Application deployment on AKS

Azure Kubernetes Service (AKS) is a managed container orchestration service, based on the open source Kubernetes system, which is available on the Microsoft Azure public cloud.

A **Pod** represents a unit of deployment: a single instance of an application in Kubernetes, which might consist of either a single container. or a small number of containers that are tightly coupled and that share resources. **Deployments** represent a set of multiple, identical Pods with no unique identities. A Deployment runs multiple replicas of our application and automatically replaces any instances that fail or become unresponsive.

Create Namespace

Kubernetes supports multiple virtual clusters backed by the same physical cluster. These virtual clusters are called **namespaces**.

To create a new namespace use the command

$ kubectl create namespace <your\_namespace>

namespace/<your\_namespace> created

Create Kubernetes secret

Kubernetes **secret** objects let us store and manage sensitive information, such as passwords, OAuth tokens, and ssh keys. Putting this information in a secret is safer and more flexible than putting it verbatim in a Pod.

Create a Kubernetes secret called productsapi-secret to hold cosmosdb and application insights details, so that you don’t need to hard-code them in the YAML files.

$ kubectl create secret generic productsapi-secret --from-literal=CosmosDb\_\_Account="<cosmos\_uri>" --from-literal=CosmosDb\_\_Key="<cosmos\_primary\_or\_secondary\_key>" --from-literal=ApplicationInsights\_\_InstrumentationKey="<app\_insights\_key>" --namespace <your\_namespace>

Running the kubectl describe command will show that the secrets are encrypted

$ kubectl get secret productsapi-secret -o yaml -n <your\_namespace>

apiVersion: v1

data:

ApplicationInsights\_\_InstrumentationKey: <app\_insights\_key>

CosmosDb\_\_Account: <cosmos\_uri>

CosmosDb\_\_Key: <cosmos\_primary\_or\_secondary\_key>

kind: Secret

metadata:

creationTimestamp: "2019-09-06T13:03:04Z"

name: productsapi-secret

namespace: <your\_namespace>

resourceVersion: "12708756"

selfLink: /api/v1/namespaces/<your\_namespace>/secrets/productsapi-secret

uid: a9ba20d7-d0a6-11e9-a0ed-36a1d23dd1a3

type: Opaque

Create Kubernetes ConfigMap

**ConfigMaps** allow us to decouple configuration artifacts from image content to keep containerized applications portable. Create a Kubernetes ConfigMap called productsapi-config to hold cosmosdb Database and Container names, so that you don’t need to hard-code them in the YAML files.

$ kubectl create configmap productsapi-config --from-literal=CosmosDb\_\_DatabaseName="Training" --from-literal=CosmosDb\_\_ContainerName="Products" -n <your\_namespace>

Running the kubectl describe command will show that the values in the ConfigMap are not encrypted

$ kubectl describe configmap productsapi-config -n <your\_namespace>

Name: productsapi-config

Namespace: saravananp

Labels: <none>

Annotations: <none>

Data

====

CosmosDb\_\_ContainerName:

----

Products

CosmosDb\_\_DatabaseName:

----

Training

Events: <none>

Deploy Products API Pod

$ cd k8s

We will be deploying Products API Pod using the productsapi-deployment.yaml file in this directory.

Let us look at some of the configurations in this yaml file:

* Replace <unique-acr-name> inside the yaml file with the name of your Azure Container Registry

image: <unique-acr-name>.azurecr.io/aksworkshop/dev/productsapi:v1

* Many applications running for long periods of time eventually transition to broken states, and cannot recover except by being restarted. Kubernetes provides liveness probes to detect and remedy such situations. In the below configuration periodSeconds specifies that the kubelet should perform a liveness probe every 30 seconds. The initialDelaySeconds field tells the kubelet that it should wait 10 seconds before performing the first probe. To perform a probe, the kubelet executes a httpGet command for path /healthcheck/live in the Container. If the command succeeds, the kubelet considers the Container to be alive and healthy. If the command returns a failure value, the kubelet kills the Container and restarts it.

