CKA Curriculum Part 3 - Application Lifecycle Management

Understand deployments and how to perform rolling updates and rollbacks

Deployments are intended to replace Replication Controllers. They provide the same replication functions (through Replica Sets) and also the ability to rollout changes and roll them back if necessary. An example configuration is shown below:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 5
  template:
    metadata:
      labels:
        app: nginx-frontend
    spec:
      containers:
      - name: nginx
        image: nginx:1.14
        ports:
        - containerPort: 80
```

We can then describe it with `kubectl describe deployment nginx-deployment`

To update an existing deployment, we have two main options:

- Rolling Update
- Recreate

A rolling update, as the name implies, will swap out containers in a deployment with one created by a new image.

Use a rolling update when the application supports having a mix of different pods (aka application versions). This method will also involve no downtime of the service, but will take longer to bring up the deployment to the requested version

A recreate will delete all the existing pods and then spin up new ones. This method will involve downtime. Consider this a “bing bang” approach

Examples listed in the Kubernetes documentation are largely imperative, but I prefer to be declarative. As an example, create a new yaml file and make the required changes, in this example, the version of the nginx container is incremented.

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 5
  template:
    metadata:
      labels:
        app: nginx-frontend
    spec:
      containers:
      - name: nginx
        image: nginx:1.15
        ports:
        - containerPort: 80
```

We can then apply this file `kubectl apply -f updateddeployment.yaml --record=true`

Followed by the following:

```
kubectl rollout status deployment/nginx-deployment

Waiting for deployment "nginx-deployment" rollout to finish: 2 out of 5 new replicas have been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 2 out of 5 new replicas have been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 2 out of 5 new replicas have been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 2 out of 5 new replicas have been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 3 out of 5 new replicas have been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 3 out of 5 new replicas have been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 3 out of 5 new replicas have been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 3 out of 5 new replicas have been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 4 out of 5 new replicas have been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 4 out of 5 new replicas have been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 4 out of 5 new replicas have been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 4 out of 5 new replicas have been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 4 out of 5 new replicas have been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 1 old replicas are pending termination...
Waiting for deployment "nginx-deployment" rollout to finish: 1 old replicas are pending termination...
Waiting for deployment "nginx-deployment" rollout to finish: 1 old replicas are pending termination...
Waiting for deployment "nginx-deployment" rollout to finish: 4 of 5 updated replicas are available...
deployment "nginx-deployment" successfully rolled out
```

We can also use the kubectl rollout history to look at the revision history of a deployment

```
kubectl rollout history deployment/nginx-deployment

deployment.extensions/nginx-deployment
REVISION  CHANGE-CAUSE
1         <none>
2         <none>
4         <none>
5         kubectl apply --filename=updateddeployment.yaml --record=true
```

Alternatively, we can also do this imperatively:

```
kubectl --record deployments/nginx-deployment set image deployments/nginx-deployment nginx=nginx:1.9.1

deployment.extensions/nginx-deployment image updated
deployment.extensions/nginx-deployment image updated
```

Rollback

To rollback to the previous version:

```
kubectl rollout undo deployment/nginx-deployment
```

To rollback to a specific version:

```
kubectl rollout undo deployment/nginx-deployment --to-revision 5
```

Where revision number comes from `kubectl rollout history deployment/nginx-deployment`

Know various ways to configure applications

I personally find this subject a bit ambiguous, but we can attempt to break this down into the following objet types:

Jobs

Configmaps

Secrets

Environment variables

Jobs

A job is simply an application that runs to completion within a pod. Use case for this would be something like batch processing.

The example below executes a perl command inside a container that calculates pi to 2000 places, printing out the result. After it has completed, the pod will terminate.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
```

```
spec:
  template:
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
  backoffLimit: 4
```

We can check the status of the job with the following:

```
kubectl get jobs

NAME   COMPLETIONS   DURATION   AGE
pi     1/1           97s        102s
```

And the respective container:

```
kubectl get pods
NAME       READY   STATUS      RESTARTS   AGE
pi-vjk5m   0/1     Completed   0          2m54s
```

To view the output from this pod:

```
kubectl logs pi-vjk5m

3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706798214808651328230664709384460955058223172535 9………..
```

Config maps

Configmaps are a way to decouple configuration from pod manifest file. Obviously, the first step is to create a config map before we can get pods to use them:

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

“Map-name” is a arbitrary name we give to this particular map, and “data-source” corresponds to a key-value pair that resides in the config map.

