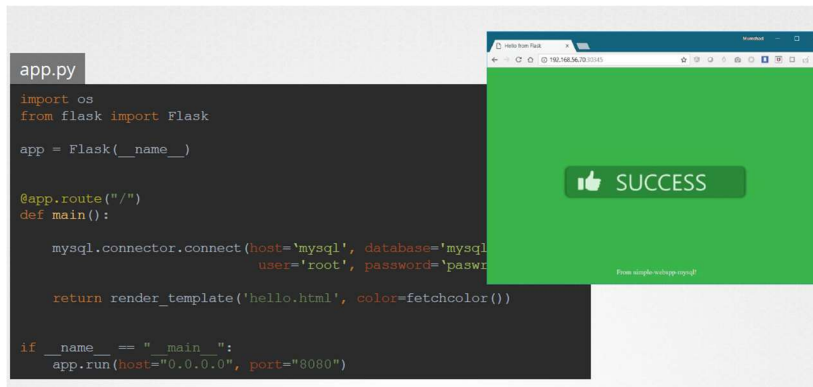
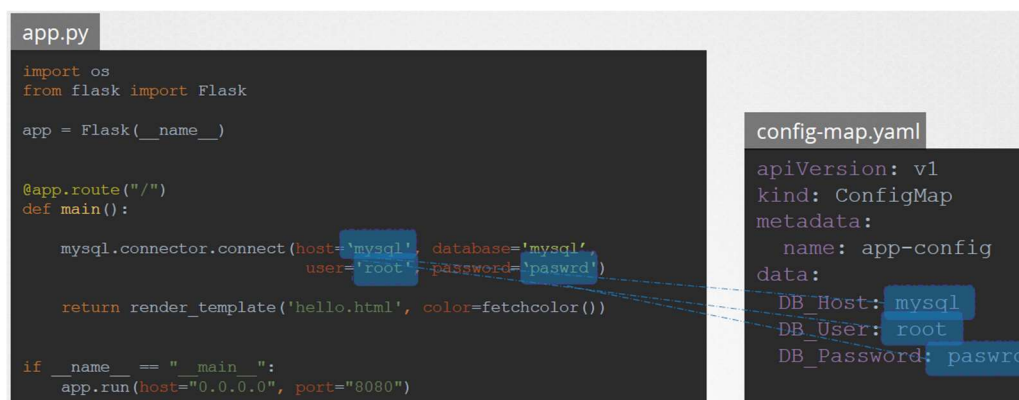


## Configure Secrets in Applications

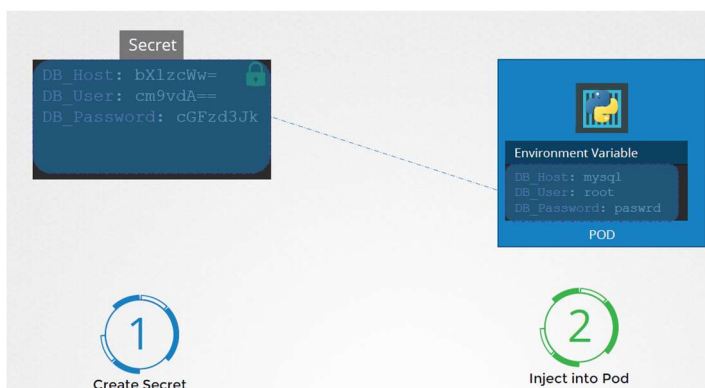
In this lecture, we discuss secrets in Kubernetes. Here we have a simple Python web application that connects to a MySQL database. On success, the application displays a successful message. If you look closely into the code, you will see the hostname, username, and password hardcoded. This is, of course, not a good idea.



As we learned in the previous lecture, one option would be to move these values into a ConfigMap. The ConfigMap stores configuration data in plain text format. So while it would be okay to move the hostname and username into a ConfigMap, it is definitely not the right place to store a password. This is where secrets come in.



**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. As with ConfigMaps, there are two steps involved in working with secrets. First, create the secret, and second, inject it into the pod.

