Kubernetes Labs: Practical Usage of ConfigMaps and Secrets

In this section, we will walk through two labs: one demonstrating the usage of **ConfigMaps** and another demonstrating the usage of **Secrets**. Both labs are designed to give you hands-on experience with real-world scenarios.

Lab 1: Using Kubernetes ConfigMap to Externalize Application Configuration

Objective:

Create a ConfigMap to externalize the configuration of a web application and inject it into the container as both environment variables and a mounted file.

Prerequisites:

- A running Kubernetes cluster (Minikube, KinD, or a cloud provider-managed cluster).
- `kubectl` CLI installed and connected to the cluster.

Steps:

Step 1: Create the ConfigMap

First, create a ConfigMap that stores configuration values for a sample web application. These values include a database connection string, log level, and maximum connections.

```
kubectl create configmap app-config \
  --from-literal=DATABASE_URL="mysql://db:3306/mydb" \
  --from-literal=LOG_LEVEL="INFO" \
  --from-literal=MAX_CONNECTIONS="100"
```

You can verify the ConfigMap with the following command:

```
kubectl get configmap app-config -o yaml
```

Step 2: Create a Deployment to Use the ConfigMap

Create a YAML file for a Deployment that uses the `app-config` ConfigMap. We will inject the ConfigMap data into the container as both environment variables and a mounted configuration file.

Create the following file called `deployment.yaml`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: configmap-demo
spec:
  replicas: 1
  selector:
    matchLabels:
      app: demo-app
  template:
    metadata:
      labels:
        app: demo-app
    spec:
      containers:
      - name: demo-container
        image: nginx
        env:
        - name: DATABASE_URL
          valueFrom:
            configMapKeyRef:
              name: app-config
              key: DATABASE_URL
        - name: LOG_LEVEL
          valueFrom:
            configMapKeyRef:
              name: app-config
              key: LOG_LEVEL
        - name: MAX_CONNECTIONS
          valueFrom:
            configMapKeyRef:
              name: app-config
              key: MAX_CONNECTIONS
        volumeMounts:
        - name: config-volume
          mountPath: /etc/config
      volumes:
      - name: config-volume
        configMap:
          name: app-config
```

Step 3: Deploy the Application

Apply the `deployment.yaml` file to your cluster:

```
kubectl apply -f deployment.yaml
```

Step 4: Verify the Environment Variables and Mounted File

1. **Check Environment Variables**: Access the running Pod and check the environment variables injected from the ConfigMap:

   ```
   kubectl get pods
   kubectl exec -it POD_NAME -- env
   ```

   You should see the following output:

   ```
   # --- other variables
   DATABASE_URL=mysql://db:3306/mydb
   LOG_LEVEL=INFO
   MAX_CONNECTIONS=100
   # -- other variables
   ```

2. **Check the Mounted File**: Verify that the ConfigMap data is mounted as a file inside the container:

   ```
   kubectl exec -it POD_NAME -- cat /etc/config/DATABASE_URL
   ```

   This should output:

   ```
   mysql://db:3306/mydb
   ```

Step 5: Clean Up

After completing the lab, clean up the resources by running:

```
kubectl delete deployment configmap-demo
kubectl delete configmap app-config
```

Lab 2: Managing Sensitive Data Using Kubernetes Secrets

Objective:

Create and use a Kubernetes Secret to securely store a database password and use it in a containerized application.

Prerequisites:

- A running Kubernetes cluster (Minikube, KinD, or a cloud provider-managed cluster).
- `kubectl` CLI installed and connected to the cluster.

Steps:

Step 1: Create the Secret

First, create a Secret to store a sensitive password for a database.

```
kubectl create secret generic db-secret --from-literal=DB_PASSWORD="supersecretpassword"
```

You can verify the Secret with the following command:

```
kubectl get secret db-secret -o yaml
```

Notice that the password is base64-encoded in the output.

Step 2: Create a Deployment to Use the Secret

Next, create a Deployment that uses the `db-secret` Secret. The database password will be injected into the container as an environment variable.

Modify the existing deployment file to be as follows:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: secret-demo
spec:
  replicas: 1
  selector:
    matchLabels:
      app: demo-app
  template:
    metadata:
      labels:
        app: demo-app
    spec:
      containers:
      - name: demo-container
        image: nginx
        env:
        - name: DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: db-secret
              key: DB_PASSWORD
```

Step 3: Deploy the Application

Apply the `secret-deployment.yaml` file to your cluster:

```
kubectl apply -f deployment.yaml
```

Step 4: Verify the Secret in the Pod

1. **Access the Running Pod**: Get the Pod name and verify that the secret was injected correctly as an environment variable:

   ```
   kubectl get pods
   kubectl exec -it POD_NAME -- env
   ```

   You should see the following output:

   ```
   DB_PASSWORD=supersecretpassword
   ```

2. **Check that the Secret is not Exposed in Plain Text**: Notice that when retrieving the Pod's definition using `kubectl`, the actual secret value is not exposed:

   ```
   kubectl get pod $POD_NAME -o yaml
   ```

   The output will only show the secret reference, not the actual password:

   ```
   - name: DB_PASSWORD
     valueFrom:
       secretKeyRef:
         key: DB_PASSWORD
         name: db-secret
   ```

Step 5: Clean Up

After completing the lab, clean up the resources by running:

```
kubectl delete deployment secret-demo
kubectl delete secret db-secret
```