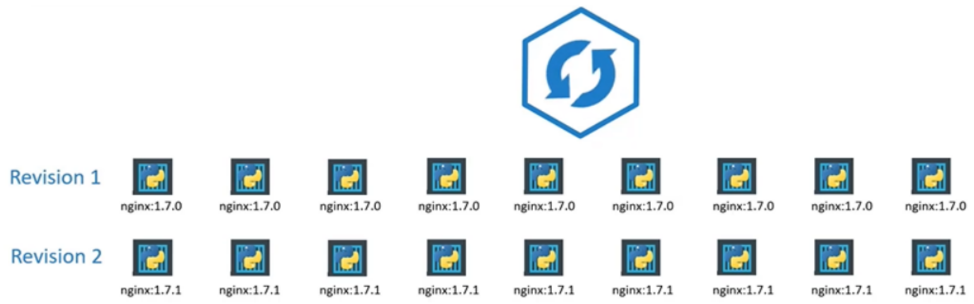
# Application Lifecycle Management

## Rolling Updates and Rollbacks

When you first create a deployment, it triggers a rollout. A new rollout creates a new deployment revision. Let's call it **Revision 1**.

In the future when the application is upgraded, meaning when the container version is updated to a new one, a new rollout is triggered and a new deployment revision is created named **Revision 2**.



This helps us keep track of the changes made to our deployment and enables us to roll back to a previous version of deployment if necessary.

- You can see the status of the rollout by the below command:

```
kubectl rollout status deployment/myapp-deployment
```

- To see the history and revisions:

```
kubectl rollout history deployment/myapp-deployment
```

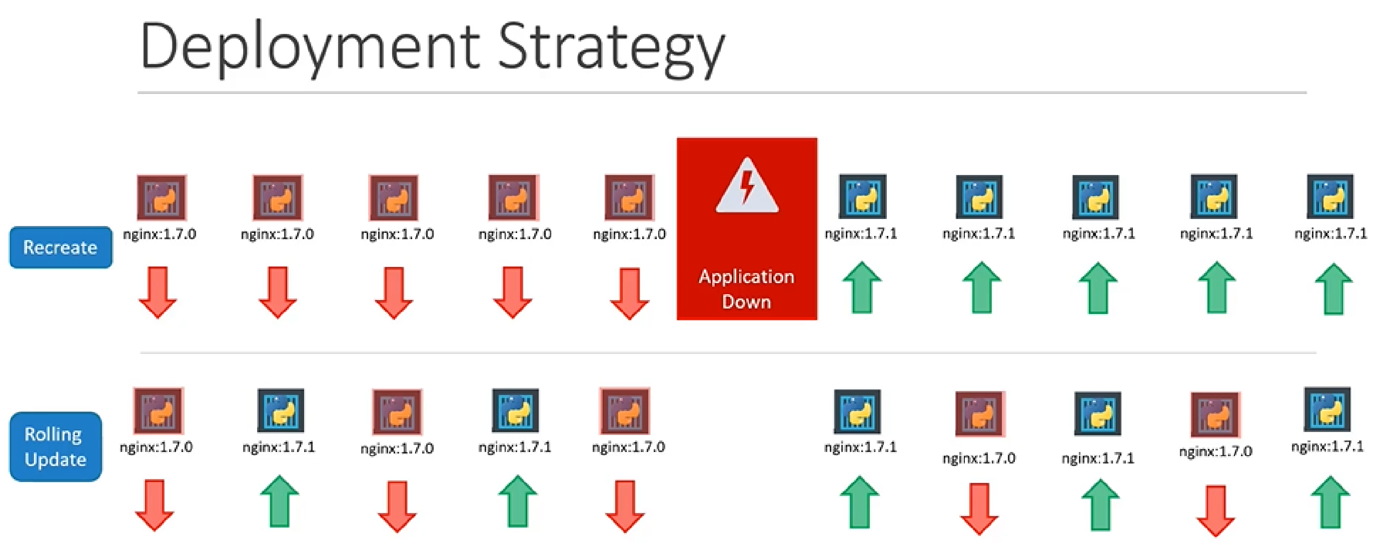
**There are two types of deployment strategies.**

Say, for example, you have five replicas of your web application instance deployed.

1. **Recreate:**
   One way to upgrade these to a newer version is to destroy all of these and then create newer versions of application instances, meaning first, destroy the five running instances and then deploy five new instances of the new application version.
   The problem with this, as you can imagine, is that during the period after the older versions are down and before any newer version is up, the application is down and inaccessible to users.
2. **RollingUpdate (Default Strategy):**

The second strategy is where we do not destroy all of them at once. Instead, we take down the older version and bring up a newer version one by one. This way the application never goes down and the upgrade is seamless.

Remember, if you do not specify a strategy while creating the deployment, it will assume it to be rolling-update. In other words, rolling update is the default deployment strategy.


