2. Workloads & scheduling - 15 %

1. Understanding deployments and performing Rolling updates:

WHY Deployment ?:

- Scaling(ReplicaSet)
- Rollout & RollBack

Feature:

- *Application Deployment* => ROLLOUT | PAUSE & RESUME | ROLLBACK
- *Scaling* => REPLICAS | SCALE DOWN & UP

Deployment Strategy Types:

- ***RECREATE*** => with this type we completely turn off existing versions and deploy new versions(down time).
- ***ROLLING UPDATE*** => K8s will rollout the newer version by removing the older version one by one (time consuming) default strategy
- ***CANARY*** => to test a newer version of the application completely before it is deployed (can rollout if any issue in newer version) .
- ***BLUE / GREEN*** => we deploy the same number of old and new versions of the app. Less impact if any issue with the new version and we can remove the old version if the new version works fine. expensive.

Note: `kubectl describe deployment <name> | grep StrategyType`

A. Create a deployment that creates a ReplicaSet to bring up three `nginx` Pods:

```
Kubectl create deployment nginx-deployment –image=nginx –replicas=3 -o yaml –dry-run=client > nginx-deploy.yaml
```

To see Replica Sets :

```
Kubectl get rs
```

To see the labels automatically generated for each Pod. The output is similar to:

```
kubectl get pods --show-labels
```

```
NAME                                READY     STATUS    RESTARTS   AGE       LABELS
nginx-deployment-75675f5897-7ci7o   1/1       Running   0          18s       app=nginx,pod-template-hash=3123191453
nginx-deployment-75675f5897-kzszj   1/1       Running   0          18s       app=nginx,pod-template-hash=3123191453
nginx-deployment-75675f5897-qqcnn   1/1       Running   0          18s       app=nginx,pod-template-hash=3123191453
```

Note:- In case maxUnavailable is not mentioned , by default it takes 35% replicas as unavailable during the rolling out . Always use **- - record** along with deployment, this will allow commands stored in history.

B. Updating a Deployment:

Note: A Deployment's rollout is triggered if and only if the Deployment's Pod template (that is, `.spec.template`) is changed, for example if the labels or container images of the template are updated. Other updates, such as scaling the Deployment, do not trigger a rollout.

1. **Let's update the nginx Pods to use the `nginx:1.16.1` image instead of the `nginx:1.14.2` image.**
   `kubectl set image deployment/nginx-deployment nginx=nginx:1.16.1`

2. **To see the rollout status, run:**
   `kubectl rollout status deployment/nginx-deployment`
   - After the rollout succeeds, you can view the Deployment by running `kubectl get deployments`.
     The output is similar to this:

```
NAME               READY   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3/3     3            3           36
```

   - Run `kubectl get rs` to get the replicasets:

```
NAME                          DESIRED   CURRENT   READY   AGE
nginx-deployment-1564180365   3         3         3       6s
nginx-deployment-2035384211   0         0         0       36s
```

C. Rolling Back a Deployment:

Sometimes, you may want to rollback a Deployment; for example, when the Deployment is not stable, such as crash looping. By default, all of the Deployment's rollout history is kept in the system so that you can rollback anytime you want.

1. Checking Rollout History of a Deployment :
   a. First, check the revisions of this Deployment:

```
kubectl rollout history deployment/nginx-deployment
```

   The output is similar to this:

```
deployments "nginx-deployment"
REVISION    CHANGE-CAUSE
1           kubectl apply --filename=https://k8s.io/examples/controllers/nginx-deployment.yaml
2           kubectl set image deployment/nginx-deployment nginx=nginx:1.16.1
3           kubectl set image deployment/nginx-deployment nginx=nginx:1.161
```

   b. To see details of each version:
   `kubectl rollout history deployment/nginx-deployment --revision=2`

2. Rolling Back to a Previous Revision:

   Now you've decided to undo the current rollout and rollback to the previous revision:
   `kubectl rollout undo deployment/nginx-deployment`

