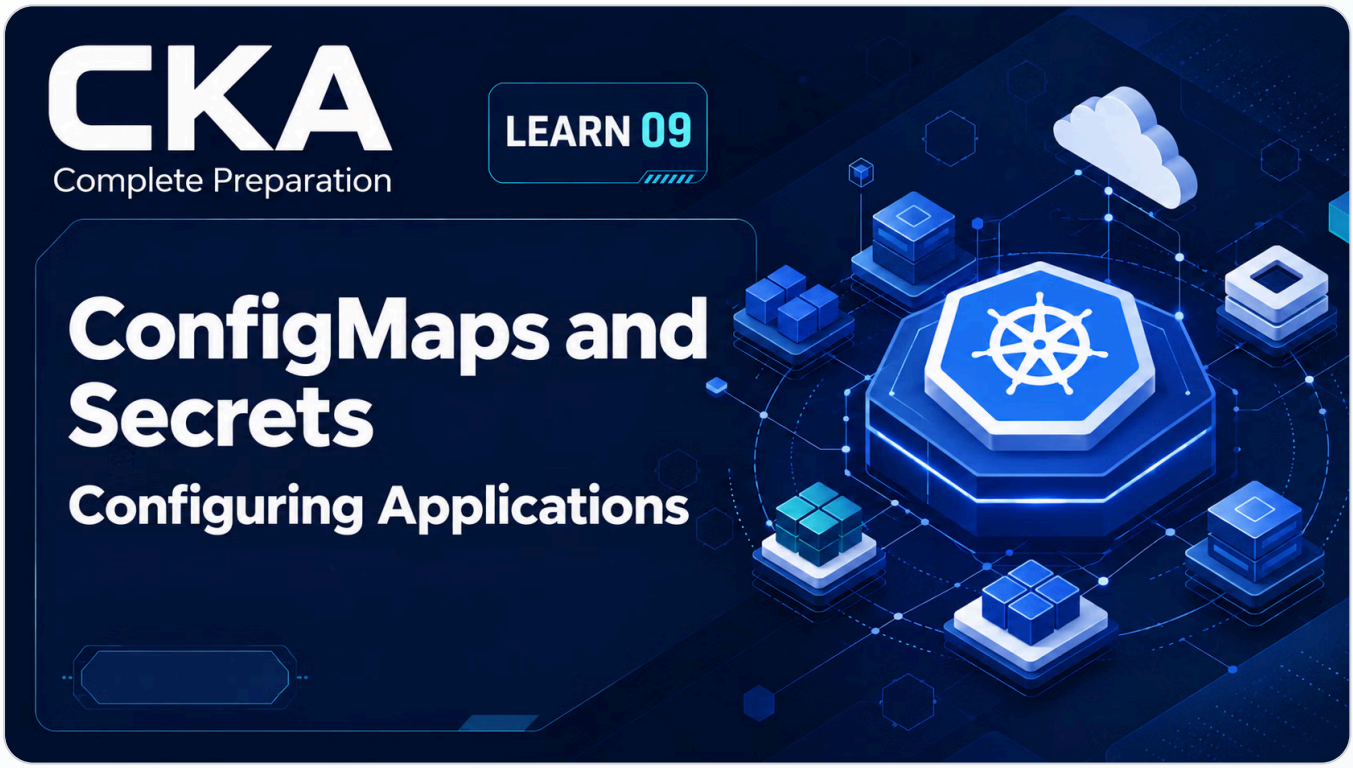
Omar Latrach

TECHNICAL ARTICLE

PUBLISHED · SEPTEMBER 19, 2026

ConfigMaps and Secrets: Configuring Applications

Build a durable mental model for Kubernetes application configuration: create ConfigMaps and Secrets, deliver selected values as environment variables or files, and reason correctly about updates and failures.



This is Learn post 09 of the 54-post Certified Kubernetes Administrator (CKA) preparation path. Earlier posts built Pods and controllers from Pod templates. This lesson separates the application image from the values that vary between environments, deployments, and credentials.

A container image should describe what the application is. A ConfigMap or Secret describes what that instance should know. Kubernetes stores that data as separate API objects, while the Pod specification declares how containers receive it.

