**ConfigMap and Secrets**

* A ConfigMap is plain text file it stores configuration data as **key-value pairs** in **plain text** (unencrypted).
* When you create a ConfigMap, Kubernetes stores it in **etcd** (its database).
* By default, **etcd does not encrypt ConfigMaps,** so they remain in plain text.
* **This is why you should never store passwords or secrets in a ConfigMap (Use Secretes instead).**

A ConfigMap holds configuration settings, like database connection strings, application properties, environment variables, database URLs, API keys, etc. separately from your application code.

**Why Do We Use ConfigMap?**

1. **Decouples configuration from the app** – You don’t need to change or rebuild your application just to update settings (like configuring new data base connection).
2. **Makes deployments flexible** – You can update configurations settings without restarting the entire application.
3. **Helps manage multiple environments** – Different configurations for **dev**, **test**, and **production** can be easily managed.
4. **Keeps sensitive and non-sensitive data separate** – Though **ConfigMap** store non-sensitive data, they work well alongside **Secrets** for sensitive data.
5. **Share Configurations between Pods**: Multiple apps (Pods) can use the same ConfigMap (e.g., a shared logging level or API endpoint).
6. ConfigMap can be used/passed in **two ways**:
   1. **As Environment Variables** (e.g., DB\_HOST=mysql-service).
   2. **As Files** (e.g., a config.properties file mounted inside the container). **(inside a Volume)**
7. Your app (Pod) **reads** the ConfigMap when it starts.

ConfigMap makes applications more portable, easier to manage, and flexible for different environments.

**Example:**

Let’s say your application needs to establish a connection to database using URL. Instead of hardcoding it inside the application pod, we store the database URL in a ConfigMap as show in below fig:

apiVersion: v1

kind: ConfigMap

metadata:

name: my-config

data:

DATABASE\_URL: "postgres://user:password@db-service:5432/mydb"

Fig: ConfigMap.

apiVersion: v1

kind: Pod

metadata:

name: my-app

spec:

containers:

- name: app-container

image: my-app:latest

env:

- name: DATABASE\_URL

valueFrom:

configMapKeyRef:

name: my-config

key: DATABASE\_URL

Fig: Using of ConfigMap inside an APP pod.

**Note1:** **Yes, you need to restart the Pod** if the ConfigMap is passed as an **environment variable.**

**Note2:** When you mount a ConfigMap as a **file (volume)**, Kubernetes will update the file inside the Pod without restarting the pod.

**Let’s work with ConfigMap practically:**

**ConfigMap using as Environmental variable**

**Step1:** Create the ConfigMap.



apiVersion: v1

kind: ConfigMap

metadata:

  name: test-cm

data:

  db-port: "3306"

Fig: cm.yaml file (ConfigMap file).

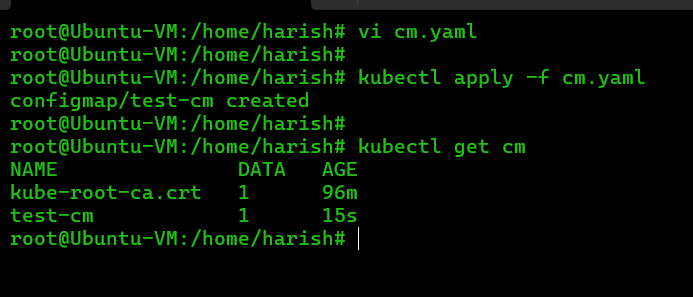


Fig: ConfigMap is created.

**Step2:** Create the deployment.yaml file by configuring the ConfigMap as a environmental variable in it.

apiVersion: apps/v1

kind: Deployment

metadata:

name: myapp1-deployment

spec:

replicas: 1

selector:

matchLabels:

app: myapp1

template:

metadata: # Dictionary

name: myapp1-pod

labels: # Dictionary

app: myapp1

spec:

containers: # List

- name: myapp1-container

image: kuchalakantikris/my\_nginx\_image1:v1

env:

- name: DB-PORT

valueFrom:

configMapKeyRef:

name: test-cm

key: db-port

ports:

- containerPort: 80

Fig: deployment.yaml file.

In above deployment.yaml file we configured or passed the ConfigMap as an environmental variable as shown in above figure.

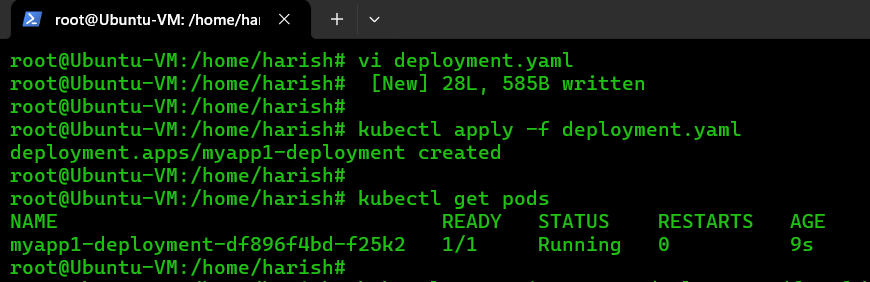


Fig: Application pod is created successfully.

