Table of Contents

Kubernetes Objects

- Pods
- ReplicaSets
- Deployments
- Namespaces
- Service
- Configmap
- Secrets

Configure Environment Variables In Applications

ENV variables in Docker

- We can use the **-e, --env, and --env-file** flags to set environment variables in the container, or overwrite variables that are defined in the Dockerfile of the image you're running.

```
docker run --env SDLC_ENV=dev -it ubuntu bash
echo $SDLC_ENV
```

- Here, the **SDLC_ENV** is available as a environment variable inside the container.

--

ENV variables in Kubernetes

- To set an environment variable set an env property in pod definition file.

```
apiVersion: v1
kind: Pod
metadata:
  name: new-testapp-pod
  labels:
     app: new-testapp-pod
     type: front-end-pod
spec:
  containers:
  - name: nginx-container
    image: nginx:1.16.1
    env:
    - name: SDLC_ENV
      value: dev
```

--

Configmap

- What Is a Kubernetes ConfigMap?
    - A Kubernetes ConfigMap is an object that allows you to store data as **key-value** pairs
    - Kubernetes pods can use ConfigMaps as **configuration files, environment variables or command-line arguments**.
    - ConfigMaps allow you to decouple environment-specific configurations from containers to make applications portable.
    - However, they are not suitable for confidential data storage.
    - ConfigMaps are not encrypted in any way, and all data they contain is visible to anyone who can access the file.
    - You can use **Kubernetes Secrets** to store sensitive information.
    - Another potential drawback of ConfigMaps is that files must be limited to 1MB. Larger datasets may require different storage methods, such as separate file mounts, file services or databases.
    - Here we will check how to create ConfigMaps and configure Pods using data stored in ConfigMaps.

--

Creating,Viewing,Using ConfigMaps

- A ConfigMap is a dictionary of key-value pairs that store configuration settings for your applications.

1. Create a ConfigMap in your cluster using kubectl command with a YAML file.
2. Consume to ConfigMap in your Pods and use its values.

- The simplest way to create a ConfigMap is to store a bunch of key-value strings in a ConfigMap YAML file and inject them as environment variables into your Pods.
- After that, you can reference the environment variables in your application container using whatever methods is necessary for your programming language.

--

Creating a ConfigMap

- To create a new ConfigMap, use this kubectl command:
  - **kubectl create configmap NAME DATA-SOURCE**
  - The **NAME** is the name of the ConfigMap, The **DATA-SOURCE** indicates the files or values from which ConfigMap data should be obtained.
  - You can create ConfigMaps based on one file, several files, directories, or env-files (lists of environment variables). The basename of each file is used as the key, and the contents of the file becomes the value.

--

- The below YAML file creates a ConfigMap with the value database set to mongodb, and database_uri, and keys set to the values in the YAML example code.
- Then, create the ConfigMap in the cluster using **kubectl apply -f config-map.yaml**

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: example-configmap
data:
  # Configuration values can be set as key-value properties
  database: mongodb
  database_uri: mongodb://localhost:27017
  sdlc_env: dev

  # Or set as complete file contents (even JSON!)
  keys: |
    image.public.key=771
    rsa.public.key=42
```

--

Viewing a ConfigMap

```
kubectl get configmaps

kubectl apply -f config-map.yaml
```

```
kubectl describe configmap example-configmap
```

Using a ConfigMap in Environment Variables

- **envFrom** property in the YAML file is used to add your ConfigMap as environment variables to your pods.
- Set **envFrom** to a reference to the ConfigMap you've created.

nginx-pod-env-var.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod-env-var
  labels:
     app: nginx-pod-env-var
     type: front-end-pod
spec:
  containers:
  - name: nginx-pod-env-var-configmap
    image: nginx:1.16.1
    envFrom:
    - configMapRef:
        name: example-configmap
```

