ConfigMaps and Secrets

ConfigMaps in Kubernetes is a key-value store that allows you to manage application configuration data independently from the application code.

This separation of configuration from the application code adheres to the 12-factor app methodology and ensures flexibility and portability across environments(development,staging,production).

In Kubernetes, ConfigMap is an API object that is mainly used to store non-confidential data. The data that is stored in ConfigMap is stored as key-value pairs.ConfigMaps are configuration files that may be used by pods as command-line arguments, environment variables, or even as configuration files on a disc.

This feature allows us to decouple environment-specific configuration from our container images. After this is done, our applications are easily portable.The thing to be noted here is that ConfigMap does not provide any sort of secrecy or encryption, so it is advised to store non-confidential data only. We can use Secrets to store confidential data.

Working with Kubernetes ConfigMaps allows you to separate configuration details from containerized apps. ConfigMaps are used to hold non-sensitive configuration data that may be consumed as environment variables by containers or mounted as configuration files.

How ConfigMaps Help Manage Application Configurations:

✅ **Decouples Configuration from Code**
ConfigMaps store configuration details outside of the application codebase, allowing you to:

- Update configurations without rebuilding or redeploying the application.
- Manage different configurations for different environments.

✅ **Centralized Management**
All configuration data can be stored in one place (ConfigMaps), making it easier to manage and update application settings.

✅ **Dynamic Updates**
If a ConfigMap is mounted as a volume, changes to the ConfigMap automatically propagate to the pod's filesystem (though not all applications reload these changes dynamically without a restart).

✅ **Portability Across Environments**
The same container image can be used across multiple environments by just updating the ConfigMap with environment-specific configurations.

✅ **Flexible Injection Methods**
ConfigMaps provide multiple ways to inject configurations into your application:

- As environment variables.
- As files mounted into the container (via volumes).
- Through command-line arguments.

✅ **Simplifies Secret Management (with Limits)**
ConfigMaps can hold non-sensitive application parameters, simplifying their management and avoiding exposure in application source code.

we are using the ConfigMap for Mysql Database

Creating a ConfigMap

You can create a ConfigMap using a YAML manifest file like this:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-configmap
  namespace: mynamespace
data:
  MYSQL_DB: "mydb"

~
```

```
root@kiran:~# kubectl apply -f my-configmap.yaml
configmap/my-configmap created
```

```
root@kiran:~# kubectl get cm
NAME               DATA   AGE
kube-root-ca.crt   1      39m
```

Explanation:

As of now, we know about the Manifest YAML file and its attributes

Here the new thing is data:.

data: This is where you define **key-value** pairs for your configuration settings. In this example, we have 1 key-value pairs: **MYSQL_DB**.

Using a ConfigMap in a Pod

Here's a YAML manifest file for creating a Pod that uses the ConfigMap my-configmap:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql-configuration
  namespace: mynamespace
  labels:
    app: mysql
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mysql
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - name: mysql-container
        image: mysql:8
        ports:
        - containerPort: 3306
        env:
        - name: MYSQL_DATABASE
          valueFrom:
            configMapKeyRef:
```

```yaml
            configMapKeyRef:
              name: my-configmap
              key: MYSQL_DB
```

```
root@kiran:~# vi mysql-configuration.yaml
root@kiran:~# kubectl apply -f mysql-configuration.yaml
deployment.apps/mysql-configuration created
```

```
root@kiran:~# kubectl get cm
NAME               DATA   AGE
kube-root-ca.crt   1      39m
```

Explanation:

In the env section, we define environment variables. For this MYSQL_DATABASE, we set its value using a reference to the my-configmap ConfigMap. This is done via configMapKeyRef, where we specify the name of the ConfigMap and the key we want to use. This way, the value of MYSQL_DB in the ConfigMap is injected as an environment variable into the container.

Kubernetes Secrets

Every software application is guaranteed to have some secret data. This secret data can range from database credentials to TLS certificates or access tokens to establish secure connections.
The platform you build your application on should provide a secure means for managing this secret data. This is why Kubernetes provides an object called **Secret** to store sensitive data you might otherwise put in a Pod specification or your application container image.