Deployment Strategy

- To update a deployment, edit the deployment.yaml file and make necessary changes and save it. Then run the below command:

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: myapp-deployment
 labels:
  app: nginx
spec:
 template:
   metadata:
     name: myap-pod
     labels:
       app: myapp
       type: front-end
   spec:
    containers:
    - name: nginx-container
      image: nginx:1.7.1
 replicas: 3
 selector:
  matchLabels:
    type: front-end
```

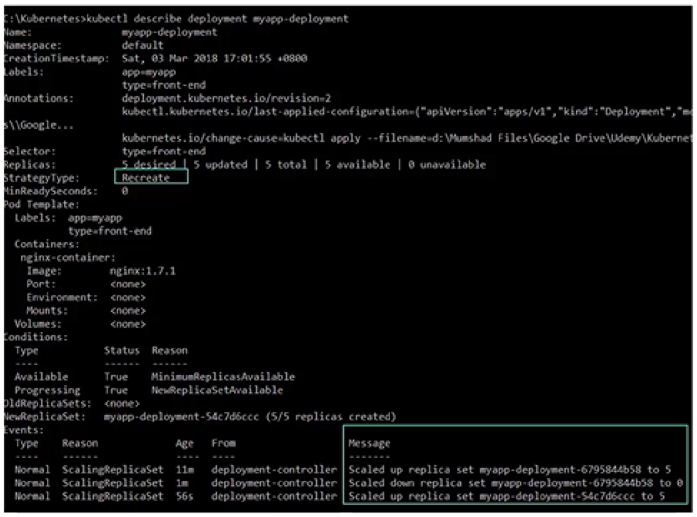
```
kubectl apply -f deployment-definition.yaml
```

- Alternate way to update a deployment say for example for updating an image:

```
kubectl set image deployment/myapp-deployment nginx=nginx:1.9.1
```

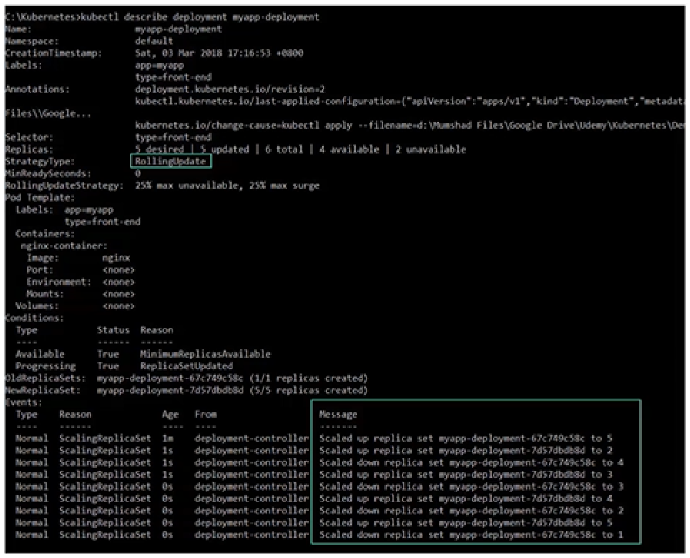
**Recreate vs RollingUpdate**

The difference between the recreate and rolling update strategies can also be seen when you view the deployments in detail.