--

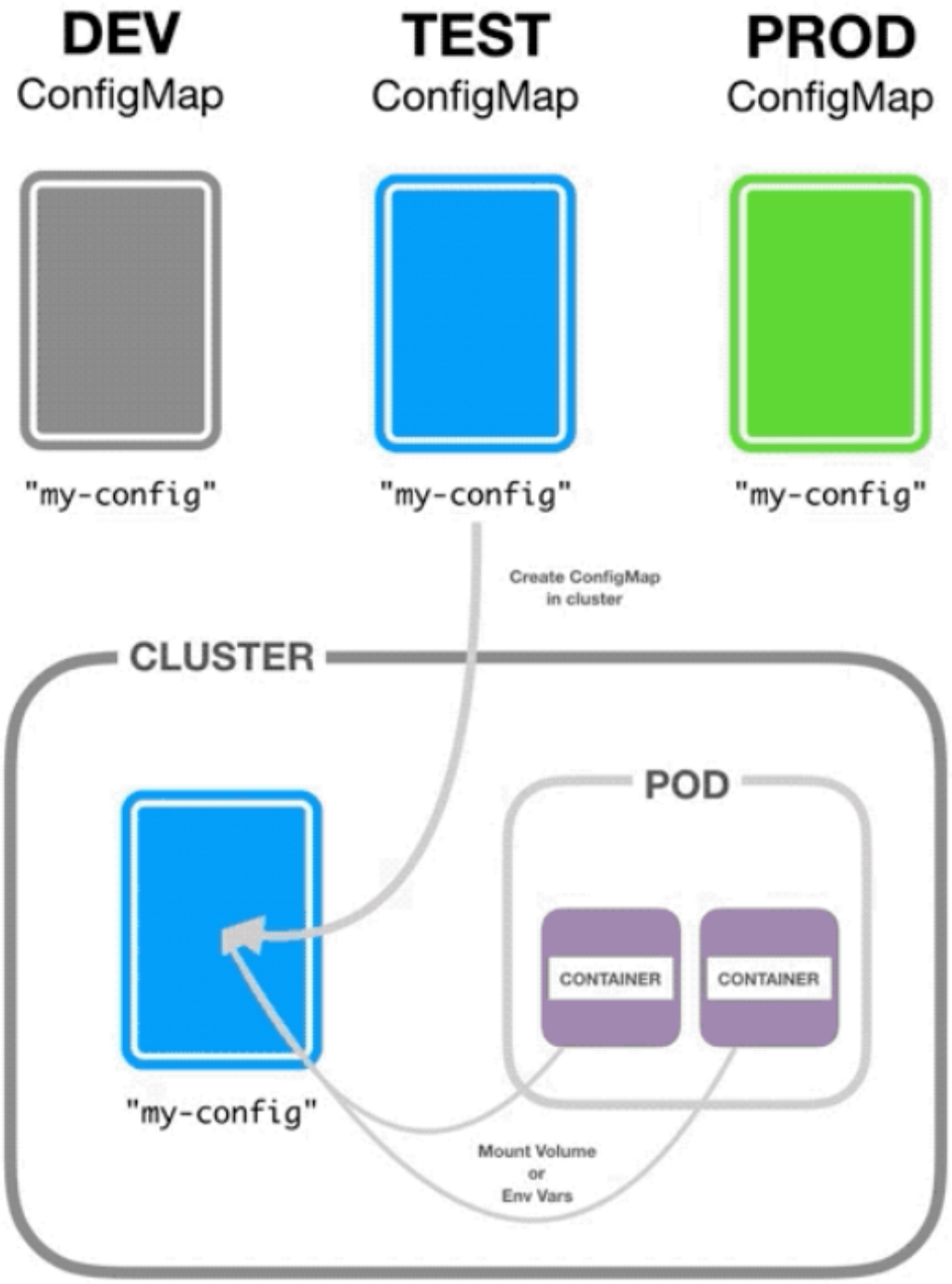
- Create the Pod, to access these environment variables set in the ConfigMap **example-configmap**

```
kubectl apply -f nginx-pod-env-var.yaml
kubectl get pods
kubectl describe pod nginx-pod-env-var
# OUTPUT
    Environment Variables from:
      example-configmap  ConfigMap  Optional: false
kubectl exec -it nginx-pod-env-var bash
printenv
# All the configmap key-value data as configuration details will be available
inside the pod.
```

- The values defined in the ConfigMap are available as environment variables inside the pod.
- Create a ConfigMap for each SDLC Environment:
    - dev-example-configmap
    - qa-example-configmap
    - prod-example-configmap

- When you launch Pods/Deployments for specific environment, respected ConfigMap can be attached.