Step3: Check whether the database port (DB-PORT) is attached or not.

In order to check first we have to enter into the app pod and search for environmental variable.

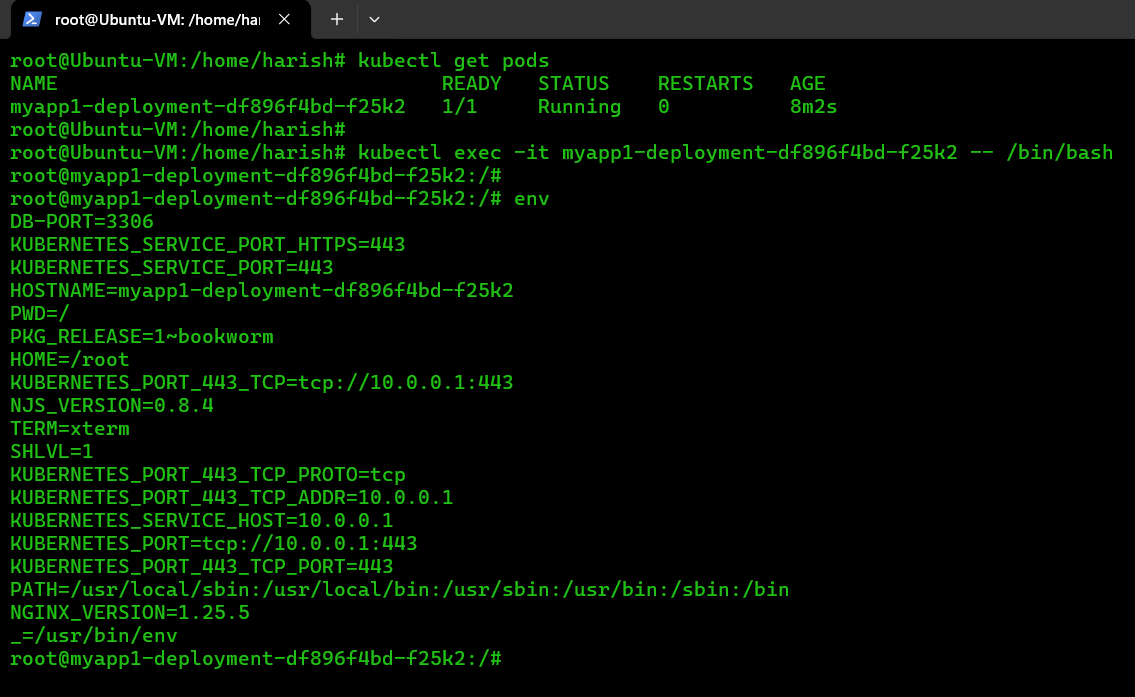


Fig: database port (DB-PORT=3306) is successfully attached to the application pod.

In the previous task, we passed the ConfigMap as an **environment variable** in the application Pod (deployment.yaml). As we know, when a ConfigMap passed as an environmental variable, any changes or updates made to it will **not** be automatically reflected in the running Pod.

To apply the updated ConfigMap, we must **restart the Pod**, as environment variables are only loaded when the container starts.

Let’s experience the above situation by changing/updating the ConfigMap.

step1: Change the database port (db-port=3306) to 3307.

step2: Delete and restart the application pod (deployment.yaml file).

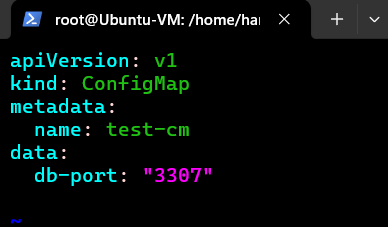


Fig: Updated ConfigMap (cm.yaml file)