What you'll learn

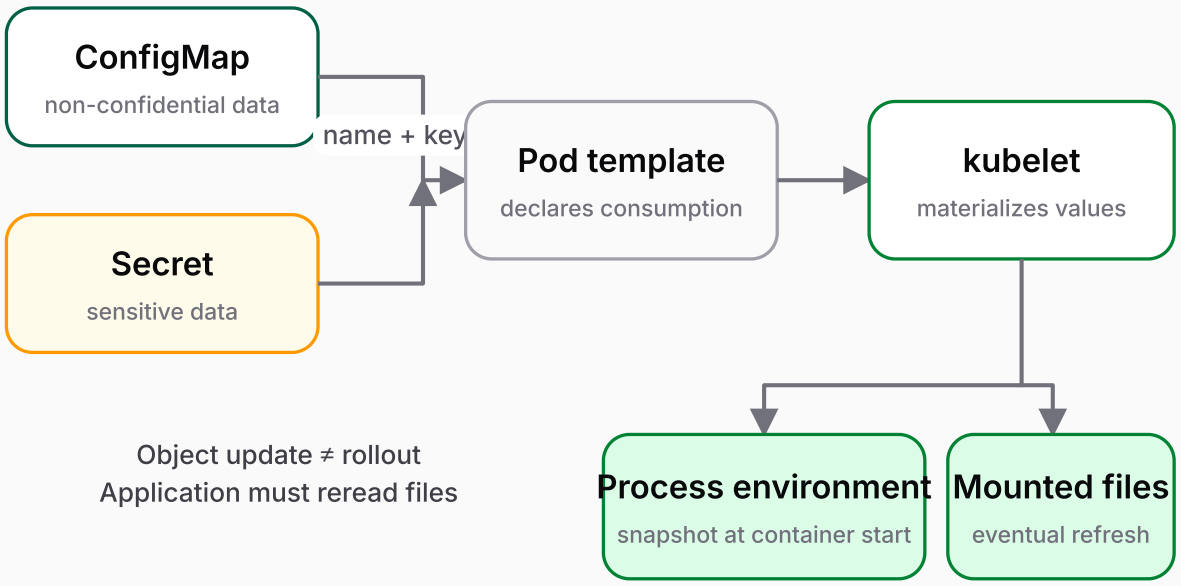
- Choose ConfigMaps for non-confidential configuration and Secrets for sensitive values.
- Create both objects imperatively and declaratively, then inspect their real API representation.
- Deliver individual keys through environment variables and selected keys through read-only volume files.
- Predict what changes inside a running Pod when a ConfigMap or Secret is updated.
- Use Pod status and events to diagnose missing objects, missing keys, and delivery failures.

The mental model: store, reference, materialize, read

> ConfigMaps and Secrets hold data. Pod templates hold references. The kubelet materializes values. The application must read them.

Creating a ConfigMap does not configure every Pod, and changing a Secret does not edit a Deployment. A workload consumes data only when its Pod template names the object and chooses a delivery mechanism. This is a reference relationship, not controller ownership.

When a Pod is assigned to a node, the kubelet resolves its required references. It can place a value in the new container's environment or project keys as files in a Pod volume. Those paths have different update behavior, so the delivery choice is part of the application's runtime design.



Configuration travels through references, not ownership

Environment variables are fixed for a running container. Projected volume files can refresh eventually, but the application still has to reread them. Updating either source object does not automatically roll out a Deployment.

Because the Pod specification records only a name and optional key, the ConfigMap or Secret remains independently manageable and reusable by multiple Pods in the same namespace.

ConfigMap and Secret solve different data-classification problems

A ConfigMap holds non-confidential strings or binary data: feature settings, service endpoints, logging levels, and configuration-file content. It has data for UTF-8 strings and binaryData for base64-encoded bytes. Unlike a Deployment, it does not wrap desired state in a spec field.

A Secret has a similar key-value shape but communicates that the values are sensitive. The common Opaque type accepts application-defined keys. The data field stores base64-encoded bytes; stringData is a write convenience that accepts clear text and is merged into data by the API server.

> Base64 is an encoding, not encryption. A Secret changes handling intent and unlocks Secret-specific controls; it does not make readable data cryptographically safe by itself.

Both resources are namespaced. Native Pod references resolve an object in the Pod's namespace, so a Pod in cka-config cannot satisfy its reference from an identically named object in another namespace.

Generate manifests without creating objects

Imperative generators are useful when literals or local files already exist. The client-side dry run validates command construction and prints a reusable manifest; it does not contact the API server to create the object.

```bash
kubectl create namespace cka-config

kubectl create configmap app-settings \
  --namespace=cka-config \
  --from-literal=APP_MODE=production \
  --from-file=app.properties \
  --dry-run=client -o yaml

kubectl create secret generic app-credentials \
  --namespace=cka-config \
  --from-literal=username=report-reader \
  --from-literal=token=demo-token-not-for-production \
  --dry-run=client -o yaml
```

With --from-file=app.properties, the filename becomes the key and the file contents become the value. The literals here are harmless demonstration values. Avoid putting real credentials on a command line where shell history or process inspection could

expose them.

