# Application Deployment on local machine

### Source code

Clone the application code. Please ignore if its already cloned.

$ git clone https://dev.azure.com/cloudiq/K8S-Dev-Workshop/\_git/k8s-dev-workshop

$ cd k8s-dev-workshop

Source code Location: src/dotnet-core/demo.cosmosdb.api/demo.cosmosdb.api

$ cd src/dotnet-core/demo.cosmosdb.api/demo.cosmosdb.api

### Build Image

Docker is a platform for developers and sysadmins to develop, deploy, and run applications with containers. In this section we will build a docker image using a Dockerfile

* Open the terminal and change directory to source code location.
* The Dockerfile in this directory has 3 stages:

1. First stage (build) uses dotnet:2.2-sdk as base image, copies the dotnet source into the image and builds the dotnet project

FROM microsoft/dotnet:2.2-sdk AS build

WORKDIR /src

COPY . .

RUN dotnet restore "demo.cosmosdb.api.csproj"

RUN dotnet build "demo.cosmosdb.api.csproj" -c Release --no-restore

1. Second stage (publish) packs the application and its dependencies into the /app folder

FROM build AS publish

RUN dotnet publish "demo.cosmosdb.api.csproj" -c Release -o /app

1. Third stage uses dotnet:2.2-aspnetcore-runtime as base image and deploys the code by copying the /app folder from publish stage. It also exposes ports 80 and 443 for http and https connectivity.

FROM microsoft/dotnet:2.2-aspnetcore-runtime AS final

WORKDIR /app

EXPOSE 80

EXPOSE 443

COPY --from=publish /app .

ENTRYPOINT ["dotnet", "demo.cosmosdb.api.dll"]

1. Run the container using docker

Build docker image

$ docker build -t aksworkshop/dev/productsapi:v1 .

### Update Environment Variables

* Open file src/dotnet-core/demo.cosmosdb.api/demo.cosmosdb.api/env/env.dev
* CosmosDb\_\_Account=<cosmos\_uri>
* CosmosDb\_\_Key=<cosmos\_primary\_or\_secondary\_key>
* CosmosDb\_\_DatabaseName=Training
* CosmosDb\_\_ContainerName=Products
* ApplicationInsights\_\_InstrumentationKey=<app\_insights\_key>
* Update appropriate values.

### Run the container

$ docker run -d --name productsapi -p 3001:80 --env-file ./env/env.dev aksworkshop/dev/productsapi:v1

In the above command:

* --name productsapi --> gives name productsapi to the container
* -p 3001:80 --> binds port 80 of the container to port 3001 on the host machine
* --env-file ./env/env.dev --> get the environment variables from file .\env\env.dev
* aksworkshop/dev/productsapi:v1 --> name and tag of the image to be run

### Testing

* Browse <http://localhost:3001/swagger>  in your favorite browser to verify the api
* Create one or two products and verify.
* Check application health in <http://localhost:3001/healthcheck/live>
* Check application dependency health (cosmosdb health) in <http://localhost:3001/healthcheck/ready>

### Exec into the running container

$ docker exec -it productsapi /bin/bash

Run below command inside the container

$ curl http://localhost/api/products

Kubernetes deployments basics

Source code

Clone the application code

$ git clone https://dev.azure.com/cloudiq/K8S-Dev-Workshop/\_git/k8s-dev-workshop

$ cd k8s-dev-workshop

Kubernetes definition file

Kubernetes uses YAML files as input for the creation of objects such as PODs, Replicas, Deployments, Services etc.

A kubernetes definition file always contains 4 top level fields. The apiVersion, kind, metadata and spec.