   Alternatively, you can rollback to a specific revision by specifying it with `--to-revision`:
   `kubectl rollout undo deployment/nginx-deployment --to-revision=2`

D. Scaling the deployment:

a. You can scale a Deployment by using the following command:

```
kubectl scale deployment/nginx-deployment --replicas=10
```

b. horizontal Pod autoscaling :

```
kubectl autoscale deployment/nginx-deployment --min=10 --max=15
--cpu-percent=80
```

E. Pausing and Resuming a rollout of a Deployment:

When you update a Deployment, or plan to, you can pause rollouts for that Deployment before you trigger one or more updates. When you're ready to apply those changes, you resume rollouts for the Deployment. This approach allows you to apply multiple fixes in between pausing and resuming without triggering unnecessary rollouts.

Pause by running the following command:]=-.

```
kubectl rollout pause deployment/nginx-deployment
```

Then update the image of the Deployment:

```
kubectl set image deployment/nginx-deployment nginx=nginx:1.16.3
```

Notice that no new rollout started:

```
kubectl rollout history deployment/nginx-deployment
```

You can make as many updates as you wish, for example, update the resources that will be used:

```
kubectl set resources deployment/nginx-deployment -c=nginx
--limits=cpu=200m,memory=512Mi
```

Eventually, resume the Deployment rollout and observe a new ReplicaSet coming up with all the new updates:

```
kubectl rollout resume deployment/nginx-deployment
```

Get the status of the latest rollout:

```
kubectl get rs
```

F. Labels & selectors:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
```

```
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

As we can see labels are used at 2 places , 1 at object level and 1 inside templates. Labels at object level are used to identify the object eg, deployment, replicaSet and the labels used inside template are at pod level used to identify the pods which should match the labels mentioned at object level.

And Selectors is used to match the pods which mentioned labels, if labels match these it will create a pod as specified.

It works the same manner in service objects also. Service contains the selectors and it matches the labels and selectors in the deployment object.

To view the pods based on the selectors:

```
Kubectl get pods --selectors <key>=<value>
```

G. Taints & tolerations:

Tolerations are applied to pods, and allow (but do not require) the pods to schedule onto nodes with matching taints.

Taints and tolerations work together to ensure that pods are not scheduled onto inappropriate nodes. One or more taints are applied to a node; this marks that the node should not accept any pods that do not tolerate the taints.

I. Taints- Nodes:

You add a taint to a node using kubectl taint. For example,
Syntax: `kubectl taints nodes <node_name> key=value:taint-effect`

```
kubectl taint nodes node1 key1=value1:NoSchedule
```

places a taint on node `node1`. The taint has key `key1`, value `value1`, and taint effect `NoSchedule`. This means that no pod will be able to schedule onto `node1` unless it has a matching toleration.

To remove the taint added by the command above, you can run:

```
kubectl taint nodes node1 key1=value1:NoSchedule-
```

II. Toleration- Pods:

You specify a toleration for a pod in the PodSpec. Both of the following tolerations "match" the taint created by the `kubectl taint` line above, and thus a pod with either toleration would be able to schedule onto `node1`:

```
tolerations:
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoSchedule"
```

III. Taint - Effect:

NoExecute:

It will tell the nodes to only accept the pods that satisfy the particular condition but does not tell the pod to go to the particular node.

Alternatively, you can use the effect of `PreferNoSchedule`. This is a "preference" or "soft" version of `NoSchedule` -- the system will *try* to avoid placing a pod that does not tolerate the taint on the node, but it is not required.