```
:\Kubernetes>kubectl describe deployment myapp-deployment
Name:                   myapp-deployment
Namespace:              default
CreationTimestamp:      Sat, 03 Mar 2018 17:01:55 +0800
Labels:                 app=myapp
                        type=front-end
Annotations:            deployment.kubernetes.io/revision=2
                        kubectl.kubernetes.io/last-applied-configuration={"apiVersion":"apps/v1","kind":"Deployment","m
s\\Google...
                        kubernetes.io/change-cause=kubectl apply --filename=d:\Mumshad Files\Google Drive\Udemy\Kubernet
Selector:               type=front-end
Replicas:               5 desired | 5 updated | 5 total | 5 available | 0 unavailable
StrategyType:           Recreate
MinReadySeconds:        0
Pod Template:
  Labels:  app=myapp
           type=front-end
  Containers:
   nginx-container:
    Image:        nginx:1.7.1
    Port:         <none>
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Available      True    MinimumReplicasAvailable
  Progressing    True    NewReplicaSetAvailable
OldReplicaSets:  <none>
NewReplicaSet:   myapp-deployment-54c7d6ccc (5/5 replicas created)
Events:
  Type    Reason             Age   From                   Message
  ----    ------             ----  ----                   -------
  Normal  ScalingReplicaSet  11m   deployment-controller  Scaled up replica set myapp-deployment-6795844b58 to 5
  Normal  ScalingReplicaSet  1m    deployment-controller  Scaled down replica set myapp-deployment-6795844b58 to 0
  Normal  ScalingReplicaSet  56s   deployment-controller  Scaled up replica set myapp-deployment-54c7d6ccc to 5
```

Recreate



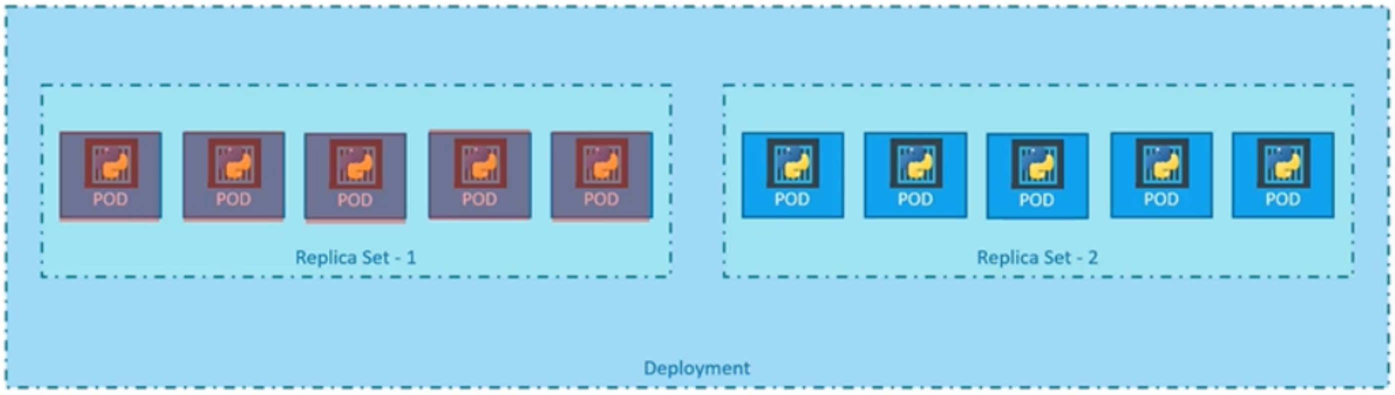
```
C:\Kubernetes>kubectl describe deployment myapp-deployment
Name:                   myapp-deployment
Namespace:              default
CreationTimestamp:      Sat, 03 Mar 2018 17:16:53 +0800
Labels:                 app=myapp
                        type=front-end
Annotations:            deployment.kubernetes.io/revision=2
                        kubectl.kubernetes.io/last-applied-configuration={"apiVersion":"apps/v1","kind":"Deployment","metadat
Files\\Google...
                        kubernetes.io/change-cause=kubectl apply --filename=d:\Mumshad Files\Google Drive\Udemy\Kubernetes\De
Selector:               type=front-end
Replicas:               5 desired | 5 updated | 6 total | 4 available | 2 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=myapp
           type=front-end
  Containers:
   nginx-container:
    Image:        nginx
    Port:         <none>