This is simple yaml to deploy nginx POD in the cluster.

apiVersion: This is the version of the kubernetes API we’re using to create the object. Depending on what we are trying to create we must use the RIGHT apiVersion.

kind: The kind refers to the type of object we are trying to create. Some other possible values here could be ReplicaSet, Deployment or Service.

metadata: The metadata is data about the object like its name, labels etc. It’s IMPORTANT to note that under metadata, you can only specify name or labels or anything else that kubernetes expects to be under metadata. You CANNOT add any anything else that kubernetes expects to be under metadata. You CANNOT add any other property as you wish under this. However, under labels you CAN have any kind other property as you wish under this.

spec: The last section in the configuration file is the specification which is written as spec. Depending on the configuration file is the specification which is written as spec. Depending on the object we are going to create, this is were we provide additional information to object we are going to create.

POD

In kubernetes our ultimate aim is to deploy our application in the form of containers on a set of machines that are configured as worker nodes in a cluster. However, kubernetes does not deploy containers directly on the worker nodes.

The containers are encapsulated into a Kubernetes object known as PODs. A POD is a single instance of an application. A POD is the smallest object, that you can create in kubernetes.

This is simple yaml to deploy nginx POD in the cluster.

apiVersion: v1

kind: Pod

metadata:

name: nginx

labels:

app: app-nginx

type: front-end

spec:

containers:

- image: nginx

name: nginx

ports:

- containerPort: 80

name: http

Once the file is created, run the command kubectl create -f <file\_name> which is pod definition.yml and kubernetes creates the pod.

Run the follwoing command to create a POD.

$ kubectl apply -f .\k8s\nginx-pod-01.yaml

Verify

# displays all pods in the current namespace

$ kubectl get pods

# displays all pods in the current namespace and filter by the selectors

$ kubectl get pods --selector app=app-nginx

Deployment

A deployment is an object in Kubernetes that lets you manage a set of identical pods. Without a deployment, you’d need to create, update, and delete a bunch of pods manually.

With a deployment, you declare a single object in a YAML file. This object is responsible for creating the pods, making sure they stay up to date, and ensuring there are enough of them running

You can also easily autoscale your applications using a Kubernetes deployment.

apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2

kind: Deployment

metadata:

name: nginx-deployment

spec:

selector:

matchLabels:

app: deployment-nginx

replicas: 2 # tells deployment to run 2 pods matching the template

template:

metadata:

labels:

app: deployment-nginx

type: front-end

spec:

containers:

- name: nginx

image: nginx

ports:

- containerPort: 80

name: http

Run the following command to create a Deployment.

$ kubectl apply -f .\k8s\nginx-deployment-01.yaml

Verify you can see two pods are created from the deployment.

# get deployments

$ kubectl get deployments

# describe deployments

$ kubectl describe deployment nginx-deployment

# displays all pods in the current namespace

$ kubectl get pods

# displays all pods in the current namespace and filter by the selectors

$ kubectl get pods --selector app=deployment-nginx

Service

A Service in Kubernetes is an abstraction which defines a logical set of Pods and a policy by which to access them. Services enable a loose coupling between dependent Pods. A Service is defined using YAML (preferred) or JSON, like all Kubernetes objects. The set of Pods targeted by a Service is usually determined by a LabelSelector.

Services can be exposed in different ways by specifying a type in the ServiceSpec:

ClusterIP (default) - Exposes the Service on an internal IP in the cluster. This type makes the Service only reachable from within the cluster.

NodePort - Exposes the Service on the same port of each selected Node in the cluster using NAT. Makes a Service accessible from outside the cluster using <NodeIP>:<NodePort>. Superset of ClusterIP.

LoadBalancer - Creates an external load balancer in the current cloud (if supported) and assigns a fixed, external IP to the Service. Superset of NodePort.

The below yaml deploys external LoadBalancer service.

apiVersion: v1

kind: Service

metadata:

name: nginx-svc

spec:

selector:

app: deployment-nginx

ports:

- protocol: TCP

port: 80

targetPort: 80

type: LoadBalancer

Run the following command to create a Deployment.

$ kubectl apply -f .\k8s\nginx-service-01.yaml

Retrieve the External-IP of the Service

$ kubectl get service nginx-svc -o jsonpath="{.status.loadBalancer.ingress[\*].ip}" -w

$ kubectl get service

Verify deployed Service

$ curl http://{ExternalIP}

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