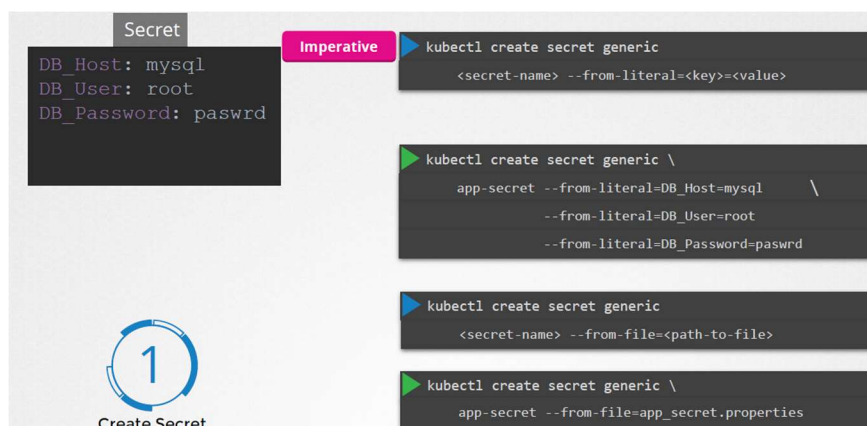


There are two ways of creating a secret, the imperative way, without using a secret definition file, and the declarative way, by using a secret definition file.

With the imperative method, you can directly specify the key-value pairs on the command line itself. To create a secret with the given values, run the `kubectl create secret generic` command. The command is followed by the secret name and the option `--from-literal`. The `--from-literal` option is used to specify the key-value pairs in the command itself.

In this example, we are creating a secret by the name `app-secret` with a key-value pair `DB_host=MySQL`. If you wish to add additional key-value pairs, simply specify the `--from-literal` option multiple times. However, this could get complicated when you have too many secrets to pass in.

Another way to input the secret data is through a file. Use the `--from-file` option to specify a path to the file that contains the required data. The data from this file is read and stored under the name of the file.

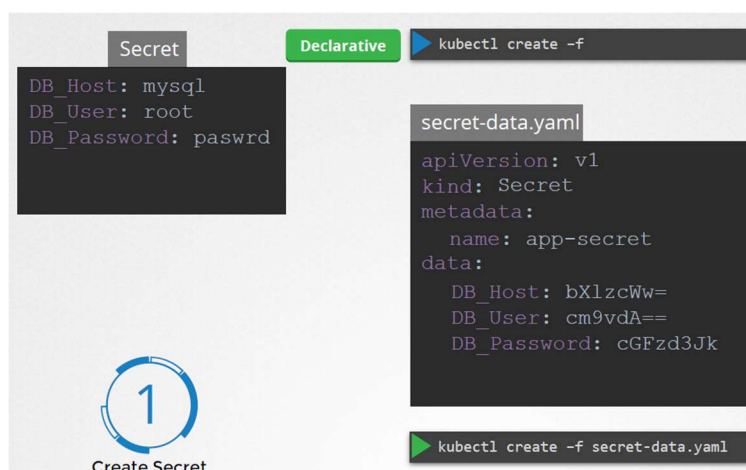


The image illustrates the imperative method for creating a secret. On the left, a 'Secret' box contains the following key-value pairs: `DB_Host: mysql`, `DB_User: root`, and `DB_Password: paswr`. Below this box is a circular icon with the number '1' and the text 'Create Secret'. To the right, under the 'Imperative' heading, four terminal commands are shown:

```
kubectl create secret generic  
<secret-name> --from-literal=<key>=<value>  
  
kubectl create secret generic \  
  app-secret --from-literal=DB_Host=mysql \  
  --from-literal=DB_User=root \  
  --from-literal=DB_Password=paswr  
  
kubectl create secret generic  
<secret-name> --from-file=<path-to-file>  
  
kubectl create secret generic \  
  app-secret --from-file=app_secret.properties
```

Let us now look at the declarative approach. For this, we create a definition file, just like how we did for the ConfigMap. The file has `apiVersion`, `kind`, `metadata`, and `data`. The `apiVersion` is `V1`, `kind` is `secret`. Under `metadata`, specify the name of the secret. We will call it `app-secret`. Under `data`, add the secret data in a key-value format.

However, one thing we discussed about secrets was that they're used to store sensitive data and are stored in an encoded format. Here we have specified the data in plain text which is not very safe. So while creating a secret with a declarative approach, you must specify the secret values in an encoded format. You must specify the data in an encoded form like this



The image illustrates the declarative method for creating a secret. On the left, a 'Secret' box contains the following key-value pairs: `DB_Host: mysql`, `DB_User: root`, and `DB_Password: paswr`. Below this box is a circular icon with the number '1' and the text 'Create Secret'. To the right, under the 'Declarative' heading, a terminal command is shown:

```
kubectl create -f
```

