap as Env Vars** (manual restart).

By mastering ConfigMap volumes, you can manage configuration changes seamlessly, a critical skill for cloud engineers who want to maintain high availability, reduce deployment downtime, and adopt DevOps best practices.