You can put multiple taints on the same node and multiple tolerations on the same pod. The way Kubernetes processes multiple taints and tolerations is like a filter: start with all of a node's taints, then ignore the ones for which the pod has a matching toleration; the remaining un-ignored taints have the indicated effects on the pod. In particular,

- if there is at least one un-ignored taint with effect **`NoSchedule`** then Kubernetes will not schedule the pod onto that node
- if there is no un-ignored taint with effect **`NoSchedule`** but there is at least one un-ignored taint with effect **`PreferNoSchedule`** then Kubernetes will *try* to not schedule the pod onto the node
- if there is at least one un-ignored taint with effect **`NoExecute`** then the pod will be evicted from the node (if it is already running on the node), and will not be scheduled onto the node (if it is not yet running on the node

H. Node selectors & affinity:

```
//pod-spec.yaml
apiVersion:
Kind: pod
metadata:
     name: myapp-pod
spec:
     containers:
     -  name: data-processor
```

```
    Image: data-processor
nodeSelector:
    Size: large           # corresponds to the labels set on nodes
```

This will allow this pod to be placed on the node that contains the label's `size: large` .
To use such types of selectors , we should specify the labels on the nodes prior .

I. **Labeling Nodes:**
`Kubectl labels nodes <node_name> <key>=<value>`

What if we want to place the pod that matches the multiple conditions like medium, dev etc. we can use node affinity .

II. **Node affinity:**
Node affinity is conceptually similar to `nodeSelector` -- it allows you to constrain which nodes your pod is eligible to be scheduled on, based on labels on the node.

1. Add a label to a node

1. List the nodes in your cluster, along with their labels:
   `kubectl get nodes --show-labels`
2. Chose one of your nodes, and add a label to it:
   `kubectl label nodes <your-node-name> disktype=ssd`
3. Verify that your chosen node has a `disktype=ssd` label:
   `kubectl get nodes --show-labels`

2. Schedule a Pod using required node affinity:

This manifest describes a Pod that has a `requiredDuringSchedulingIgnoredDuringExecution` node affinity,`disktype: ssd`. This means that the pod will get scheduled only on a node that has a `disktype=ssd` label.

```
spec:
 affinity:
   nodeAffinity:
     requiredDuringSchedulingIgnoredDuringExecution:
       nodeSelectorTerms:
       - matchExpressions:
         - key: disktype
```

```
            operator: In
            values:
            - ssd
```

1. Apply the manifest to create a Pod that is scheduled onto your chosen node:

`kubectl apply -f https://k8s.io/examples/pods/pod-nginx-required-affinity.yaml`

2. Verify that the pod is running on your chosen node:

`kubectl get pods --output=wide`

3. Schedule a Pod using preferred node affinity:

This manifest describes a Pod that has a `preferredDuringSchedulingIgnoredDuringExecution` node affinity, `disktype: ssd`. This means that the pod will prefer a node that has a `disktype=ssd` label.

```
spec:
  affinity:
    nodeAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 1
        preference:
          matchExpressions:
          - key: disktype
            operator: In
            values:
            - ssd
```

1. Apply the manifest to create a Pod that is scheduled onto your chosen node:

`kubectl apply -f https://k8s.io/examples/pods/pod-nginx-preferred-affinity.yaml`

2. Verify that the pod is running on your chosen node:

`kubectl get pods --output=wide`

The new node affinity syntax supports the following operators: `In`, `NotIn`, `Exists`, `DoesNotExist`, `Gt`, `Lt`.

Example of `requiredDuringSchedulingIgnoredDuringExecution` would be "only run the pod on nodes with Intel CPUs" and an example `preferredDuringSchedulingIgnoredDuringExecution` would be "try to run this set of pods in failure zone XYZ, but if it's not possible, then allow some to run elsewhere".

I. Resources request and limit:

I. Requests and limits:

If the node where a Pod is running has enough of a resource available, it's possible (and allowed) for a container to use more resources than its `request` for that resource specifies. However, a container is not allowed to use more than its resource `limit`.

If you set a `memory` limit of 4GiB for that container, the kubelet (and container runtime) enforce the limit. The runtime prevents the container from using more than the configured resource limit. For example: when a process in the container tries to consume more than the allowed amount of memory, the system kernel terminates the process that attempted the allocation, with an out of memory (OOM) error.