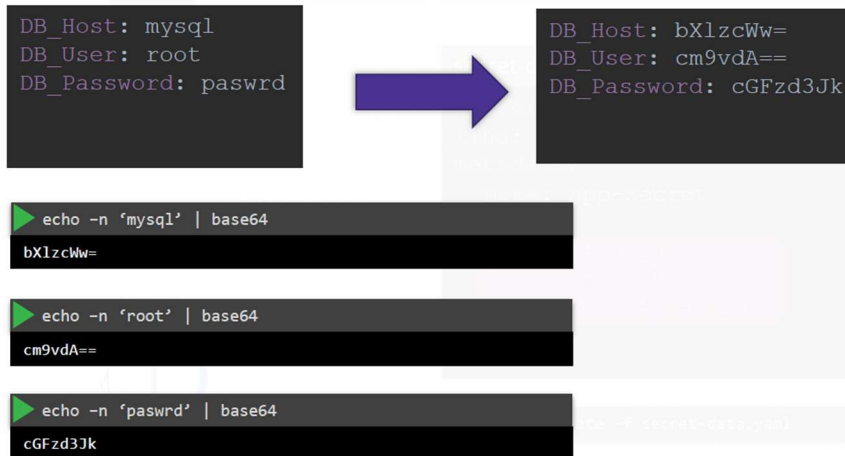
Below the command, a file named `secret-data.yaml` is shown with the following content:

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

Below the file content, another terminal command is shown:

```
kubectl create -f secret-data.yaml
```

But how do you convert the data from plain text to an encoded format? On a Linux host, run the command `echo -n`, followed by the text you're trying to convert, which is MySQL in this case. And pipe that to the `base64` utility.



To view secrets, run the `kubectl get secrets` command. This lists the newly created secret along with another secret previously created by Kubernetes for its internal purposes.

To view more information on the newly created secret, run the `kubectl describe secret` command. This shows the attributes in the secret but hides the values themselves.

To view the values as well, run the `kubectl get secret` command with the output displayed in a YAML format, using the `-o yaml` option. You can now see the hand-coded values as well.

```
kubectl get secrets
```

NAME	TYPE	DATA	AGE
app-secret	Opaque	3	10m
default-token-mvtkv	kubernetes.io/service-account-token	3	2h

```
kubectl describe secrets
```

Name: app-secret  
Namespace: default  
Labels: <none>  
Annotations: <none>  
  
Type: Opaque  
  
Data  
====  
DB\_Host: 10 bytes  
DB\_Password: 6 bytes  
DB\_User: 4 bytes

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

```
apiVersion: v1
data:
  DB_Host: bXlzcWw=
  DB_Password: cGFzd3Jk
  DB_User: cm9vdA==
kind: Secret
metadata:
  creationTimestamp: 2018-10-18T10:01:12Z
  labels:
    name: app-secret
  name: app-secret
  namespace: default
  uid: be96e989-d2bc-11e8-a545-080027931072
type: Opaque
```

Now, how do you decode encoded values? Use the same base 64 command used earlier to encode it but this time add a `decode` option to it.

```
DB_Host: mysql
DB_User: root
DB_Password: paswrđ
```

```
DB_Host: bXlzcWw=
DB_User: cm9vdA==
DB_Password: cGFzd3Jk
```



```
➤ echo -n 'bXlzcWw=' | base64 --decode
mysql
```

```
➤ echo -n 'cm9vdA==' | base64 --decode
root
```

```
➤ echo -n 'cGFzd3Jk' | base64 --decode
paswrđ
```

Now that we have a secret created, let us proceed with step two, configuring it with a pod. Here I have a simple pod definition file that runs my application.

To inject an environment variable, add a new property to the container called `envFrom`. 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 variables 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

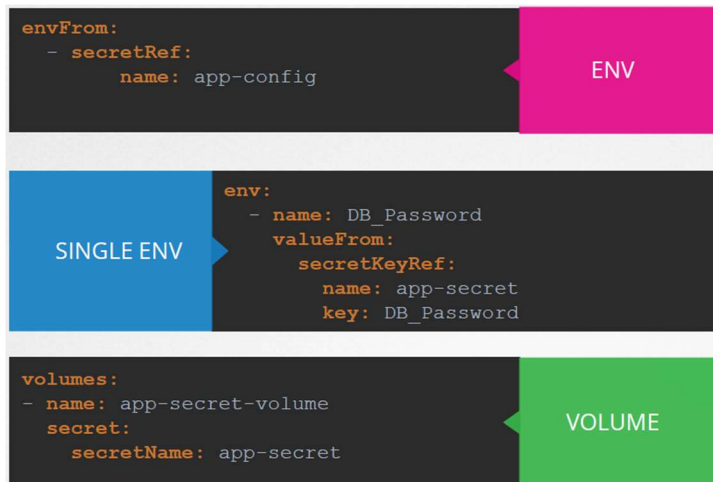
```
apiVersion: v1
kind: Secret
metadata:
  name: app-secret
data:
  DB_Host: bXlzcWw=
  DB_User: cm9vdA==
  DB_Password: cGFzd3Jk
```

```
➤ kubectl create -f pod-definition.yaml
```