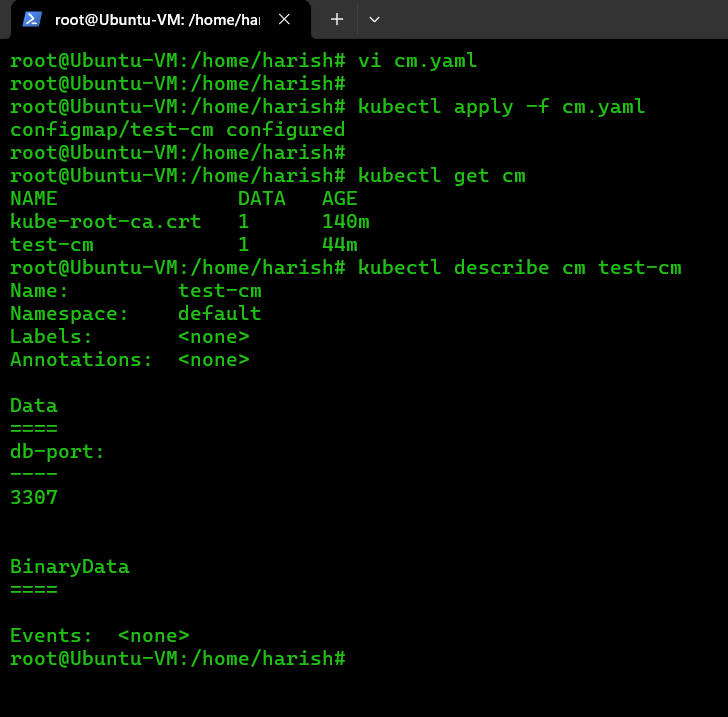


Fig: ConfigMap is updated/changed successfully.

Now delete and restart the application pod (myapp1-deployment) in order to reflect the changes of ConfigMap in the application pod.

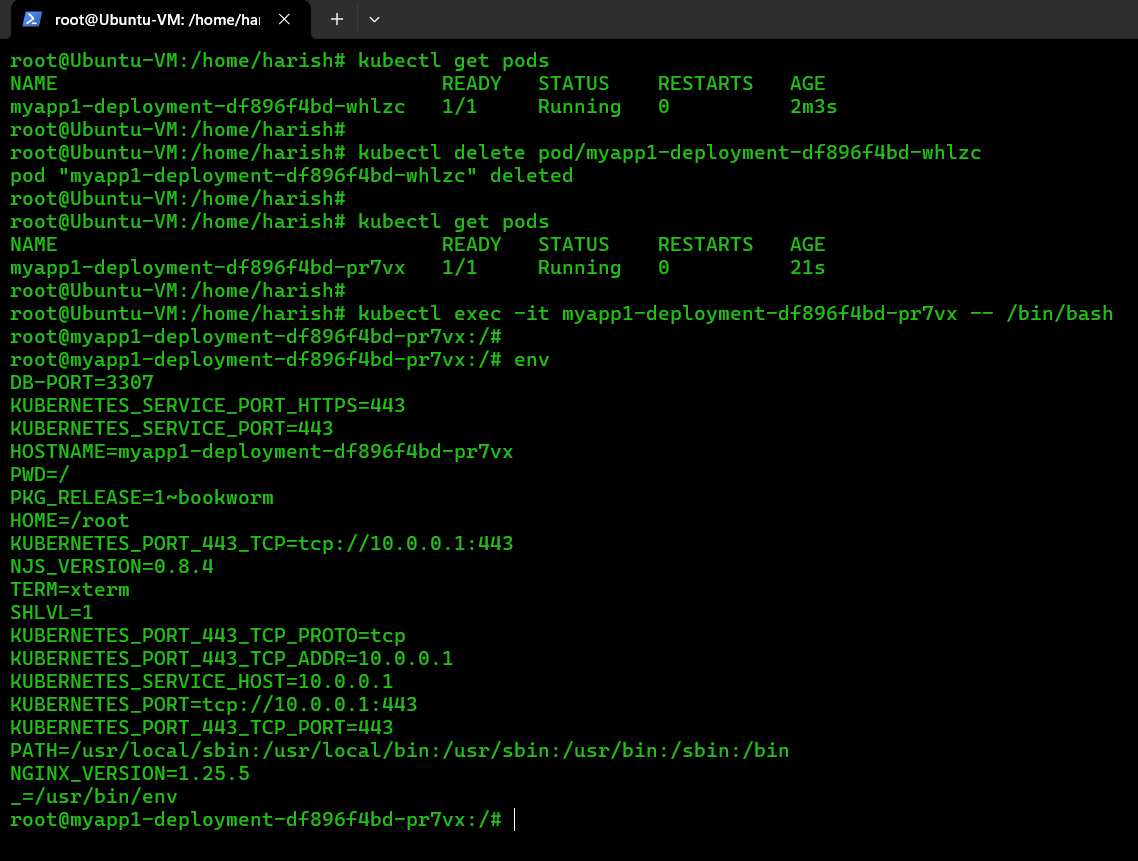


Fig: App pod is deleted and restarted so changes of ConfigMap are reflected on it.

**Disadvantage:**

Restarting of app pods is required, in order to reflect the changes/updates of the ConfigMap in to the application pod.

**ConfigMap Using file (Volume)**

To overcome the above disadvantage which is faced when we passed the ConfigMap as an environmental variable in an application pod we use the files (volumes).

In this approach we place the ConfigMap (plain text file or template) in a volume which acts as a centralized location.

Here instead of passing ConfigMap values as **environment variables**, we can **mount** a ConfigMap as files inside a Pod. This is known as “**ConfigMap file passing”** or “**ConfigMap volume mounting”.**

### ****How to Pass a ConfigMap as a File?****

1. **Create a ConfigMap** that stores configuration data as key-value pairs.
2. **Mount the ConfigMap as a volume** inside the Pod.
3. **The application reads the configuration from the file** inside the container.