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Available      True    MinimumReplicasAvailable
  Progressing    True    ReplicaSetUpdated
OldReplicaSets:  myapp-deployment-67c749c58c (1/1 replicas created)
NewReplicaSet:   myapp-deployment-7d57dbdb8d (5/5 replicas created)
Events:
  Type    Reason             Age   From                   Message
  ----    ------             ----  ----                   -------
  Normal  ScalingReplicaSet  1m    deployment-controller  Scaled up replica set myapp-deployment-67c749c58c to 5
  Normal  ScalingReplicaSet  1s    deployment-controller  Scaled up replica set myapp-deployment-7d57dbdb8d to 2
  Normal  ScalingReplicaSet  1s    deployment-controller  Scaled down replica set myapp-deployment-67c749c58c to 4
  Normal  ScalingReplicaSet  1s    deployment-controller  Scaled up replica set myapp-deployment-7d57dbdb8d to 3
  Normal  ScalingReplicaSet  0s    deployment-controller  Scaled down replica set myapp-deployment-67c749c58c to 3
  Normal  ScalingReplicaSet  0s    deployment-controller  Scaled up replica set myapp-deployment-7d57dbdb8d to 4
  Normal  ScalingReplicaSet  0s    deployment-controller  Scaled down replica set myapp-deployment-67c749c58c to 2
  Normal  ScalingReplicaSet  0s    deployment-controller  Scaled up replica set myapp-deployment-7d57dbdb8d to 5
  Normal  ScalingReplicaSet  0s    deployment-controller  Scaled down replica set myapp-deployment-67c749c58c to 1
```

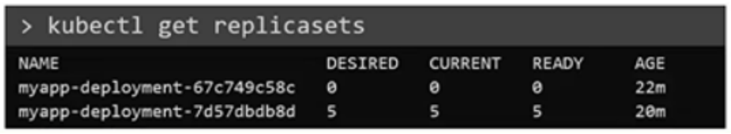
RollingUpdate

The events indicate that the old replica set was scaled down to 0/1, and then the new replica set scaled up to five. **However**, when the rolling update strategy was used, the old replica set was scaled down one at a time, at the same time scaling up the new replica set one at a time.

**Upgrades:**

When you upgrade your application, as we saw earlier, the Kubernetes deployment object creates a new replica set under the hood and starts deploying the containers there at the same time taking down the parts in the old replica set.





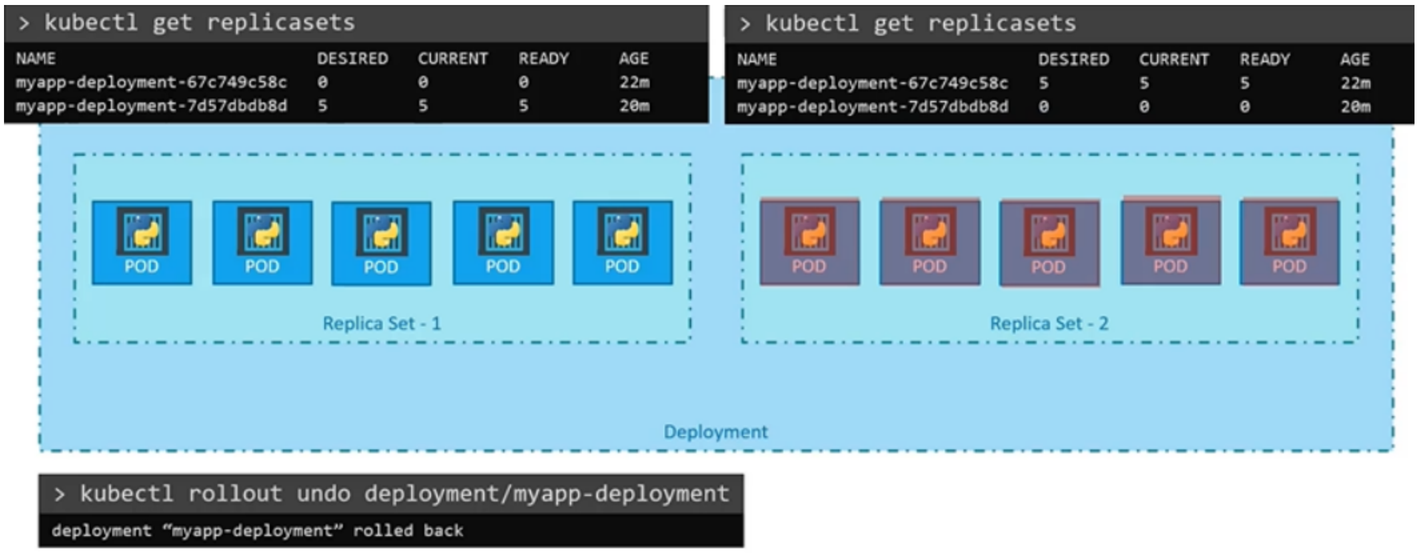
```
> kubectl get replicasets
NAME                          DESIRED   CURRENT   READY   AGE
myapp-deployment-67c749c58c   0         0         0       22m
myapp-deployment-7d57dbdb8d   5         5         5       20m
```

**Rollback:**

When you compare the output of the cube control, get replica sets, command before and after the rollback, you will be able to notice this difference.

Before the rollback.

The first replica set had zero parts and new replica set had five parts and this is reversed after the rollback is finished:



```
> kubectl get replicasets
NAME                          DESIRED   CURRENT   READY   AGE
myapp-deployment-67c749c58c   0         0         0       22m
myapp-deployment-7d57dbdb8d   5         5         5       20m
```

```
> kubectl get replicasets
NAME                          DESIRED   CURRENT   READY   AGE
myapp-deployment-67c749c58c   5         5         5       22m
myapp-deployment-7d57dbdb8d   0         0         0       20m
```

```
> kubectl rollout undo deployment/myapp-deployment
deployment "myapp-deployment" rolled back
```

- Kubernetes deployments allow you to roll back to a previous revision to undo a change, run the cube control rollout undo command followed by the name of the deployment:

```
kubectl rollout undo deployment/myapp-deployment
```

Summarize kubectl commands:

```
kubectl create -f deployment-definition.yaml
kubectl get deployments
kubectl apply -f deployment-definition.yaml
kubectl set image deployment/myapp-deployment nginx=nginx:1.9.1
kubectl rollout status deployment/myapp-deployment
kubectl rollout history deployment/myapp-deployment
kubectl rollout undo deployment/myapp-deployment
```

```
strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
```

---

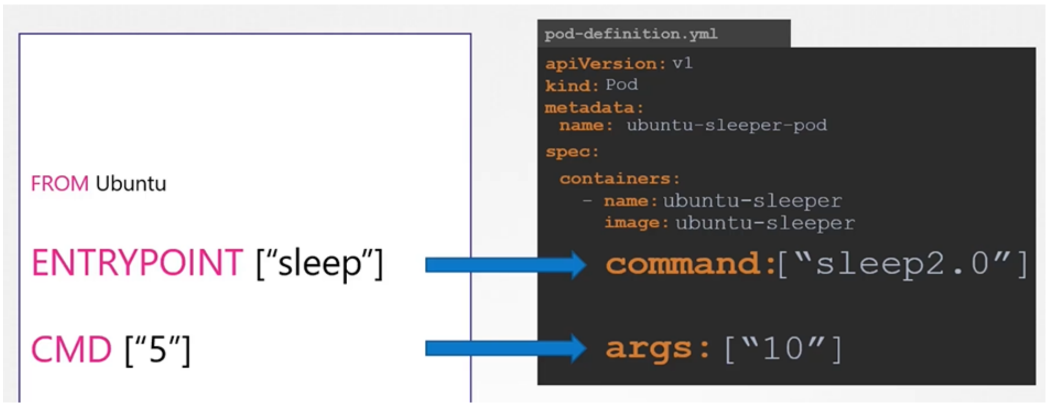
## Commands and Arguments

Let's imagine, we have a docker image that sleeps for a given number of seconds. We named it ubuntu-sleeper and we ran it using the docker command 'docker run ubuntu-sleeper'.

```
FROM: Ubuntu
ENTRYPOINT: ["sleep"]
CMD ["5"]
```

By default it sleeps for five seconds but you can override it by passing a command line argument. We will now create a pod using this image.

We start with a blank pod definition template, input the name of the pod and specify the image name. When the pod is created, it creates a container from the specified image, and the container sleeps for five seconds before exiting.


```
FROM Ubuntu

ENTRYPOINT ["sleep"]

CMD ["5"]
```

```
pod-definition.yml
apiVersion: v1
kind: Pod
metadata:
  name: ubuntu-sleeper-pod
spec:
  containers:
    - name: ubuntu-sleeper
      image: ubuntu-sleeper
      command:["sleep2.0"]

      args: ["10"]
```

Now if you need the container to sleep for 10 seconds as in the second command how do you specify the additional argument in the pod definition file. Anything that is appended to the docker run command will go into the “args” property of the pod definition file in the form of an array like this previoiusly picture.

```
apiVersion: v1
kind: Pod
metadata:
  name: ubuntu-sleeper-pod
spec:
 containers:
 - name: ubuntu-sleeper
   image: ubuntu-sleeper

   args: ["10"]
```

The Dockerfile has an Entrypoint as well as a CMD instruction specified. **The entrypoint is the command that is run at startup, and the CMD is the default parameter passed to the command.** With the args option in the pod definition file we override the CMD instruction in the Dockerfile.

**But what if you need to override the entrypoint?**

Say from sleep to a hypothetical sleep2.0 command?

In the docker world. We would run the docker run command with the **--entrypoint** option set to the new command.

```
docker run --name ubuntu-sleeper \
    --entrypoint sleep2.0 ubuntu-cleeper 10
```

The corresponding (*соответствующая) entry in the pod definition file would be using a **command** field. The command field - corresponds to entrypoint instruction in the docker file!

So to summarize there are two fields that correspond to two instructions in the docker file.

```
apiVersion: v1
kind: Pod
metadata:
  name: ubuntu-sleeper-pod
spec:
 containers:
 - name: ubuntu-sleeper
   image: ubuntu-sleeper
```

```
    command: ["sleep2.0"]

    args: ["10"]
```

LINKS:

[https://kubernetes.io/docs/tasks/inject-data-application/define-command-argument-container/](https://kubernetes.io/docs/tasks/inject-data-application/define-command-argument-container/)

---

## Configure Environment Variables in Applications

In this mini-article we will see how to set an environment variable in Kubernetes. Given a pod definition file which uses the same image as the docker command:

```
docker run -e APP_COLOR=pink simple-webapp-color
```

To set an environment variable, use the ENV property. ENV is an array. So every item under the env property starts with a dash, indicating an item in the array. Each item has a name and a value property.

```
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp-color
spec:
 containers:
 - name: simple-webapp-color
   image: simple-webapp-color
   ports:
   - containerPort: 8080
   env:
   - name: APP_COLOR
     value: pink
```

The name - is the name of the environment variable made available with the container.

The value - is its value of this name variable.

What we just saw was a direct way of specifying the environment variables using a plain key value pair format. **However**, there are other ways of setting the environment variables such as using config maps and secrets. The difference in this case is that instead of specifying value, we say valueFrom. And then a specification of configMap or secret. We will discuss about configMaps and secretKeys in the next part.

LINKS:

[https://kubernetes.io/docs/tasks/inject-data-application/define-environment-variable-container/](https://kubernetes.io/docs/tasks/inject-data-application/define-environment-variable-container/)

---

## Configuring ConfigMaps in Applications

In this section, we will take a look at configuring configmaps in applications

We saw how to define environment variables in it pod definition file. When you have a lot of pod definition files it will become difficult to manage the environment data stored within the query files.

We can take this information out of the pod definition file and manage it centrally using Configuration Maps.

**ConfigMaps are used to pass configuration data in the form of key value pairs in Kubernetes.** When it pod is created inject the config map into the pod. So the key value pairs that are available as environment variables for the application hosted inside the container in the pod.
There are two phases involved in configuring ConfigMaps. First create the ConfigMaps and second Inject them into the POD. Just like any other Kubernetes object.

- There are 2 phases involved in configuring ConfigMaps:
    1. First, create the configMaps
    2. Second, Inject then into the pod.
- There are 2 ways of creating a configmap.
    1. The Imperative way

