🌟 What Are Configmap and Secret in Kubernetes?

| **Feature** | **ConfigMap** | **Secret** |
| --- | --- | --- |

|  |  |  |
| --- | --- | --- |
| **Purpose** | Store **non-sensitive** config data | Store **sensitive** data (like passwords) |

|  |  |  |
| --- | --- | --- |
| **Data Type** | Plaintext (key-value pairs) | Base64-encoded (but still not encrypted) |

|  |  |  |
| --- | --- | --- |
| **Use Cases** | App names, URLs, ports | DB passwords, API keys, TLS certs |

**❓ Why Do We Need Them?**

**Without ConfigMap/Secret:**

* Values like DB URLs and credentials are **hardcoded inside your YAML files** or even your application.
* This creates **security issues** and **inflexibility**—you must re-deploy the app for every config change.

**With ConfigMap/Secret:**

* It improves **security** by separating secrets from your codebase.
* You **externalize configuration** from your application.
* You can update configuration **without modifying the app**.
* It allows **reusability** across different deployments or environments (dev, staging, prod).

**1. What is a ConfigMap in Kubernetes?**

A **ConfigMap** is a Kubernetes object used to **store non-sensitive configuration data** as key-value pairs.  
These can be used by pods to configure apps **without hardcoding values** into container images or YAML specs.

**🚨 Why NOT Hardcode Config in YAML?**

| **⚠️ Problem** | **🛠 Solution via ConfigMap** |
| --- | --- |
| Need to change value often (port, URL) | Change just the ConfigMap |
| Use same config in many apps | Share one ConfigMap |
| Different configs per environment | Use different ConfigMaps (dev, staging, prod) |
| Want config managed by DevOps, not devs | ConfigMap separates app code from config |

**🛠️ 5. Create ConfigMap YAML (Step-by-Step)**

**✅ Step 1: Create spring-config.yaml**

apiVersion: v1

kind: ConfigMap

metadata:

name: spring-config

namespace: default # Change if you're using a different namespace

data:

SPRING\_APPLICATION\_NAME: creditcard\_service

SERVER\_PORT: '9090'

SPRING\_DATASOURCE\_URL: jdbc:postgresql://20.245.16.15:5432/springbootdb

**Step 2: Apply ConfigMap to Cluster**

kubectl apply -f spring-config.yaml

kubectl get configmap spring-config -o yaml

**📦 6. Use ConfigMap in Deployment**

Update your deployment to **read these values from the ConfigMap**.

env:

- name: SPRING\_APPLICATION\_NAME

valueFrom:

configMapKeyRef:

name: spring-config

key: SPRING\_APPLICATION\_NAME

- name: SERVER\_PORT

valueFrom:

configMapKeyRef:

name: spring-config

key: SERVER\_PORT

- name: SPRING\_DATASOURCE\_URL

valueFrom:

configMapKeyRef:

name: spring-config

key: SPRING\_DATASOURCE\_URL

Let's dive into **Feature Flags** with a **real-time enterprise use case**, showing how it's implemented in Kubernetes using **ConfigMap**.

**What Are Feature Flags?**

**Feature flags (aka feature toggles)** let you enable or disable specific functionality in your application **at runtime** without deploying new code.

It’s like a remote control for your app features.

**Business Scenario:**

Your company is launching a new **OTP-based verification** for credit card payments. But:

* You want to **gradually roll it out** only to users in the Asia region.
* You want to **quickly disable it** in case it causes issues.
* You don’t want to redeploy the service every time.

**✅ Solution: Use a ConfigMap as a Feature Toggle**

**Step 1: Create a ConfigMap with Feature Flags**

apiVersion: v1

kind: ConfigMap

metadata:

name: feature-flags

namespace: default

data:

ENABLE\_OTP\_VERIFICATION: "true"

ENABLE\_PROMO\_DISCOUNTS: "false"

Step 2: Reference It in Deployment (as env vars)

env:

- name: ENABLE\_OTP\_VERIFICATION

valueFrom:

configMapKeyRef:

name: feature-flags

key: ENABLE\_OTP\_VERIFICATION

- name: ENABLE\_PROMO\_DISCOUNTS

valueFrom:

configMapKeyRef:

name: feature-flags

key: ENABLE\_PROMO\_DISCOUNTS

Step 3: Use It in Your Spring Boot Code

@Value("${ENABLE\_OTP\_VERIFICATION:false}")

private boolean otpVerificationEnabled;

public void processPayment(PaymentRequest request) {

if (otpVerificationEnabled) {

// Call OTP service before processing

otpService.sendOtp(request.getUserPhone());

}

// Continue with payment logic

}

**Step 4: Toggle the Flag Without Code Change**

Want to turn off OTP temporarily?

**kubectl edit configmap feature-flags**

# Change ENABLE\_OTP\_VERIFICATION: "false"

Then:

**kubectl rollout restart deployment creditcard-service**

The app restarts with new env value → OTP verification is disabled.

A screenshot of a phone

AI-generated content may be incorrect.

Geo Targeting Features

You run a global fintech platform with customers across:

* **Asia**
* **Europe**
* **North America**

You’ve developed a **new cashback feature** and want to:

* Enable it **only for users in Asia** (as a pilot).
* Keep it **disabled elsewhere**.
* Control the rollout via **ConfigMap** without code changes or redeployment.

**Client Request → App Checks Region → App Reads Feature Flag from ENV (via ConfigMap) → Feature Enabled/Disabled**

**✅ Step 1: Create a ConfigMap with Region Flags**

**# geo-feature-flags.yaml**

**apiVersion: v1**

**kind: ConfigMap**

**metadata:**

**name: geo-feature-flags**

**namespace: default**

**data:**

**CASHBACK\_FEATURE\_ASIA: "true"**

**CASHBACK\_FEATURE\_EUROPE: "false"**

**CASHBACK\_FEATURE\_NORTHAMERICA: "false"**