II. Resource types:

CPU and *memory* are each a *resource type*. A resource type has a base unit. CPU represents compute processing and is specified in units of Kubernetes CPUs. Memory is specified in units of bytes.
Huge pages are a Linux-specific feature where the node kernel allocates blocks of memory that are much larger than the default page size.

III. Resource requests and limits of Pod and container:

For each container, you can specify resource limits and requests, including the following:

- `spec.containers[].resources.limits.cpu`
- `spec.containers[].resources.limits.memory`
- `spec.containers[].resources.limits.hugepages-<size>`
- `spec.containers[].resources.requests.cpu`
- `spec.containers[].resources.requests.memory`
- `spec.containers[].resources.requests.hugepages-<size>`

IV. Resource units in Kubernetes :

1. **CPU resource units:**

   1 CPU = 1 physical CPU core or 1 virtual core
   Expressing in Quantity 0.1 == 100 m (one hundred millicpu)

2. **Memory resource units :**

   Limits and requests for `memory` are measured in bytes.
   You can express memory as a plain integer or as a fixed-point number using one of these quantity suffixes: E, P, T, G, M, k.
   You can also use the power-of-two equivalents: Ei, Pi, Ti, Gi, Mi, Ki. For example, the following represent roughly the same value:

   ```
   128974848, 129e6, 129M,  128974848000m, 123Mi
   ```

   Take care about the case for suffixes. If you request `400m` of memory, this is a request for 0.4 bytes

V. Container resources example:

The following Pod has two containers. Both containers are defined with a request for 0.25 CPU and 64MiB (226 bytes) of memory. Each container has a limit of 0.5 CPU and 128MiB of memory. You can say the Pod has a request of 0.5 CPU and 128 MiB of memory, and a limit of 1 CPU and 256MiB of memory.

```
---
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: app
    image: images.my-company.example/app:v4
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
  - name: log-aggregator
    image: images.my-company.example/log-aggregator:v6
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

VI. Default resources request & limit

"When a pod is created the containers are assigned a default CPU request of .5 and memory of 256Mi". For the POD to pick up those defaults you must have first set those as default values for request and limit by creating a LimitRange in that namespace.

```
apiVersion: v1
kind: LimitRange
metadata:
 name: mem-limit-range
spec:
 limits:
 - default:
     memory: 512Mi
   defaultRequest:
     memory: 256Mi
   type: Container
```

https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/memory-default-namespace/

