Demo - Introduction to Kubernetes

Create a cluster

```
# create a resource group
az group create -n mycluster -l northeurope

# create the k8s cluster in this group
az acs create -g mycluster -n mycluster --ssh-key-value ~/.ssh/id_rsa.pub -t kubernetes

# show that context is not there yet
kubectl config get-contexts     # "mycluster" does not show up

# import credentials to kubectl client
az acs kubernetes get-credentials -n ccdemo -g ccdemo
kubectl config get-context      # "mycluster" context is added
kubectl get nodes               # shows cluster nodes
```

Create a pod on the cluster

Create a yaml file mywebserver-pod.yaml with the following contents:

```
apiVersion: v1
kind: Pod
metadata:
  name: mywebserver
  labels:
    app: myapp
spec:
  containers:                   # alternatively, use the image which was
  - name: nginx                 # provisioned to docker hub in the prev demo
    image: nginx
    ports:
    - containerPort: 80
```

Now deploy this pod resource onto the cluster:

```
# create the pod
kubectl create -f mywebserver-pod.yaml

# list the pod
kubectl get pods

# where does it run?
kubectl describe pod mywebserver
```

To show that there's a real docker container running somewhere on a node, we can track it down on the cluster:

```
ssh -A azureuser@[address-or-name-of-master]
> ssh azureuser@[address-or-name-of-node-where-pod-is-running]
> docker ps                     # we can see that our "mywebserver" container is running
> exit
> exit
```

How can we access the web server content now? The port is *not* exposed in any way, so one way would be to go into the container itself:

```
# run a new command inside of our "mywebserver" pod
kubectl exec -ti mywebserver /bin/bash
# install curl if needed: apt-get update and apt-get install curl
```

```
> curl localhost                # we see our webserver answering
```

Alternatively, we can just use port-forwarding to our local machine for convenience:

```
kubectl port-forward mywebserver 8080:80

# from local machine:
curl localhost:8080             # we see our webserver answering
```

Creating a deployment of pods onto the cluster

Create a yaml file mywebserver-deployment.yaml with the following contents:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
    name: mywebserverdeployment
    namespace: default
spec:
    replicas: 3
    selector:
        matchLabels:
          app: myapp
    template:
        metadata:
            labels:
                app: myapp
                tier: data
        spec:
            containers:
            - name: nginx
              image: nginx
              imagePullPolicy: Always
              resources:
                requests:
                  cpu: 100m
                  memory: 200Mi
              ports:
              - containerPort: 80
```

Now create the deployment on the cluster:

```
kubectl delete pod mywebserver                # delete the single pod
kubectl get pods                              # show that there's no pods
kubectl create -f mywebserver-deployment.yaml  # create the deployment
kubectl get deployment -o wide                # show deployment
kubectl get pods -o wide                      # show 3 pods
```

Exposing a set of pods as a Service

Now that we have multiple pods, it would not be unreasonable to expect some form of traffic balancing across those so we utilize all resources. This is done through a Kubernetes "Service". A service can have assigned a type, depending on how public it is, ranging from: ClusterIP over NodePort to LoadBalancer. A service selects its pods based on a selector, which queries the pod's labels and label values.

To create a service, create a file mywebserver-service.yaml with the following contents:

```
apiVersion: v1
kind: Service
metadata:
         name: myservice
         namespace: default
         labels:
             app: myapp
```

```
            tier: tier-label
spec:
         type: ClusterIP
         ports:
         - port: 80
         selector:
              app: myapp
```

Then create the service on the cluster:

```
kubectl create -f mywebserver-service.yaml  # create the svc
kubectl get svc                             # show the svc
kubectl describe svc                        # show which nodes/ip's it matches
```

Now with a ClusterIP type-of service, we can access this from within the cluster. To do so, let's create a test-pod from where we can run the command. We create a pod like this:

```
apiVersion: v1
kind: Pod
metadata:
  name: myclient
  labels:
    app: myapp
spec:
  containers:
  - name: client
    image: ubuntu
    command: ["sleep"]
    args: ["infinity"]
```

To test connectivity with the myservice service:

```
kubectl exec -ti myclient /bin/bash
> apt-get update
> apt-get install curl
> curl myservice
> apt-get install dnsutils
> nslookup myservice
> exit
```

As a next step, let's modify the service and expose it publically using a load balancer:

```
kubectl edit svc myservice
# change the service type to "LoadBalancer"
kubectl get svc        # show how public ip is being provisioned
curl [public-ip-of-myservice]
```

Persistent Storage

There is different ways in Azure to have containers persist data: Azure Disks or Azure Files. We'll have a look at Azure files here.

In order to set up Azure Files with your cluster:

- make sure you have a pre-provisioned Azure Storage account in the same resource group and location as your cluster. The Kubernetes auto-provisioning will later try to find this account automatically.
- create a storageClass for it first, in a file azurefiles-storageclass.yaml:

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: azurefile
provisioner: kubernetes.io/azure-file
```

```
parameters:
  skuName: Standard_LRS
```

```
kubectl create -f azurefiles-storageclass.yaml
```

Now that we have a storageClass, indicating that we have storage available of the kind "Azure Files", we need to "claim" dynamically chunks of this storage. To do so, we're creating a so-called Persistent Volume Claim, reffering to our storage kind called azurefile in a file mypvc.yaml:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
    name: mypvc
    labels:
        app: app-label
        tier: tier-label
spec:
    storageClassName: azurefile # standard or default
    accessModes:
        - ReadWriteMany
    resources:
        requests:
            # The amount of the volume's storage to request
            storage: 5Gi
```

```
kubectl create -f mypvc.yaml
kubectl get pvc -o wide      # the pvc should show up as being "Bound"
```

Now that we have a claimed piece of storage, we can start using it within our pods. Let's once more have nginx serve up some content; this time served from within Azure Files. Adapt the previous deployment to something like the below:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
    name: mywebserverdeployment
    namespace: default
spec:
    replicas: 3
    selector:
        matchLabels:
          app: myapp
    template:
        metadata:
            labels:
                app: myapp
                tier: data
        spec:
            volumes:
            - name: myvol
              persistentVolumeClaim:
                claimName: mypvc
            containers:
            - name: nginx
              image: nginx
              imagePullPolicy: Always
              resources:
                requests:
                  cpu: 100m
                  memory: 200Mi
              ports:
              - containerPort: 80
              volumeMounts:
              - mountPath: /usr/share/nginx/html
                name: myvol
```

Redeploy the deployment.

> TODO - NGINX SEEMS TO RETURN A 403 FORBIDDEN - NEED TO FIND OUT WHY