Let’s Work with ConfigMap file passing:

Step1: create the ConfigMap using YAML file (cm.yaml file).

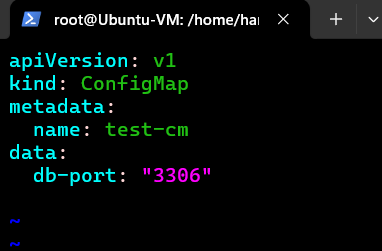


Fig: cm.yaml file (ConfigMap).

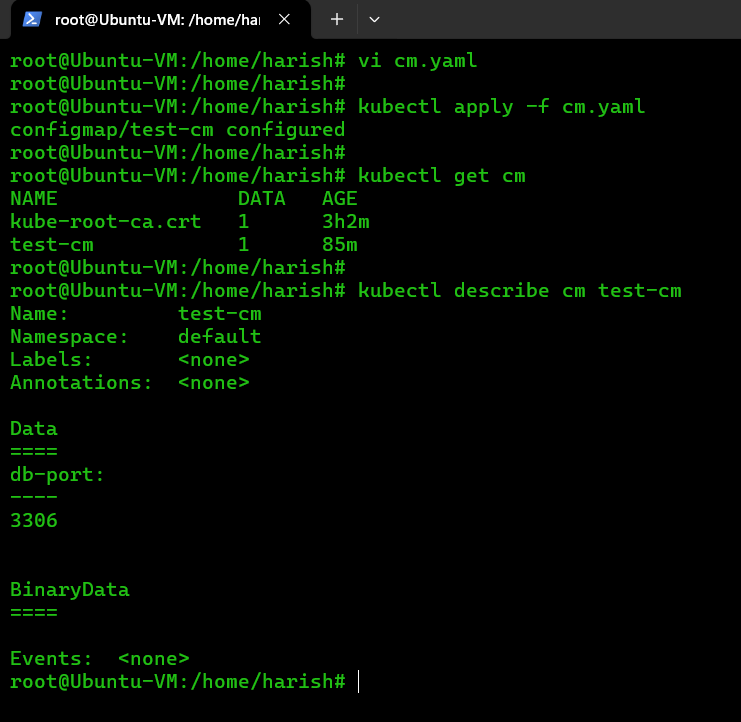
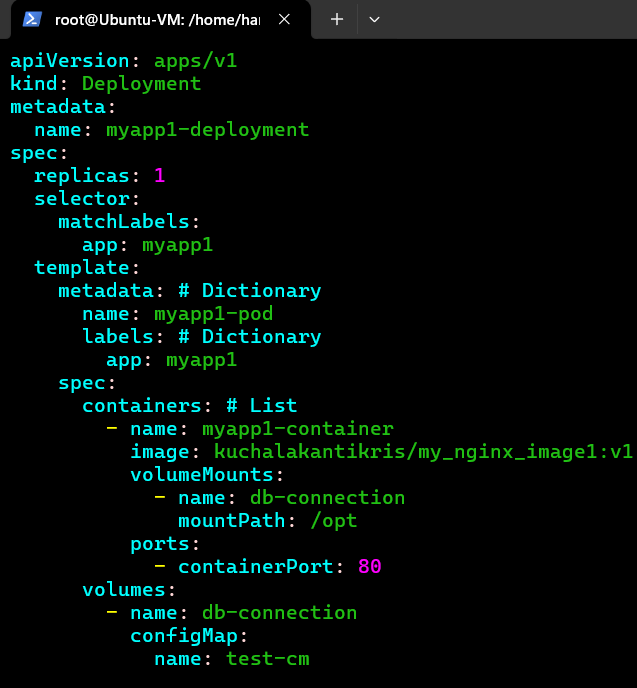


Fig: ConfigMap is created successfully using cm.yaml file.

Step2: Create application pod by configuring the volume for the ConfigMap.

**Fig:** deployment.yaml file.

apiVersion: apps/v1

kind: Deployment

metadata:

  name: myapp1-deployment

spec:

  replicas: 1

  selector:

    matchLabels:

      app: myapp1

  template:

    metadata: # Dictionary

      name: myapp1-pod

      labels: # Dictionary

        app: myapp1

    spec:

      containers: # List

        - name: myapp1-container

          image: kuchalakantikris/my\_nginx\_image1:v1

          volumeMounts:

            - name: db-connection

              mountPath: /opt

          ports:

            - containerPort: 80

      volumes:

        - name: db-connection

          configMap:

            name: test-cm

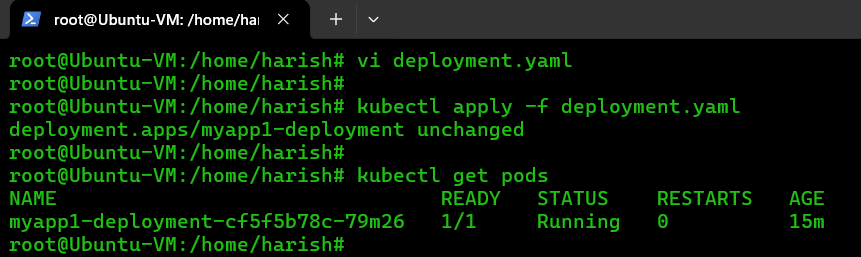


Fig: App pod is created by configuring the Volume in it.