--



Secrets

- Your containerized applications need certain data or credentials to run properly, but how you store that data/credentials and make it available to the pod/container is important.
- Sensitive data like passwords or tokens are often required for operation of workloads in Kubernetes.
- **Kubernetes Secrets** are the mechanism that facilitates the use of the sensitive data, in a way that does not expose them when defining or viewing the operations of Kubernetes itself.
- Kubernetes Secrets are objects meant to store sensitive data consumed by the application Pods.

--

How and Why to Create a Kubernetes Secret

- A `Secret` object stores sensitive data used by Pods to access services.
    - Here you might need a Secret to store the **username** and **password** needed to access a **database**.
- The data stored in secrets is **base64** encoded, meaning that encoded data is not in an encrypted form, and anyone can easily decode it back to the original plaintext.
- Since a user can create Kubernetes Secrets separately from Pods, updating and deleting the Pods does not impact the Secrets.
- Secrets can be mounted into the Pods or passed as environment variables.
- The pods can import an external secret and manipulate/change it internally.
- But the original, external secret will remain unaffected.
- **Secrets** are functionally similar to **ConfigMaps**.
    - The only difference is that the data in the Kubernetes Secrets is **base64** encoded.
- You can create the Secret by passing the raw data in the command, or by storing the credentials in files that you pass in the command.
- The following commands create a **Secret** that stores the username `admin` and the password `S!B\*d$zDsb=`.

--

- We can create secrets using two methods:
    - Create Kubernetes Secret using **kubectl**.
    - Create Kubernetes Secret using a YAML manifest file

--

Create Kubernetes Secrets using kubectl

- There are two ways of providing the Secret data to kubectl when creating Secrets using Kubectl, and there are:
    - Providing the secret data through a file using the --from-file=**filename** tag OR
    - Providing the literal secret data using the --from-literal=**key**=**value** tag
- For creating a Secret with **kubectl** provide the **Secret data** from a file in a directory.

```
echo -n 'admin' > username.txt
echo -n 'password123' > password.txt
ls -ltr
cat username.txt
cat password.txt
```

- The **-n flag** in the above command ensures that no newline character is added at the end of the text.

--

- Create the Kubernetes Secret with the files using the **kubectl** command below, here pass the key name and value as file paths in the `kubectl` command by using `--from-file=[key=]source`:

```
kubectl create secret generic database-credentials-1 \
--from-file=username=username.txt \
--from-file=password=password.txt

kubectl get secrets
# Describing a Kubernetes Secret
kubectl describe secret database-credentials-1

# Below command will output the encoded key-value pairs of the secret data

kubectl get secret database-credentials-1 -o jsonpath='{.data}'
kubectl get secret database-credentials-1 -o jsonpath='{.data.username}'
kubectl get secret database-credentials-1 -o jsonpath='{.data.password}'

# Decoding a Kubernetes Secret
kubectl get secret database-credentials-1 -o jsonpath='{.data.username}' | base64
--decode
kubectl get secret database-credentials-1 -o jsonpath='{.data.password}' | base64
--decode
```

- Raw secret data into **kubectl** command

```
kubectl create secret generic database-credentials-2 \
    --from-literal=username=admin \
    --from-literal=password='password123'

kubectl get secrets

kubectl describe secret database-credentials-2
```

--

Create Kubernetes Secret using a YAML manifest file

- Before you create a **Secret** using a manifest file, you must first decide how you want to add the Secret data using the **data** field and/or the **stringData** field.
- Using the data field, you must encode the secret data using **base64**.
- To convert the **username** and **password** to **base64**, run the following command:

```
# For encoding a value for username
echo -n "admin" | base64
# OUTPUT
YWRtaW4=
# For encoding a value for password
```

```
echo -n "password123" | base64
# OUTPUT
cGFzc3dvcmQxMjM=
# For decoding the above values use below commands
echo -n "YWRtaW4=" | base64 -d
echo -n "cGFzc3dvcmQxMjM=" | base64 -d
```