livenessProbe:

failureThreshold: 3

httpGet:

path: /healthcheck/live

port: 80

initialDelaySeconds: 10

timeoutSeconds: 1

periodSeconds: 30

successThreshold: 1

* Readiness probes are configured similarly to liveness probes. The only difference is that you use the readinessProbe field instead of the livenessProbe field.

readinessProbe:

failureThreshold: 3

httpGet:

path: /healthcheck/ready

port: 80

initialDelaySeconds: 15

timeoutSeconds: 5

periodSeconds: 60

successThreshold: 1

* Requests and Limits

For each resource, containers can specify a resource request and limit, 0 <= request <= Node Allocatable & request <= limit <= Infinity. If a pod is successfully scheduled, the container is guaranteed the amount of resources requested. Scheduling is based on requests and not limits. The pods and its containers will not be allowed to exceed the specified limit.

resources:

requests:

memory: "128Mi"

cpu: "100m"

limits:

memory: "256Mi"

cpu: "500m"

* We use envFrom to define all of the ConfigMap’s data as container environment variables. The key from the ConfigMap becomes the environment variable name in the Pod. Similary all keys from the secret given below become environment variable name in the Pod.

envFrom:

- secretRef:

name: productsapi-secret

- configMapRef:

name: productsapi-config

Having looked at the key configurations in this file, we will now deploy the Pod using the yaml file.

$ kubectl apply -f productsapi-deployment.yaml -n <your\_namespace>

**Verify that the pods are up and running**

$ kubectl get pods -l app=productsapi -n <your\_namespace>

If the pods are not starting, not ready or are crashing, you can view their logs using kubectl logs

$ kubectl logs <pod name> -n <your\_namespace>

$ kubectl describe pod <pod name> -n <your\_namespace>

Issuing below command will get into the pod and verify the api (curl <http://localhost/api/products> ).

$ kubectl exec -it {podName} /bin/bash -n <your\_namespace>

Deploy Products API Kubernetes Service with External Load Balancer

In Kubernetes, a **Service** is an abstraction which defines a logical set of Pods and a policy by which to access them . The set of Pods targeted by a Service is usually determined by a selector.

The different types of services in Kubernetes are

* *ClusterIP*: Exposes the Service on a cluster-internal IP. Choosing this value makes the Service only reachable from within the cluster.
* *NodePort*: Exposes the Service on each Node’s IP at a static port (the NodePort). We will be able to contact the NodePort Service, from outside the cluster, by requesting <NodeIP>:<NodePort>.
* *LoadBalancer*: Exposes the Service externally using a cloud provider’s load balancer.
* *ExternalName*: Maps the Service to the contents of the externalName field (e.g. [foo.bar.example.com](http://foo.bar.example.com/) ), by returning a CNAME record with its value.

In this exercise we will create a LoadBalancer service using productsapi-external-service.yaml

apiVersion: v1

kind: Service

metadata:

name: productsapisvc

spec:

selector:

app: productsapi

ports:

- protocol: TCP

port: 80

targetPort: 80

type: LoadBalancer

In the above yaml file, the service connects to all pods having label '*app: productsapi*'. External users will be able to use this service to connect to the targetPort (port 80) on pods with this label.

$ kubectl apply -f productsapi-external-service.yaml -n <your\_namespace>

**Retrieve the External-IP of the Service (This will take 1-2 mins)**

$ kubectl get service productsapisvc -o jsonpath="{.status.loadBalancer.ingress[\*].ip}" -w -n <your\_namespace>

$ kubectl get service -n <your\_namespace>

**Verify Products API Kubernetes Service**

$ curl http://{ExternalIP}/api/products

Browse one of the links below in your favorite browser to verify the api http://{ExternalIP}/swagger http://{ExternalIP}/api/products