Now let’s enter into the app pod and check whether the ConfigMap is mounted in a volume or not.

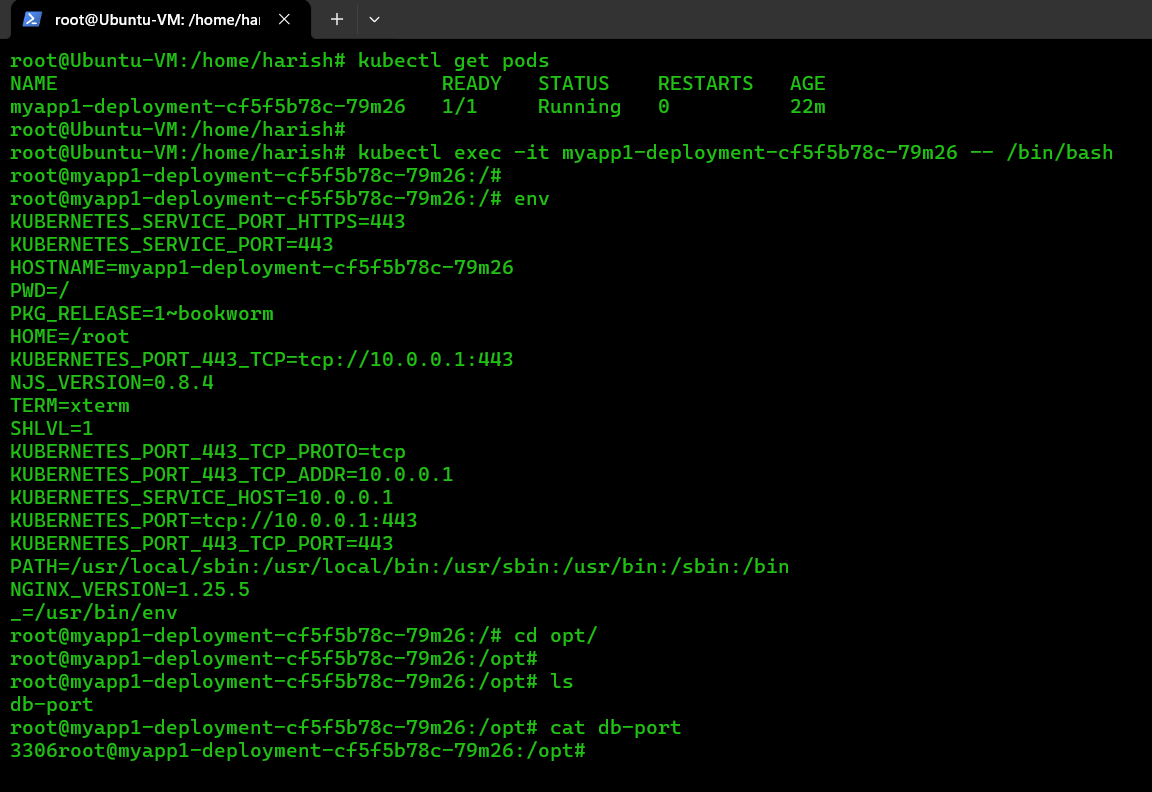


Fig: ConfigMap is passed as a file (volume) in the app pod.

**To view the port clearly use the command:**

cat /opt/db-port | more

**Or**

kubectl exec -it <pod-name> -- cat /opt/db-port

Here we configured a volume with a directory “opt” and mounted the ConfigMap to this directory of the volume so it can be used as a centralized location.

Let’s make any update/change in the ConfigMap and check whether the changes are reflected quickly in the application pod without restarting the pod or not.

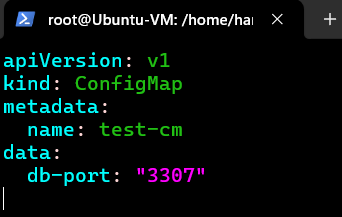


Fig: Updated ConfigMap file (cm.yaml).