--

- Now create a **demo-secret.yaml** manifest file and add the following configuration:

- demo-secret.yaml

```
apiVersion: v1
kind: Secret
metadata:
    name: database-credentials
type: Opaque
data:
    username: YWRtaW4=
    password: cGFzc3dvcmQxMjM=
```

- In the above YAML manifest file, the **username** and **password** values in the data field are the **base64** encoded values of the original credentials.
- When using the **stringData** field, the manifest file will be:

- demo-secret.yaml

```
apiVersion: v1
kind: Secret
metadata:
    name: database-credentials
type: Opaque
stringData:
    username: admin
    password: password123
```

--

- Execute the below kubectl commands to create and view the secret.

```
kubectl apply -f demo-secret.yaml

kubectl get secrets
```

```
kubectl describe secret database-credentials

kubectl get secret database-credentials -o jsonpath='{.data}'
kubectl get secret database-credentials -o jsonpath='{.data.username}'
kubectl get secret database-credentials -o jsonpath='{.data.password}'
```

--

Use Kubernetes Secrets inside a Pod

- The following are the ways a Pod can use a Secret:
  - As container environment variables.
  - As files in a volume mounted on one or more of its containers.

--

Using Secret data as container environment variables

- Below is a Pod YAML manifest with the Kubernetes Secret data you created exposed as environment variables.
- Create a file **secret-test-env-pod.yaml** and paste the configuration in it.

secret-test-env-pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: secret-test-env-pod
spec:
  containers:
    - name: secret-test-container
      image: nginx
      env:
        - name: USER
          valueFrom:
            secretKeyRef:
              name: database-credentials
              key: username
        - name: PASSWORD
          valueFrom:
            secretKeyRef:
              name: database-credentials
              key: password
```

--

- Execute the below kubectl commands to create pod and and view the secret details attached to the Pod.

```
kubectl apply -f secret-test-env-pod.yaml

kubectl get pods

kubectl describe pod secret-test-env-pod
# OUTPUT
    Environment:
      USER:      <set to the key 'username' in secret 'database-credentials'>
Optional: false
      PASSWORD:  <set to the key 'password' in secret 'database-credentials'>
Optional: false

# Running individual commands in a container
kubectl exec secret-test-env-pod -- printenv
kubectl exec -it secret-test-env-pod bash

kubectl get secrets
kubectl get secret database-credentials -o jsonpath='{.data}'
```

--

Using Secret data as files in a volume mounted on a Pod's container(s)

- In Kubernetes, Volumes are Objects attached to Pods, here below is a Pod manifest with the Kubernetes Secret data you created as files in a volume mounted on the Pod's containers.

Create a **secret-test-volume-pod.yaml** and paste the configuration in it.

secret-test-volume-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-test-volume-pod
spec:
  containers:
  - name: secret-test-container
    image: nginx
    volumeMounts:
    - name: secret-volume
      mountPath: /etc/config/secret
  volumes:
  - name: secret-volume
    secret:
      secretName: database-credentials
```

--

- Execute the below kubectl commands to create pod and and view the secret details attached to the Pod.

```
kubectl apply -f secret-test-volume-pod.yaml

kubectl get pods

kubectl describe pod secret-test-volume-pod
# OUTPUT
    Mounts:
      /etc/config/secret from secret-volume (rw)
Volumes:
  secret-volume:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  database-credentials
    Optional:    false

# Running individual commands in a container
# To verify that the Pod can access the Secret data, connect to the container and
run the following commands in the volume directory:
kubectl exec secret-test-volume-pod -- printenv
kubectl exec secret-test-volume-pod -- ls -ltr /etc/config/secret
kubectl exec secret-test-volume-pod -- cat /etc/config/secret/username
kubectl exec secret-test-volume-pod -- cat /etc/config/secret/password

# Login inside the pod
kubectl exec -it secret-test-volume-pod bash
```

Kubernetes Interactive Steps

- Sign in with Docker Hub ID in : play-with-k8s
- yaml-basics-and-usage-in-kubernetes

Reference

- Interactive Steps - Exposing Your App
- Configure Pods and Containers
- Read The Twelve-Factor App to understand the motivation for separating code from configuration.