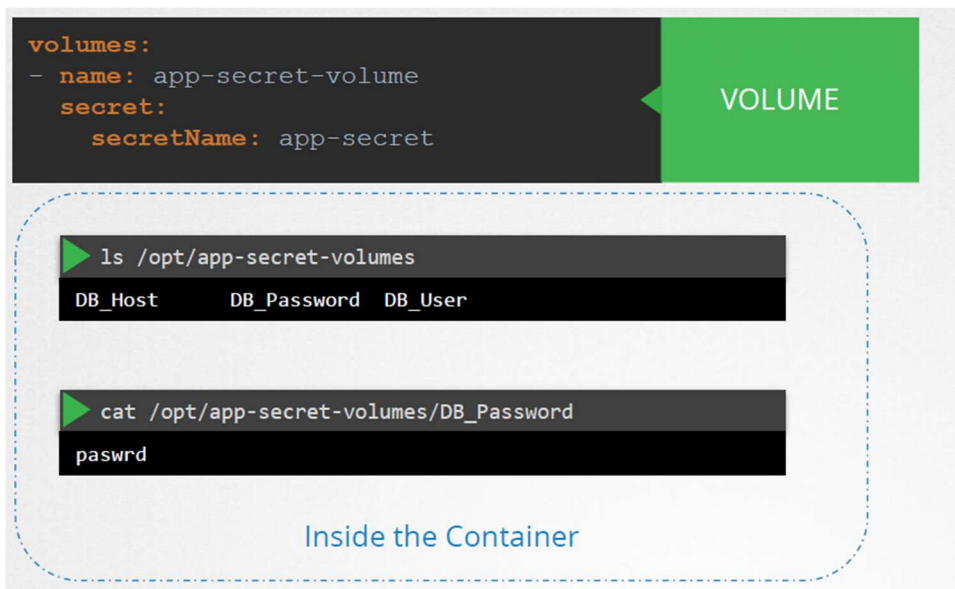


Inject into Pod

What we just saw was injecting secrets as environment variables into the pods. There are other ways to inject secrets into pods. You can inject a single environment variable or inject the whole secret as files in a volume.



If you were 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 have three attributes in our secret, three files are created. If we look at the contents of the DB password file, we see the password in it.



So here are some things to keep in mind when working with secrets. First of all, note that secrets are not encrypted. They're only encoded, meaning anyone can look up the file that you created for secrets or get the secret object and then decode it using the methods that we discussed before to see the confidential data. So remember not to check in your secret definition files along with your code when you push to GitHub or something.

There are lots of repositories already on GitHub where users have pushed their secret objects along with their code. The thing is you could easily get those secret objects, read them, and then just decode them using the base64 option that we just discussed, and you can get to see what the underlying passwords are. So keep that in mind.

Another note is that the secrets are not encrypted in etcd. None of the data in etcd is encrypted by default. So consider enabling encryption at rest. There is a document about it.

Also note that anyone able to create pods or deployments in the same namespace can access the secrets as well. Once you create a secret in a particular namespace, if anyone with access to creating a pod or deployment in the same namespace just goes in and creates a pod or deployment and uses that same secret, then they're able to see the secret objects mounted onto those pods. You should consider configuring role-based access control to restrict access.

And finally, consider third-party secret providers, such as AWS provider or Azure provider or GCP provider or the vault provider. This way, the secrets are stored not in etcd but in an external secret provider, and those providers take care of most of the security.

- ✗ Secrets are not Encrypted. Only encoded.
  - ✗ Do not check-in Secret objects to SCM along with code.
- ✗ Secrets are not encrypted in ETCD
  - ✓ Enable encryption at rest
- ✗ Anyone able to create pods/deployments in the same namespace can access the secrets
  - ✓ Configure least-privilege access to Secrets - RBAC
- ✓ Consider third-party secrets store providers  
AWS Provider, Azure Provider, GCP Provider, Vault Provider