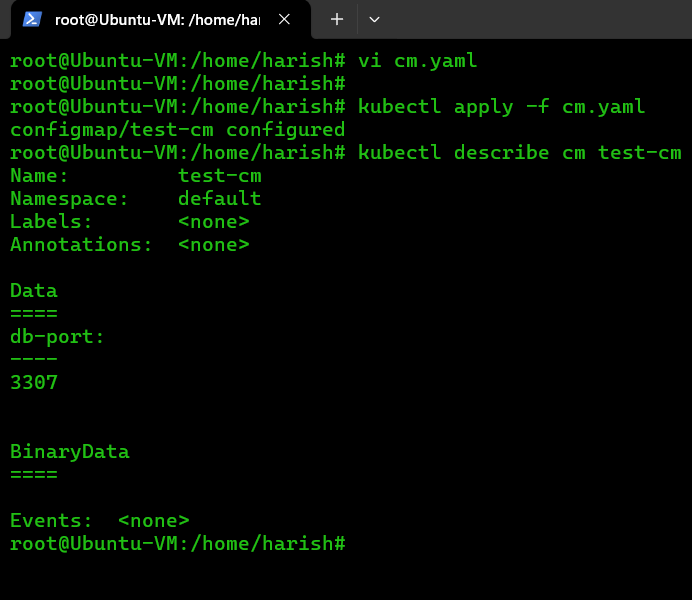


Fig: Updated ConfigMap.

Check whether the update/changes of ConfigMap is reflected in the app pod or not.

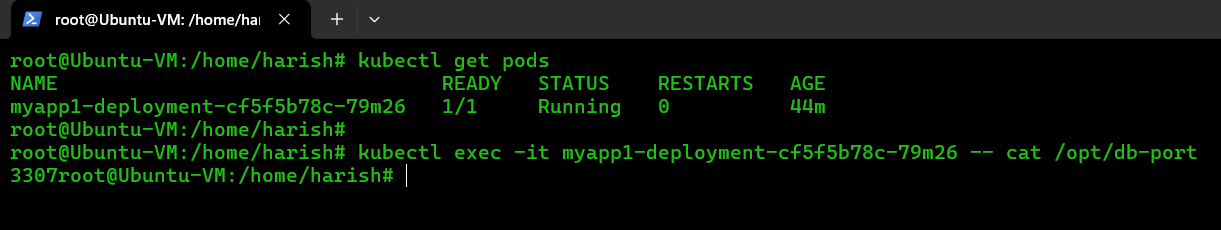


Fig: Changes of the ConfigMap are reflected in the app pod without restart.

**Advantage:**

We no need to restart the pod when the ConfigMap is updated/changed its settings in order to reflect the changes in the pod.

**Secrete:**

A **Secret** in Kubernetes is a way to store **sensitive data** (like passwords, API keys, SSH keys, or certificates) separately from your application code and configuration. It is similar to a **ConfigMap,** but **Secrets are encrypted** and **more secure**.

**Features:**

**Security** – Secrets are stored in an encoded format (Base64) and can be encrypted in Kubernetes storage.  
**Easier Management** – Secrets can be updated without modifying or exposing sensitive information in deployment files.  
**Better Access Control** – Kubernetes RBAC (Role-Based Access Control) can restrict who can view or modify Secrets.

We can create the Secrete in two ways in Kubernetes:

* 1. Manually (using Kubctl cli)
  2. Using YAML file.

1. **Manually:**

The command to create the Secrete using kubctl CLI is

**Command:**

kubectl create secret generic my-secret --from-literal=username=admin --from-literal=password=MyP@ssword

This will create a Secret named “my-secret” with two key-value pairs (username and password).

