11 Configure Secrets in applications

app.py

```
@app.route("/")
def main():
    mysql.connector.connect(host='mysql', database='mysql',
                            user='root', password='password')
```

⇒ Consider a python file (app.py) that wants to connect to database (MySql)

On success, the app displays a successful message

If you look closely, you will see the hostname, username & password post coded

⇒ This is not a good idea, as learned earlier there is an option to move these values into a ConfigMap

The configuration map stores the configuration data in plain text format. So while it would be okay to move the hostname & username in configmap, but it is not the right place to store the password. This is where secrets come-in

config-map.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  DB_Host: mysql
  DB_User: root
  DB_Password: password
```

Secret

=> Secrets are used to store sensitive information like passwords or keys

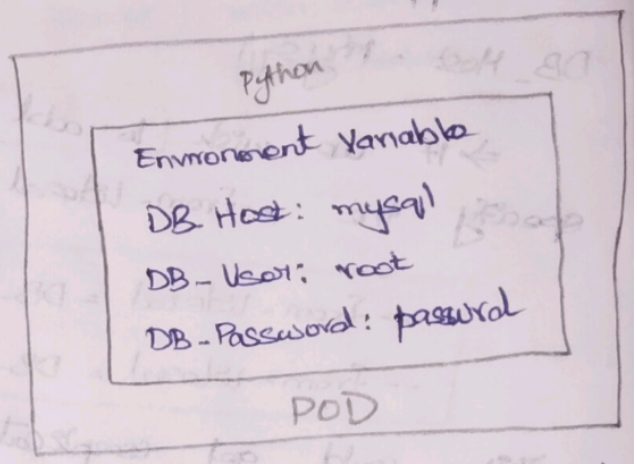
=> They're similar to ConfigMaps except that they're stored in an encoded format

Secret

DB_Host = bXlzcWw=

DB_User = Cm9vdA==

DB_Password = cGFzd3Jk



=> As with ConfigMaps, there are 2 steps involved in working with secrets

1) Create the secret
2) Inject it into pod

Secrets can be created in 2 ways

1) Imperative 2) Declarative

Imperative

specify the key value pairs in the command line itself

To create a secret of given values

Syntax:

```
kubectl create secret generic
    <secret-name> --from-literal=<key>=<value>
```

--from-literal option used to specify the key, value pairs in the command itself

Eg:-

```
Kubectl create secret generic \
    app-secret --from-literal = DB_Host = mysql
```

⇒ In this eg:- we're creating a secret by the name "appsecret", with the key value pair DB_Host = MySql

⇒ If we wish to add additional key value pair specify the --from-literal option multiple times.

```
--from-literal = DB_User = root
--from-literal = DB_Password = password
```

This could get complicated when we have too many secrets to pass in

To input the secret data through file

```
Kubectl create secret generic
    <secret-name> --from-file = <path-to-file>
```

```
Kubectl create secret generic \
    app-secret --from-file = app_secret.properties
```

use the --from-file option to specify a path to the file that contains the required data

This data from this file is read & stored under the name of the file app-secret

Declarative approach :-

⇒ Secrets used to store 'sensitive data' & are stored in an encoded format

⇒ We should not specify the data in plain text as this is not safe.

⇒ So while creating a secret with declarative approach, we must specify the secret values in an encoded format

secret-data.yaml

```
apiVersion : v1
kind: Secret
metadata:
    name: app-secret
data:
    DB_Host = bXlzcWw=
    DB_User = cm9vdA==
    DB_Password = cGFzd3Jk
```

`kubectl create -f secret-data.yaml`

How to Convert the plain text to encoded format on linux host

```
>echo -n 'mysql' | base64
bXlzcWw

>echo -n 'root' | base64
cm9vdA==

>echo -n 'paswrd' | base64
cZFzd3Jk
```

View Secrets

```
kubectl get secrets
```

The above command lists the newly created secret along with another secret previously created for its internal purposes.

To view more information on the newly created secret

```
kubectl describe secrets
```

This shows the attributes in the secret but hides the value themselves

To view the value as well with output displayed in YAML format

```
kubectl get secret app-secret -o yaml
```

Now, you can see the encoded values as well

How to decode encoded value

> echo -n 'bXlzcHw' | base64 --decode
mysql

> echo -n 'cm9vdA==' | base 64 - decode
root

> echo -n 'cGFzd3Jk' | base 64 - decode
pasword

Step 2:-

Configuring it with pod

4:20

⇒ To inject an environment variable add a new property to the container called ENV from

⇒ The envFrom property is a list, so we can pass as many environment variables as required.

⇒ Each item in the list corresponds to a secret item specify the name of the secret we created earlier

⇒ Creating the pod-definition-file now makes the data in the secret available as environment variable for the application.

pod-definition.yaml

```
apiVersion : V1
kind: Pod
metadata:
    name: simple-webapp-color
    labels:
      name: simple-webapp-color
spec:
    Containers:
    - name: simple-webapp-color
    image: simple-webapp-color
    ports:
      - containerPort: 8080
    envFrom:
      - secretRef:
          name: app-secret
```

secret-data.yaml

```
apiVersion: V1
kind: secret
metadata:
    name: app-secret
data:
    DB_Host: bXlzcWw=
    DB_User: cm9vdA==
    DB_Password: cGFzd3Jk
```

What we just saw was injecting ~~pods~~ secrets as environment variables into the pods

Other ways to inject secrets into pods

* We can inject as single environment Variables (or)
* Inject the whole secret as files in a volume

ENV

```
envFrom:
  - secretRef:
      name: app-config
```

SINGLE ENV

```
env:
  - name: DB_Password
    ValueFrom:
      secretkeyRef:
        name: app-secret
        key: DB_Password
```

VOLUME

```
Volumes:
  - name: app-secret-Volume
    Secret:
      secretName: app-secret
```

Secrets in Pod as Volume

If you want to mount the secret as a volume in the pod, Each attribute in the secret is Created as a file with the value of the secret as its content

In this case, Since we've 3 attributes in our secret 3 files are Created, & if we look at the contents of the DB password file, we see the password in it.

on to those pods,

So you should consider configuring role-based access control to restrict access

* Configure least-privilege access to Secrets - RBAC

⇒ Consider third-party secrets store providers
AWS Provider, Azure Provider, GCP provider, Vault provider.

This way, the secrets are stored not in etcd but in external secret providers and that those providers takes care of most of the security