```
apiVersion: v1
```

```
kind: LimitRange
metadata:
 name: cpu-limit-range
spec:
 limits:
 - default:
     cpu: 1
   defaultRequest:
     cpu: 0.5
   type: Container
```

https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/cpu-default-namespace

J. DaemonSet:

A *DaemonSet* ensures that all (or some) Nodes run a copy of a Pod. As nodes are added to the cluster, Pods are added to them. As nodes are removed from the cluster, those Pods are garbage collected. Deleting a DaemonSet will clean up the Pods it created.

Some typical uses of a DaemonSet are:

- running a cluster storage daemon on every node
- running a logs collection daemon on every node
- running a node monitoring daemon on every node

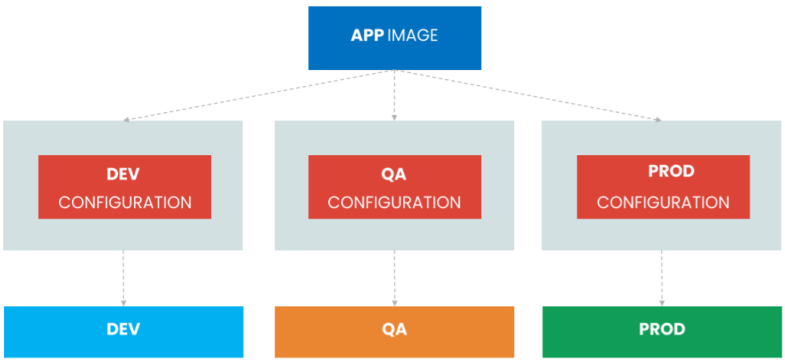
UseCases:

Monitoring solutions or log viewers.
Kube-proxy- deployed as a daemonSet
Networking- weaveNet

2. Use configmaps and secrets to configure applications:

A. ConfigMaps

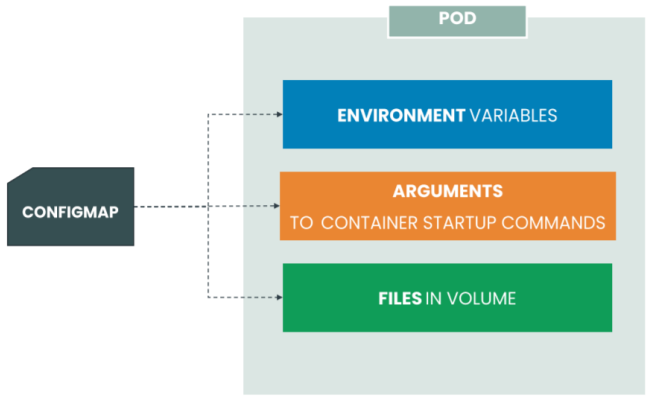
WHY ConfigMaps?:



WHAT is ConfigMaps:

Object to store non-sensitive data .
Can use single configmaps in multiple pods.

USED FOR:



I. Create a ConfigMaps:

```
kubectl create configmap <map-name> <data-source>
```

```
cat special-config.yaml

apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
data:
  ENV_ONE: “va1ue1”
  ENV_TWO: “va1ue2”
```

```
cat nginx-config.yaml

apiVersion: v1
kind: ConfigMap
metadata:
  name: my-nginx-config
data:
  my-nginx-config.conf: |-
    server {
      listen 80;
      server_name www.kubia-example.com;
      gzip on;
      gzip_types text/plain application/xml;
      location / {
        root /usr/share/nginx/html;
        index index.html index.htm;
      }
    }
    sleep-interval: 25
```

File Name - KEY

File Content - VALUE

Diff ways of creating configMaps:

a. Using directories:

```
# Create the local directory
mkdir -p configure-pod-container/configmap/

# Download the sample files into `configure-pod-container/configmap/`
directory
wget https://kubernetes.io/examples/configmap/game.properties -O
configure-pod-container/configmap/game.properties

# Create the configmap
kubectl create configmap game-config
--from-file=configure-pod-container/configmap/
```

// To see yaml file created out of above command

```
kubectl get configmaps game-config -o yaml
```

b. Using files:

```
kubectl create configmap game-config-2
--from-file=configure-pod-container/configmap/game.properties
```

// We can –from-env-file when we use <key>=<value> list in the file to refer

```
kubectl create configmap game-config-env-file \
--from-env-file=configure-pod-container/configmap/game-env-file.properties
```

c. Define the key to use when creating a ConfigMap from a file:
You can define a key other than the file name to use in the `data` section of your ConfigMap when using the `--from-file` argument:

```
kubectl create configmap game-config-3
--from-file=<my-key-name>=<path-to-file>
```

For example:

```
kubectl create configmap game-config-3
--from-file=game-special-key=configure-pod-container/configmap/game.properties
```

where the output is similar to this:

```
apiVersion: v1
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:54:22Z
  name: game-config-3
  namespace: default
  resourceVersion: "530"
  uid: 05f8da22-d671-11e5-8cd0-68f728db1985
data:
  game-special-key: |
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
```

d. from literal values

```
kubectl create configmap special-config --from-literal=special.how=very
--from-literal=special.type=charm
```

Define container environment variables using ConfigMap data:

a. Define the environment variables in the Pod specification.