Build one reproducible configuration flow

The following objects keep public configuration separate from credentials. stringData keeps this learning manifest readable; do not commit a real secret value to source control merely because the field accepts clear text.

config-data.yaml · yaml

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-settings
  namespace: cka-config
data:
  APP_MODE: production
  app.properties: |
    color=blue
    feature.enabled=true
---
apiVersion: v1
kind: Secret
metadata:
  name: app-credentials
  namespace: cka-config
type: Opaque
stringData:
  username: report-reader
  token: demo-token-not-for-production
```

The ConfigMap contains one scalar setting and one file-shaped value. Kubernetes does not distinguish those meanings; the Pod decides whether a key becomes an environment variable or a file. The Secret uses Opaque because these keys are application-specific.

Reference precise values from the Pod template

config-reader.yaml · yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: config-reader
  namespace: cka-config
spec:
  replicas: 1
  selector:
    matchLabels:
      app: config-reader
  template:
    metadata:
      labels:
        app: config-reader
    spec:
      containers:
        - name: reader
          image: busybox:1.36.1
          command: ["sh", "-c"]
          args:
            - while true; do sleep 3600; done
          env:
            - name: APP_MODE
              valueFrom:
                configMapKeyRef:
                  name: app-settings
                  key: APP_MODE
            - name: APP_USER
              valueFrom:
                secretKeyRef:
                  name: app-credentials
                  key: username
          volumeMounts:
            - name: settings
              mountPath: /etc/demo/config
              readOnly: true
            - name: credentials
              mountPath: /etc/demo/credentials
              readOnly: true
      volumes:
        - name: settings
          configMap:
            name: app-settings
            items:
              - key: app.properties
                path: app.properties
        - name: credentials
          secret:
            secretName: app-credentials
            defaultMode: 0400
            items:
              - key: token
                path: token
```

The two env entries select one key each and may rename it for the process. The Pod-level volumes select file-shaped keys; each container that needs them has its own volumeMounts entries. items limits projection to the listed keys and can rename each destination path. The Secret volume applies read-only owner permissions with YAML's octal 0400 value.

The Pod template stores references, not copies of the current values. The kubelet resolves those references for each Pod. This also means a missing required object or key can block container startup even though the Deployment and Pod objects themselves were accepted by the API server.

Apply, observe, and read without exposing the token

```bash
kubectl apply -f config-data.yaml
kubectl apply -f config-reader.yaml
kubectl rollout status deployment/config-reader -n cka-config

kubectl get configmap app-settings -n cka-config
kubectl get secret app-credentials -n cka-config
```

```
Representative output; age varies · plaintext

NAME           DATA   AGE
app-settings   2      12s

NAME              TYPE     DATA   AGE
app-credentials   Opaque   2      12s
```

The DATA column counts keys, not bytes and not consumers. kubectl intentionally avoids printing Secret values in this table. Verify delivery from inside the application container, but test only what is needed; the command below confirms that the token file is non-empty without displaying it.

```bash
kubectl exec -n cka-config deployment/config-reader -- sh -c '
  printf "APP_MODE=%s APP_USER=%s\n" "$APP_MODE" "$APP_USER"
  cat /etc/demo/config/app.properties
  test -s /etc/demo/credentials/token && echo "token file is present"
'
```

```plaintext
APP_MODE=production APP_USER=report-reader
color=blue
feature.enabled=true
token file is present
```

One ConfigMap supplied both an environment value and a mounted file. One Secret supplied a selected environment value and a mounted credential file. The container image contains none of those environment-specific values.

Choose the delivery mechanism by how the application reads

- Use env.valueFrom with configMapKeyRef or secretKeyRef when the process expects one named environment variable. It makes the dependency and rename explicit, as APP_USER demonstrates.
- Use envFrom with configMapRef or secretRef when every compatible key should become an environment variable. The keys become variable names unless a prefix is configured, so bulk import creates a wider and less explicit contract.
- Use a configMap or secret volume when the application expects files, needs multi-line content, or can reread changed files. One Pod volume may be mounted by multiple containers, but every consumer needs its own volumeMounts entry.
- Use items when only selected keys should appear, when destination filenames should differ from source keys, or when per-item file modes matter. If items is omitted, every key is projected as a file named after that key.

Environment delivery is convenient for startup configuration but fixed for that container process. Volume delivery supports eventual file refresh, but only helps if the application watches or periodically rereads the files. Kubernetes delivers bytes; it does not force an application to reload them.

Official ConfigMap-to-Pod examples show the supported environment, command, and volume patterns.