What are Kubernetes Secrets?

A Secret is an object that contains a small amount of sensitive data such as login usernames and passwords, tokens, keys, etc.
The primary purpose of Secrets is to reduce the risk of exposing sensitive data while deploying applications on Kubernetes.

Key Points about Kubernetes Secrets:

- You create Secrets **outside of Pods** — you create a Secret before any Pod can use it.

- When you create a Secret, it is stored inside the Kubernetes data store (i.e., an etcd database) on the Kubernetes Control Plane.
- When creating a Secret, you specify the data and/or stringData fields.
    - The values for all the data field keys must be **base64-encoded** strings.
    - If you don't want to convert to base64, you can specify the stringData field instead, which accepts arbitrary strings as values.
- When creating Secrets, you are limited to a size of **1MB per Secret**. This discourages creation of very large secrets that could exhaust memory.
- You can mark a Secret as **immutable** with immutable: true, which prevents accidental or unwanted updates after creation.
- After creating a Secret, you inject it into a Pod either by:
    - Mounting it as **data volumes**
    - Exposing it as **environment variables**
    - Using it as **imagePullSecrets**

Types of Kubernetes Secrets:

- **Opaque Secrets**: Default type. Stores arbitrary user-defined data.
- **Service Account Token Secrets**: Store token credentials tied to a service account. Must include the annotation kubernetes.io/service-account.name.
- **Docker Config Secrets**: Store credentials to access a container registry.
- **Basic Authentication Secret**: Used for credentials. Must contain:
    - username: the user name
    - password: the password or token
- **SSH Authentication Secrets**: Store SSH private keys with the key ssh-privatekey.
- **TLS Secrets**: Store certificates and keys with tls.key and tls.crt.
- **Bootstrap Token Secrets**: Store token data for the node bootstrap process. Created in the kube-system namespace and named bootstrap-token-<token-id>.

Hands-on

we are using the Secret for Mysql Database

Creating a Secret

Here's a YAML manifest file for creating a Secret named my-secret:

Before that, we are passing the password value into the data field in base64 encrypted format so let's make the password base64 encrypted.

```
root@kiran:~# echo -n 'kiran@123' | base64
a2lyYW5AMTIz
root@kiran:~# echo -n 'a2lyYW5AMTIz' | base64 --decode
kiran@123root@kiran:~# vi my-secret.yaml
```

```
apiVersion: v1
kind: Secret
metadata:
  namespace: mynamespace
  name: my-secret
type: Opaque
data:
  password: a2lyYW5AMTIz
~
~
~
```

```
root@kiran:~# kubectl apply -f my-secret.yaml
secret/my-secret created
```

Explanation:

type: Hre we use an opaque secret, allows for unstructured key:value pairs that can contain arbitrary values.

data: This is where you define key-value pairs for your configuration settings. In this example, we have 1 key-value pairs: password in a base64 encrypted format.

Using a Secret in a Pod

We have our Deployment file just add the new env variable attach the secret there and update the Deployment file.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql-configuration
  namespace: mynamespace
  labels:
    app: mysql
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mysql
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - name: mysql-container
        image: mysql:8
        ports:
        - containerPort: 3306
        env:
        - name: MYSQL_DATABASE
          valueFrom:
            configMapKeyRef:
```

```yaml
            configMapKeyRef:
              name: my-configmap
              key: MYSQL_DB
        - name: MYSQL_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: my-secret
              key: password
```

```
root@kiran:~# vi mysql-configuration.yaml
root@kiran:~# kubectl apply -f mysql-configuration.yaml
deployment.apps/mysql-configuration created
```

```
root@kiran:~# kubectl get secret -n mynamespace
NAME        TYPE     DATA   AGE
my-secret   Opaque   1      11m
```