```
kubectl create configmap app-config --from-literal=APP_COLOR=blue --from-literal=APP_MODE=prod
# OR
kubectl create configmap app-config --from-file=app_config.properties
```

1. The Declarative way

```
apiVersion: v1
kind: ConfigMap
metadata:
 name: app-config
data:
 APP_COLOR: blue
 APP_MODE: prod
```

Create a config map definition file and run the 'kubectl create` command to deploy it.

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

You can create as many configmaps as you need in the same way for various different purposes.

- To view configMaps:

```
kubectl get configmaps
(or)
kubectl get cm
```

- To describe configmaps:

```
kubectl describe configmaps
```

### Using ConfigMap in Pods

- Inject configmap in pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp-color
```

```
spec:
 containers:
 - name: simple-webapp-color
   image: simple-webapp-color
   ports:
   - containerPort: 8080
   envFrom:
   - configMapRef:
       name: app-config
```

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  APP_COLOR: blue
  APP_MODE: prod
```

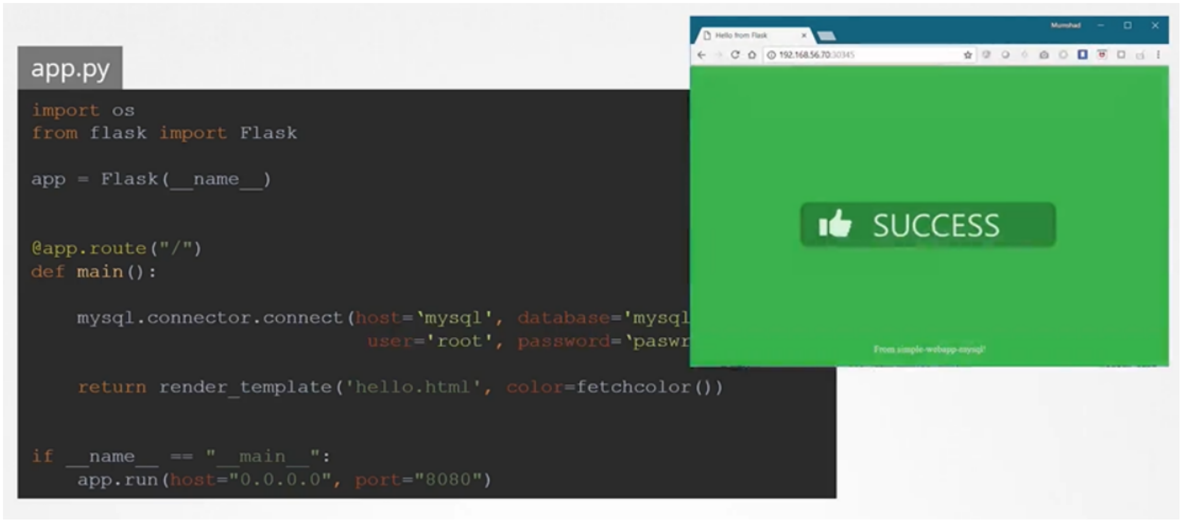
```
kubectl create -f pod-definition.yaml
```

LINKS:

- *https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/*
- *https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/#define-container-environment-variables-using-configmap-data*
- *https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/#create-configmaps-from-files*

---

## Configure Secrets in Applications


app.py

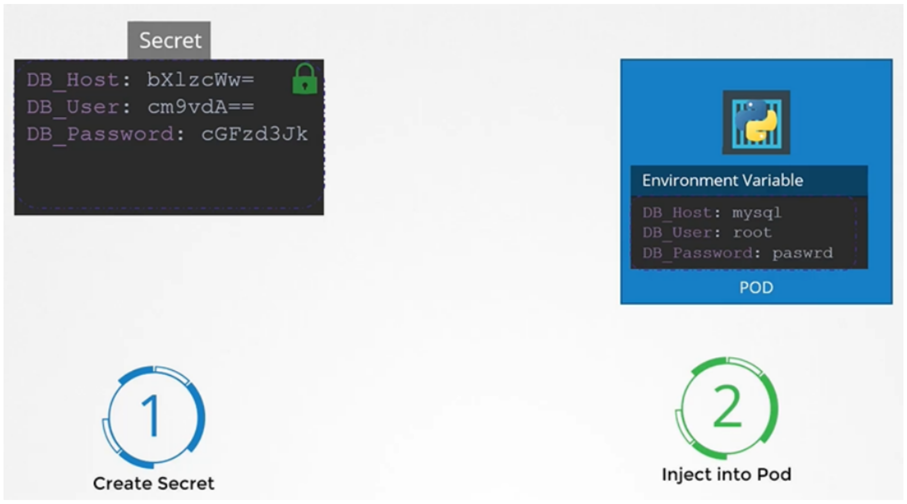
```
import os
from flask import Flask

app = Flask(__name__)