```
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: SPECIAL_LEVEL_KEY
```

```
        valueFrom:
          configMapKeyRef:
            name: special-config
            key: special.how
      - name: LOG_LEVEL
        valueFrom:
          configMapKeyRef:
            name: env-config
            key: log_level
```

b. Configure all key-value pairs in a ConfigMap as container environment variables

```
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      command: [ "/bin/sh", "-c", "env" ]
      envFrom:
      - configMapRef:
          name: special-config
  restartPolicy: Never
```

c. Add ConfigMap data to a Volume

```
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      command: [ "/bin/sh", "-c", "ls /etc/config/" ]
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        # Provide the name of the ConfigMap containing the files you want
        # to add to the container
        name: special-config
  restartPolicy: Never
```

B. Secrets

A Secret is an object that contains a small amount of sensitive data such as a password, a token, or a key. Such information might otherwise be put in a Pod specification or in a container image. Using a Secret means that you don't need to include confidential data in your application code.

1. Creating Secrets Declaratively (Using YAML):

a. Base-64 Encoding:

```
echo -n 'admin' | base64
echo -n '1f2d1e2e67df' | base64
```

b. Using Base64 Encoding in creating Secret:

```
# db-user-pass.yaml
apiVersion: v1
kind: Secret
metadata:
  name: db-user-pass
  namespace: default
data:
  username: YWRtaW4=
  password: MWYyZDFlMmU2N2Rm
```

Deploy:

```
kubectl apply –f secret-db-user-pass.yaml
```

2. Creating Secrets Imperatively (From Command line):

If you want to skip the Base64 encoding step, you can create the same Secret using the kubectl create secret command. This is more convenient.

Example:

```
kubectl create secret generic test-secret
--from-literal='username=my-app' --from-literal='password=39528$vdg7Jb'
```

Example:

```
echo -n 'admin' > ./username.txt
echo -n '1f2d1e2e67df' > ./password.txt

kubectl create secret [TYPE] [NAME] [DATA]
```

Example

```
kubectl create secret generic db-user-pass-from-file
--from-file=./username.txt --from-file=./password.txt
```

Example:

```
kubectl get secrets db-user-pass –o yaml
```

3. Injecting "Secrets" into Pod As Environmental Variables:

 # my-secrets-pod-env.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-env-pod
spec:
  containers:
  - name: mycontainer
    image: redis
    env:
      - name: SECRET_USERNAME
        valueFrom:
          secretKeyRef:
            name: db-user-pass
            key: username
      - name: SECRET_PASSWORD
        valueFrom:
          secretKeyRef:
            name: db-user-pass
            key: password
  restartPolicy: Never
```

Validate:

```
kubectl exec secret-env-pod -- env
kubectl exec secret-env-pod -- env | grep SECRET
```

4. Injecting "Secrets" into Pod As Files inside the Volume:

```
# my-secrets-vol-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: secret-vol-pod
spec:
  containers:
    - name: test-container
      image: nginx
      volumeMounts:
        - name: secret-volume
```

```
        mountPath: /etc/secret-volume
  volumes:
    - name: secret-volume
      secret:
        secretName: test-secret
```

Validate:

```
kubectl exec secret-vol-pod -- ls /etc/secret-volume
kubectl exec secret-vol-pod -- cat /etc/secret-volume/username
kubectl exec secret-vol-pod -- cat /etc/secret-volume/password
```

5. Displaying Secret:

```
kubectl get secret <NAME>
kubectl get secret <NAME> -o wide
kubectl get secret <NAME> -o yaml
kubectl get secret <NAME> -o json

kubectl describe secret <NAME>
```

6. Running operations directly on the YAML file:

```
kubectl [OPERATION] –f [FILE-NAME.yaml]

kubectl get –f [FILE-NAME.yaml]
kubectl delete –f [FILE-NAME.yaml]
kubectl get -f [FILE-NAME.yaml]
kubectl create -f [FILE-NAME.yaml]
```

7. Delete Secret:

```
kubectl delete secret <NAME>
```