**EX:** kubectl create secrete generic test-secret --from-literal=db-port=”3306”

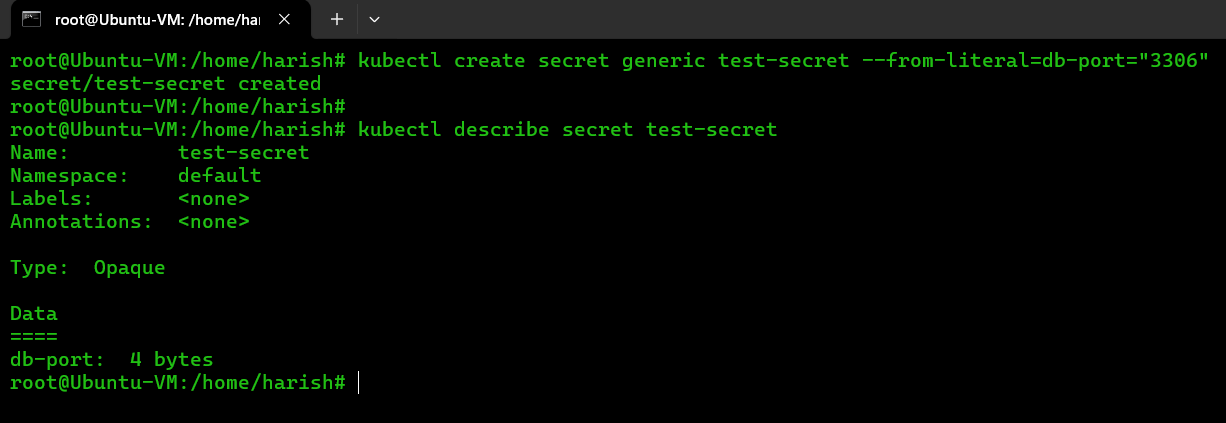
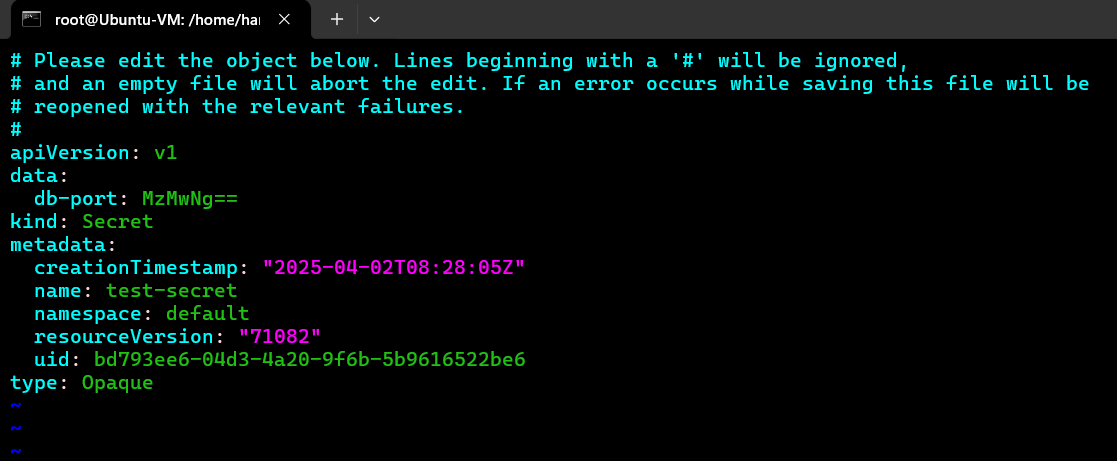


Fig: Secrete is created by encrypting the db-port.

In above figure Secrete is created by encrypting the db-port=3306, it only showing the size of the port not the exact port number.

By using the below command we can view the manually created secrete.

**Command:** kubectl edit secret test-secret



We can also decode the above encrypted db-port by using below command.

**Command:**  echo MzMwNg== | base64 --decode | more

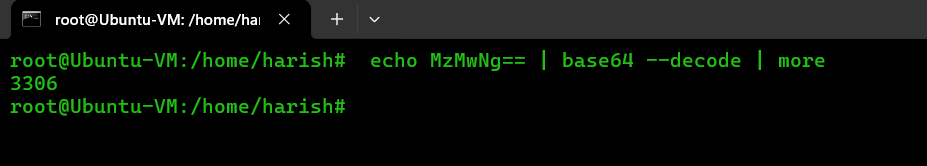


Fig: Decoding of encrypted db-port.

1. **Creating of Secret Using YAML file:**

By using YAML file we can create secrete, this created secrete is used/passed as environmental variable or mount as file in the application pod.

**Note:** You can pass a Secret to a Pod as **environment variables** or **mounted as files**.

**Pass Secret as an Environmental variable:**

**Step1:** Create YAML file to create Secret.

apiVersion: v1

kind: Secret

metadata:

name: my-secret

type: Opaque

data:

username: YWRtaW4= # Base64 encoded "admin"

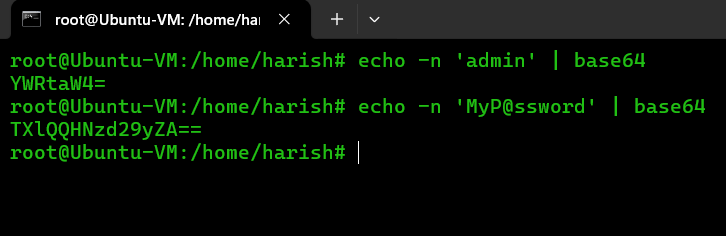
password: TXlQQHNzd29yZA== # Base64 encoded "MyP@ssword"

Fig: secrete.yaml file.