@app.route("/")
def main():

    mysql.connector.connect(host='mysql', database='mysql
                            user='root', password='paswr

    return render_template('hello.html', color=fetchcolor())


if __name__ == "__main__":
    app.run(host="0.0.0.0", port="8080")
```

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 part, 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 coming secrets are used to store sensitive information like passwords or keys. They're similar to configMap except that they're stored in an encoded or hashed format as with config maps.

**There are two steps involved in working with secrets:**

1. First, Create a secret
2. Second, Inject the secret into a pod.



**There are two ways of creating a secret:**

- The imperative way - without using a Secret definition file!
  You can directly specify the key value pairs in the command line itself to create a secret of the given values, run the kubectl create secret generic command:

```
kubectl create secret generic app-secret --from-literal=DB_Host=mysql --from-literal=DB_User=root --from-literal=DB_Password=paswrd
```

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.

If you wish to add additional key value pairs, simply specify the from literal options 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:

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

The data from this file is read and stored under the name of the file

- The Declarative way

  Create a secret definition file and run kubectl create to deploy it:

```
apiVersion: v1
kind: Secret
metadata:
 name: app-secret
data:
  DB_Host: bX1zcWw=
  DB_User: cm9vdA==
  DB_Password: cGFzd3Jk
```

**So while creating a secret with a declarative approach you must specify the secret values in a hashed format.**

But how do you convert the data from plain text to an encoded format?

  - On a linux host from the command `echo -n` followed by the text you are trying to convert, which is mysql in this case and pipe that to the base64 utility:

```
echo -n "mysql" | base64
bX1zcWw=

echo -n "root" | base64
cm9vdA==

echo -n "paswrd"| base64
cGFzd3Jk
```

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

**View Secrets**

- To view secrets:

```
kubectl get secrets
```

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:

```
kubectl describe secret
```

- To view the values of the secret:

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

Now how do you **decode** these hashed values?

- To decode secrets:

```
echo -n "bX1zcWw=" | base64 --decode
echo -n "cm9vdA==" | base64 --decode
echo -n "cGFzd3Jk" | base64 --decode
```

### Configuring secret with a pod

- To inject a secret to a pod add a new property **envFrom** followed by **secretRef** name and then create the pod-definition:

```
apiVersion: v1
kind: Secret
metadata:
 name: app-secret
data:
  DB_Host: bX1zcWw=
  DB_User: cm9vdA==
  DB_Password: cGFzd3Jk
```

```
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp-color
spec:
 containers:
 - name: simple-webapp-color
   image: simple-webapp-color
   ports:
   - containerPort: 8080
   envFrom:
   - secretRef:
       name: app-secret
```

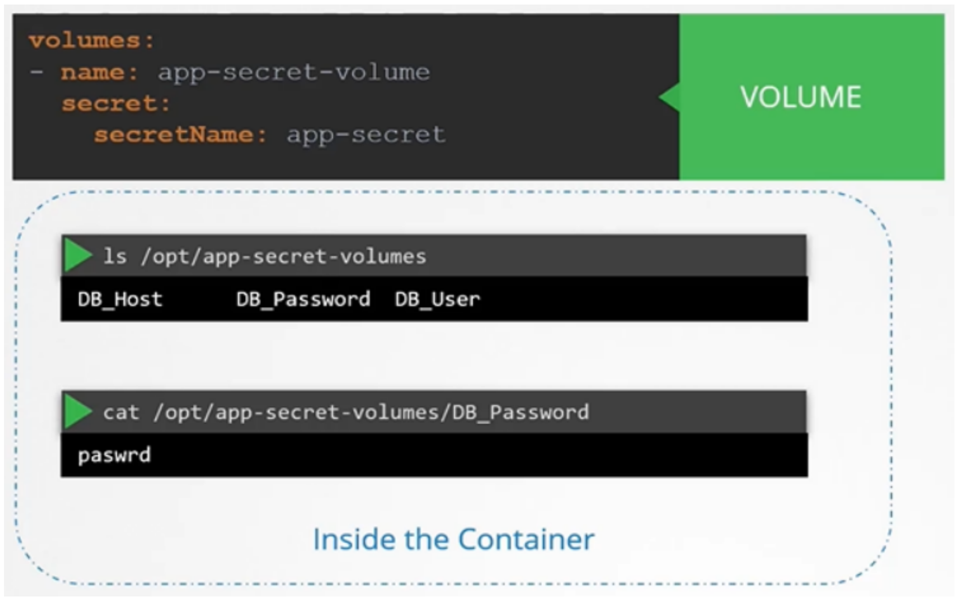
```
kubectl create -f pod-definition.yaml
```

### There are other ways to inject secrets into pods:

1. You can inject as Single ENV variable
2. You can inject as whole secret as files in a Volume

```
volumes:
- name: app-secret-volume
  secret:
    secretName: app-secret
```

Each attribute in the secret is created as a file with the value of the secret as its content.