```
root@kiran:~# kubectl get pod -n mynamespace
NAME                                READY   STATUS    RESTARTS   AGE
mysql-configuration-56cd666675-6gb2s   1/1     Running   0          29s
root@kiran:~# kubectl get pod -n mynamespace -o wide
NAME                                READY   STATUS    RESTARTS   AGE   IP             NODE                  NOMINATED NODE   READINESS GATES
mysql-configuration-56cd666675-6gb2s   1/1     Running   0          52s   100.96.0.109   i-0db63a50e11f1b1ec   <none>           <none>
root@kiran:~# kubectl get all -n mynamespace
NAME                                    READY   STATUS    RESTARTS   AGE
pod/mysql-configuration-56cd666675-6gb2s   1/1     Running   0          73s

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/mysql-configuration   1/1     1            1           58m

NAME                                           DESIRED   CURRENT   READY   AGE
replicaset.apps/mysql-configuration-56cd666675   1         1         1       73s
replicaset.apps/mysql-configuration-5786d97cd6   0         0         0       25m
replicaset.apps/mysql-configuration-57c565d6f6   0         0         0       48m
replicaset.apps/mysql-configuration-6748c8dcdd   0         0         0       30m
replicaset.apps/mysql-configuration-6999664d67   0         0         0       23m
replicaset.apps/mysql-configuration-76ddbdfb65   0         0         0       58m
root@kiran:~# kubectl get configmap -n mynamespace
NAME               DATA   AGE
kube-root-ca.crt   1      65m
my-configmap       1      65m
root@kiran:~# kubectl get secret -n mynamespace
NAME        TYPE     DATA   AGE
my-secret   Opaque   1      24m
root@kiran:~# kubectl describe configmap/my-configmap -n mynamespace
Name:         my-configmap
Namespace:    mynamespace
Labels:       <none>
Annotations:  <none>
```

```
Data
====
MYSQL_DB:
----
mydb


BinaryData
====

Events:  <none>
root@kiran:~# kubectl describe secret/my-secret -n mynamespace
Name:         my-secret
Namespace:    mynamespace
Labels:       <none>
Annotations:  <none>

Type:  Opaque

Data
====
password:  9 bytes
root@kiran:~# kubectl exec -n mynamespace mysql-configuration-56cd666675-6gb2s -c mysql-container -it -- /bin/sh
sh-5.1# mysql -u root -p
```

Login Into the container running inside the Pod using the command.

kubectl exec -n <namespace here> <pod-name> -c <container-name> -it -- /bin/sh

```
root@kiran:~# kubectl exec -n mynamespace mysql-configuration-56cd666675-6gb2s -c mysql-container -it -- /bin/sh
sh-5.1# mysql -u root -p a21yYW5AMTIz
```

Once you log in to the container you are inside the container then login to the MySQL database running inside the container using the command.

mysql -u root -p <your-decrepted password>

Then hit the command to show the database we have specified in the ConnfigMap file.

show databases;

```
sh-5.1# mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 12
Server version: 8.4.5 MySQL Community Server - GPL

Copyright (c) 2000, 2025, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases
    -> ;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mydb               |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
5 rows in set (0.00 sec)

mysql>
```

Summary

- ConfigMaps store configuration data and are excellent for decoupling configuration from application code.
- Secrets secure sensitive information and are indispensable for safeguarding credentials and certificates.

ConfigMaps and Secrets are like the keys to unlocking the full potential of your Kubernetes applications. Use them wisely to streamline configuration management and secure your sensitive data.

Best Practices for Handling Secrets in Kubernetes:

1. **Encrypt Secrets at Rest**
   Configure encryption for etcd to secure Secrets in storage.
2. **Limit Access with RBAC**
   Restrict who can view or modify Secrets using strict RBAC policies.
3. **Avoid Hardcoding Sensitive Data**
   Use Secrets to inject sensitive data dynamically.
4. **Use Minimal Permissions**
   Only provide the necessary permissions required by pods or users.
5. **Enable Automatic Secret Rotation**
   Automate Secret rotation using CI/CD pipelines or tools like SealedSecrets.
6. **Use External Secrets Manager for Critical Workloads**
   For highly sensitive data, rely on an external Secret management solution.
7. **Monitor and Audit**
   Continuously monitor for unauthorized access to Secrets and audit logs regularly.
8. **Disable Secrets Access to Unauthorized Pods**
   Implement network policies to prevent accidental exposure.