```
kubectl create configmap vt-cm --from-literal=blog=virtualthoughts.co.uk
```

At which point we can then describe it:

```
kubectl describe configmap vt-cm
Name:         vt-cm
Namespace:    default
Labels:       <none>
Annotations:  <none>

Data
====
blog:
----
virtualthoughts.co.uk
```

To reference this config map in a pod, we declare it in the respective yaml:

```
apiVersion: v1
kind: Pod
metadata:
 name: config-test-pod
spec:
 containers:
 - name: test-container
   image: busybox
   command: [ "/bin/sh", "-c", "env" ]
   env:
     - name: BLOG_NAME
       valueFrom:
         configMapKeyRef:
           name: vt-cm
           key: blog
 restartPolicy: Never
```

The pod above will output the environment variables, so we can validate it's leveraged the config map by extracting the logs from the pod:

```
kubectl logs config-test-pod
KUBERNETES_PORT=tcp://10.96.0.1:443
KUBERNETES_SERVICE_PORT=443
HOSTNAME=config-test-pod
SHLVL=1
HOME=/root
BLOG_NAME=virtualthoughts.co.uk
KUBERNETES_PORT_443_TCP_ADDR=10.96.0.1
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
KUBERNETES_PORT_443_TCP_PORT=443
KUBERNETES_PORT_443_TCP_PROTO=tcp
KUBERNETES_PORT_443_TCP=tcp://10.96.0.1:443
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_SERVICE_HOST=10.96.0.1
PWD=/
```

Secrets

Secrets allow us to store and manage sensitive information pertaining to our applications, which can take the form of a variety of objects such as usernames, passwords, ssh keys and much more. Similarly to configmaps, secrets are designed to decouple this information directly from the pod declaration.

As part of a Kubernetes cluster stand up, secrets are already leveraged, and we can view them by executing the following:

```
kubectl get secrets --all-namespaces
```

An example to create one:

```
kubectl create secret generic my-secret --from-literal=username=dbu --from-literal=pass=dbp
```

We leverage secrets in a very similar way to configmaps. (note as environment variables is one of a number of ways to leverage secrets)

```
apiVersion: v1
kind: Pod
metadata:
 name: secret-test-pod
spec:
 containers:
 - name: test-container
   image: busybox
   command: [ "/bin/sh", "-c", "env" ]
   env:
     - name: DB_Username
       valueFrom:
         secretKeyRef:
           name: my-secret
           key: username
     - name: DB_Pass
       valueFrom:
         secretKeyRef:
           name: my-secret
           key: pass
 restartPolicy: Never
```

```
kubectl logs secret-test-pod
KUBERNETES_SERVICE_PORT=443
KUBERNETES_PORT=tcp://10.96.0.1:443
HOSTNAME=secret-test-pod
DB_Username=dbu
SHLVL=1
HOME=/root
DB_Pass=dbp
KUBERNETES_PORT_443_TCP_ADDR=10.96.0.1
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
KUBERNETES_PORT_443_TCP_PORT=443
KUBERNETES_PORT_443_TCP_PROTO=tcp
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_PORT_443_TCP=tcp://10.96.0.1:443
KUBERNETES_SERVICE_HOST=10.96.0.1
```

```
PWD=/
```

Environment Variables

For more simple, direct configuration we can manipulate the environment variables directly within the pod manifest:

```
apiVersion: v1
kind: Pod
metadata:
 name: secret-test-pod
spec:
 containers:
 - name: test-container
   image: busybox
   command: [ "/bin/sh", "-c", "env" ]
   env:
     - name: DB_Username
       value:  "some username"
     - name: DB_Pass
       Value: "some password"
 restartPolicy: Never
```

Know how to Scale Applications

Constantly adding more, individual pods is not a sustainable model for scaling an application. To facilitate applications at scale, we need to leverage higher level constructs such as replicasets or deployments.

As an example, if the following is deployed:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
 name: nginx-deployment
spec:
 replicas: 5
 template:
   metadata:
     labels:
       app: nginx-frontend
   spec:
     containers:
     - name: nginx
       image: nginx:1.14
       ports:
       - containerPort: 80
```

If we wanted to scale this, we can simply modify the yaml file and scale up/down the deployment by modifying the “replicas” field, or modify it in the fly:

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

Understand the primitives necessary to create a self-healing application

Deployments facilitate this by employing a reconciliation loop to check the number of deployed pods matches what’s defined in the yaml file. Under the hood, deployments leverage ReplicaSets, which are primarily responsible for this feature.

Stateful Sets are similar to deployments, for example they manage the deployment and scaling of a series of pods. However, in addition to deployments they also provide guarantees about the ordering and uniqueness of Pods. A StatefulSet maintains a sticky identity for each of their Pods. These pods are created from the same spec, but are not interchangeable: each has a persistent identifier that it maintains across any rescheduling.

StatefulSets are valuable for applications that require one or more of the following.

- Stable, unique network identifiers.
- Stable, persistent storage.
- Ordered, graceful deployment and scaling.
- Ordered, automated rolling updates.

```
apiVersion: apps/v1
```

```
kind: StatefulSet
metadata:
 name: nginx-statefulset
spec:
 selector:
   matchLabels:
     app: vt-nginx
 serviceName: "nginx"
 replicas: 2
 template:
   metadata:
     labels:
       app: vt-nginx
   spec:
     containers:
     - name: vt-nginx
       image: nginx:1.7.9
       ports:
       - containerPort: 80
```