**Important-Note:** Values must be **Base64-encoded**. You can encode them using below commands.

* echo -n 'admin' | base64
* echo -n ‘password’ | base64

**Example:** kubectl create secret generic my-secret --from-literal=username=admin --from-literal=password=MyP@ssword



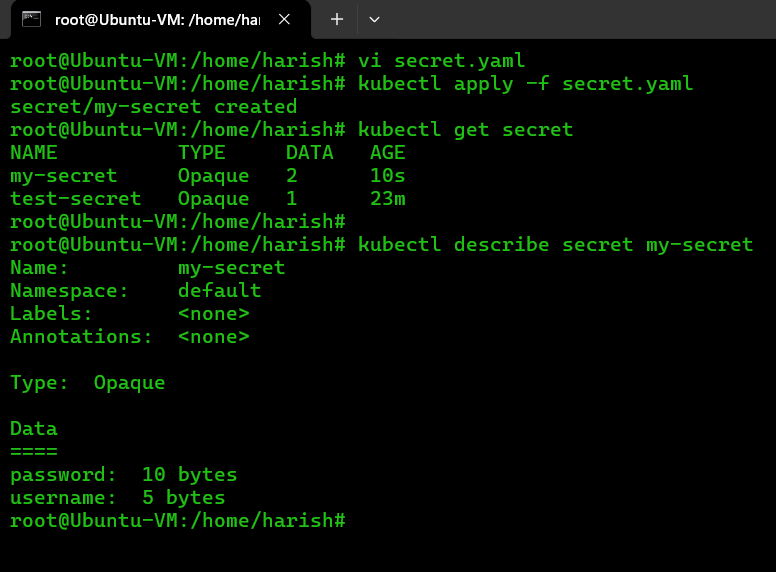


Fig: Secrete is created with name my-secret.

Step2: Create an app pod and configure the secret as an environmental variable in it.

apiVersion: v1

kind: Pod

metadata:

name: my-app

spec:

containers:

- name: app-container

image: my-app:latest

env:

- name: DB\_USER

valueFrom:

secretKeyRef:

name: my-secret

key: username

- name: DB\_PASSWORD

valueFrom:

secretKeyRef:

name: my-secret

key: password

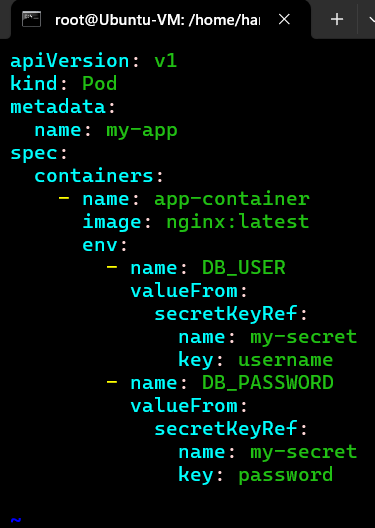


Fig: deployment.yaml.

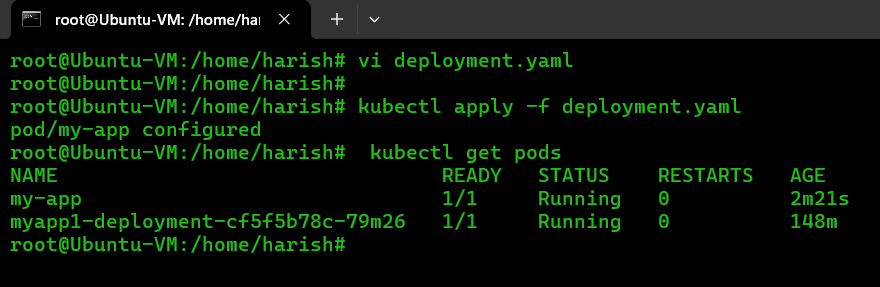


Fig: App pod is running successfully by passing secret as environment variable.

Inside the container, the application can read DB\_USER and DB\_PASSWORD from environment variables.

**Mount Secret as a File (Volume):**

Here we mount the secret to the application pod by creating the volume in it.

apiVersion: v1

kind: Pod

metadata:

name: my-app

spec:

containers:

- name: app-container

image: my-app:latest

volumeMounts:

- name: secret-volume

mountPath: "/etc/secret"

volumes:

- name: secret-volume

secret:

secretName: my-secret

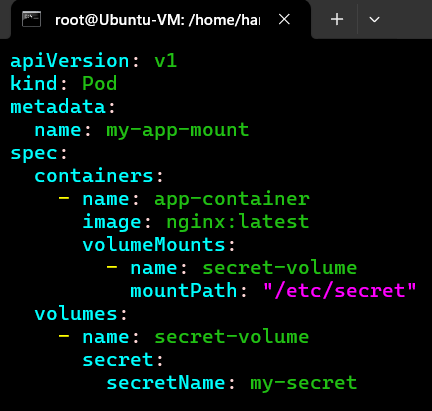


Fig: deployement01.yaml

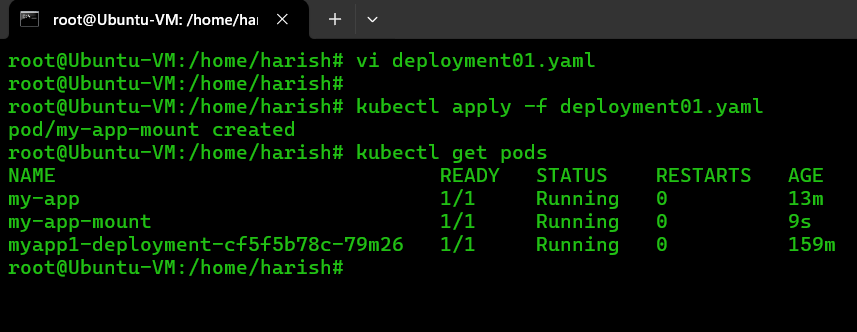


Fig: APP pod is created by cpassing the Secret as a file (volume).