Updates follow the delivery path

Update both ConfigMap values while the original Pod keeps running. The patch changes the source API object; it does not change the Deployment's Pod template and therefore does not create a new ReplicaSet or begin a rollout.

```bash
kubectl patch configmap app-settings -n cka-config --type=merge \
  -p '{"data":{"APP_MODE":"maintenance","app.properties":"color=green\nfeature.enabled=true\n"}}'

kubectl get configmap app-settings -n cka-config \
  -o jsonpath='{.data.APP_MODE}{"\n"}'
```

The API now reports maintenance immediately. After the kubelet's periodic synchronization and cache propagation, the projected app.properties file is replaced with the green version. That delivery is eventually consistent, not instantaneous.

```bash
# Repeat after a short interval if the projected file still has the old value.
kubectl exec -n cka-config deployment/config-reader -- sh -c '
  printf "environment: %s\n" "$APP_MODE"
  printf "file: "
  head -n 1 /etc/demo/config/app.properties
'
```

Representative result after volume propagation · plaintext

```
environment: production
file: color=green
```

The difference is causal. The running process received APP_MODE when its container started, so that environment remains production. The mounted file is managed by the kubelet, so its content can be refreshed in place. If the application read the file only during startup, its behavior may still remain unchanged despite the new bytes on disk.

Restart Pods when the consumer needs a fresh startup snapshot

bash

```
kubectl rollout restart deployment/config-reader -n cka-config
kubectl rollout status deployment/config-reader -n cka-config
kubectl exec -n cka-config deployment/config-reader -- printenv APP_MODE
```

New container value · plaintext

```
maintenance
```

rollout restart changes a Pod-template annotation, so the Deployment creates replacement Pods. Each new container resolves the current ConfigMap and receives maintenance. The same distinction applies to Secret-backed environment variables and Secret volume files.

A ConfigMap or Secret mounted through volumeMounts.subPath does not receive these automatic file updates. A normal directory mount is required for kubelet-managed refresh. Even then, avoid building automation around an exact propagation time.

Setting immutable: true protects an object from later data changes and can reduce API-server watch load at large scale. Immutability cannot be turned off and the data cannot be mutated; create a new named object and replace the consuming Pods when configuration must change. For ordinary CKA work, first master the mutable update paths shown above.

The official configuration-update tutorial demonstrates the environment snapshot, projected-volume refresh, and immutable-object patterns.

Missing data blocks materialization, not object creation

The API server can accept a Deployment whose Pod template references an object or key that does not exist. The Deployment controller still creates a Pod. On the node, however, the kubelet cannot build the required environment or volume, so the container does not start successfully.

A missing environment source commonly appears as CreateContainerConfigError. A missing volume source commonly leaves the container waiting while events report a setup or mount failure. The status label is only a summary; the event message normally names the missing ConfigMap, Secret, or key.

```bash
kubectl get pods -n cka-config -l app=config-reader
kubectl describe pod -n cka-config -l app=config-reader
kubectl get events -n cka-config --sort-by=.metadata.creationTimestamp
```

Start with the Pod, then read Events from describe. Check the referenced name, namespace, key spelling, and object contents. Do not repeatedly delete the Pod: its controller will recreate the same broken template until the reference or source data is corrected.

References are required by default. optional: true can allow a Pod to start when an object or key is absent, but that changes the application's contract; use it only when the application genuinely has a safe default.

Handle Secrets according to what they really guarantee

Secret values are readable by clients authorized to get the Secret. Anyone who can read the data field can base64-decode it. Real confidentiality therefore depends on narrow API authorization, encryption at rest for the API data store, secure transport, and careful application behavior—not on the object name or its encoding.

Avoid printing credentials in logs, embedding them in images, placing them in command arguments, or committing clear text and base64 variants to a repository. Restrictive file modes such as 0400 reduce access inside the container filesystem but do not control who may read the Secret through the Kubernetes API.

Built-in Secret types such as TLS and image-pull credentials communicate expected keys and may receive limited format checks. Opaque is appropriate for arbitrary application credentials. This lesson stays with application consumption; API permissions are taught with RBAC in post 22.

Official ConfigMap documentation defines its data model and update behavior. Official Secret documentation covers Secret types, security considerations, and consumption paths.

Clean up and connect the next lessons

```bash
kubectl delete namespace cka-config
```

Deleting the demonstration namespace removes its Deployment, Pods, ConfigMap, and Secret together. The image remains unchanged because configuration was always external to it.

Post 08 supplied the workload-controller choices whose Pod templates can consume this data. Post 10 next adds requests, limits, LimitRanges, and ResourceQuotas. RBAC protection comes in post 22; Kustomize generation in post 24; and broken-configuration diagnosis returns as guided work in Practice post 40.