```
volumes:
- name: app-secret-volume
  secret:
    secretName: app-secret
```

```
ls /opt/app-secret-volumes
DB_Host       DB_Password  DB_User
```

```
cat /opt/app-secret-volumes/DB_Password
paswrd
```

---

Remember that secrets encode data in base64 format. Anyone with the base64 encoded secret can easily decode it. As such the secrets can be considered as not very safe.
Having said that, there are other better ways of handling sensitive data like passwords in Kubernetes, such as using tools like Helm Secrets, HashiCorp Vault. I hope to make a lecture on these in the future.

---

LINKS:

- *https://kubernetes.io/docs/concepts/configuration/secret/*
- *https://kubernetes.io/docs/concepts/configuration/secret/#use-cases*
- *https://kubernetes.io/docs/tasks/inject-data-application/distribute-credentials-secure/*

---

## Multi Container PODs

At times you may need two services to work together such as a web server and a logging service.
You need one agent instance per web server instance paired together. You don't want to march and bloat the code of the two services as each of them target different functionalities and you'd still like them to be developed and deployed separately you only need the two functionality to work together.
You need one agent per web server instance paired together that can scale up and down together and that is why you have multi-container pods that share the same lifecycle which means **they are created together and destroyed together** they share the same network space which means they can refer to each other as local host and they have access to the same storage volumes.
This way you do not have to establish volume sharing or services between the pods. To enable communication between them to create a multi container pod. Add the new container information to the pod definition file. Remember the container section under the spec section in a pod definition file is an array. And the reason it is an array is to allow multiple containers in a single pod.

- To create a new multi-container pod, add the new container information to the pod definition file:

```
apiVersion: v1
```

```
kind: Pod
metadata:
  name: simple-webapp
  labels:
    name: simple-webapp
spec:
  containers:
  - name: simple-webapp
    image: simple-webapp
    ports:
    - ContainerPort: 8080
  - name: log-agent
    image: log-agent
```

LINKS:

- *https://kubernetes.io/docs/tasks/access-application-cluster/communicate-containers-same-pod-shared-volume/*

---

## InitContainers

In a multi-container pod, each container is expected to run a process that stays alive as long as the POD's lifecycle. For example in the multi-container pod that we talked about earlier that has a web application and logging agent, both the containers are expected to stay alive at all times. The process running in the log agent container is expected to stay alive as long as the web application is running. If any of them fails, the POD restarts.

But at times you may want to run a process that runs to completion in a container. For example a process that pulls a code or binary from a repository that will be used by the main web application. That is a task that will be run only one time when the pod is first created. Or a process that waits for an external service or database to be up before the actual application starts. That's where **initContainers** comes in.

An **initContainer** is configured in a pod like all other containers, except that it is specified inside a `initContainers` section, like this:

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox:1.28
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
  initContainers:
  - name: init-myservice
```

```
    command: ['sh', '-c', 'git clone <some-repository-that-will-be-used-by-application> ; done;']
```

When a POD is first created the initContainer is run, and the process in the initContainer must run to a completion before the real container hosting the application starts.

You can configure multiple such initContainers as well, like how we did for multi-pod containers. In that case each init container is run one at a time in sequential order.

If any of the initContainers fail to complete, Kubernetes restarts the Pod repeatedly until the Init Container succeeds:

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox:1.28
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
  initContainers:
  - name: init-myservice
    image: busybox:1.28
    command: ['sh', '-c', 'until nslookup myservice; do echo waiting for myservice; sleep 2; done;']
  - name: init-mydb
    image: busybox:1.28
    command: ['sh', '-c', 'until nslookup mydb; do echo waiting for mydb; sleep 2; done;']
```

LINKS:

*https://kubernetes.io/docs/concepts/workloads/pods/init-containers/*

---

## Self Healing Applications

Kubernetes supports self-healing applications through ReplicaSets and Replication Controllers. The replication controller helps in ensuring that a POD is re-created automatically when the application within the POD crashes. It helps in ensuring enough replicas of the application are running at all times.

Kubernetes provides additional support to check the health of applications running within PODs and take necessary actions through Liveness and Readiness Probes.

---

## END of the Application Lifecycle Management section.