JUNE 7,2020

6 Steps To Run .NET Core Apps In Azure Kubernetes

TAGS AZURE KUBERNETES SERVICE KUBERNETES

Mastering Kubernetes is hard and a time-consuming task. However, getting started with Kubernetes in Azure as a .NET Core Developer is relatively easy. It takes six steps to get your application up and running in Azure Kubernetes Service (AKS), starting with a plain, regular .NET Core Web API.

This article will guide you through the entire process and explain all steps in detail, so you can follow and end up with having your application being executed in Kubernetes and being exposed to the internet.

Scope

The idea is to tell the story and to get you up and to run within a few paragraphs. To achieve this, several important topics are not in the scope of this article. However, you have to take care of those topics when building both - Docker Images and your Azure infrastructure for production. To name just a few of those topics:

- Securing Docker images
- Minimizing the layers and size of Docker images
- Securing Azure Container Registry
- Securing Azure Kubernetes Service
- Using a dedicated ingress controller to route inbound traffic in AKS
- Continuous observation using Azure Security Center

For demonstration purposes, I will take the "Weather Forecast" Web API blueprint created by Microsoft when you use `dotnet new webapi -lang c# -n HelloAKS`.

Requirements

To follow this guide, you should have three things installed and configured on your machine:

- Docker
- Azure CLI 2.0
- An Editor (I use Visual Studio Code)

With those tools installed and configured, we can jump in and move the .NET Core Application to a new Azure Kubernetes cluster.

1. Create A Docker Image

Having the .NET Core Application on your local machine, we have to create a Docker image from it. To do so, create two new files in the main project folder: `Dockerfile` and `.dockerignore`. Before we jump into writing the `Dockerfile`, let's add some typical patterns for .NET applications to `.dockerignore` to prevent them from being transferred to the Docker daemon when building the Docker image.

```
# .dockerignore
/bin/
/obj/
```

Creating Docker Images from .NET Core Applications is straight forward. Microsoft provides all the required Base-Images to make your application run in a Linux-based container. We are going to create a so-called multi-stage Docker Image.

That means we are starting from a Base-Image that contains .NET Core SDK to compile and publish the application inside of that image. Once that's done, we leverage another Base-Image that just holds all required stuff to run .NET Core Applications and copy our previously published application into the new Base-Image. From here, we expose the desired port and start the application.

Following this pattern allows us to dramatically optimize the size of the resulting Docker Image, as you can see here:

```
mcr.microsoft.com/dotnet/core/sdk       3.1-buster        a0dbfc5[illegible]
mcr.microsoft.com/dotnet/core/aspnet    3.1-buster-slim   4bee399eb313     2
```

Put the following in the `Dockerfile`

```
#Dockerfile
FROM mcr.microsoft.com/dotnet/core/sdk:3.1-buster AS builder
WORKDIR /src
COPY ./HelloAKS.csproj .
RUN dotnet restore HelloAKS.csproj
COPY . .
RUN dotnet build HelloAKS.csproj -c Debug -o /src/out
```

```
FROM mcr.microsoft.com/dotnet/core/aspnet:3.1-buster-slim
WORKDIR /app
COPY --from=builder /src/out .

EXPOSE 80
ENTRYPOINT ["dotnet", "HelloAKS.dll"]
```

Let's now build the Docker Image use Docker CLI.

Bash Copy

```
# build Docker image
docker build . -t HelloAKS:0.0.1
# Watch Docker

# pulling the images from MCR if they are not on your machine
# building .NET Application and creating the Docker Image
```

With your Docker image in place (you can always check all the Docker Images on your system using `docker images`), we can move on to the next step.

2. Create An Azure Container Registry

Although you can use the public Docker Hub to host your Docker Images, I use Azure Container Registry (ACR) because it integrates nicely with Azure Kubernetes Service (AKS), Azure Active Directory (Azure AD), Azure Security Center (ASC) and others. If you want to dive deeper into ACR, consult the articles of my Azure Container Registry Unleashed series.

We will group all our Azure resources in a dedicated Azure Resource Group called `hello-aks`. Let's create that one using Azure CLI.

Bash Copy

```
# create the resource group
az group create -n hello-aks -l westeurope
```

Once the resource is provisioned, we can spin-up a new ACR instance. We will stick with the "Basic" tier for the scope of this article, which costs around 0.15

Euro per day. The name of an ACR instance has to be globally unique across all Azure customers. So you should use `az acr check-name` to verify if the desired service-name is available.

Bash Copy

```bash
# check if your desired name is available
az acr check-name -n thorstenhans

# create the ACR instance
az acr create -n thorstenhans \
  -g hello-aks \
  --admin-enabled false \
  --sku Basic
```

Having the ACR in place, you should use the following command to store the unique cloud-resource-identifier in a variable. We need it in a few minutes:

Bash Copy

```bash
# store unique ACR id in ACR_ID
ACR_ID=$(az acr show -n thorstenhans -g hello-aks --query id -o tsv)
```

3. Push The Docker Image To ACR

We can't simply push the previously created Docker image to Azure Container Registry. Docker images have to follow the ACR image naming conventions (`uniquearname.azurecr.io/imagename:tag`), to push them into ACR. We can easily re-tag our image using `docker tag` as shown here:

Bash Copy

```bash
# re-tag the existing Docker Image
docker tag hello-aks:0.0.1 thorstenhans.azurecr.io/hello-aks:0.0.1
```

Additionally, we have to authenticate against ACR.

Bash Copy

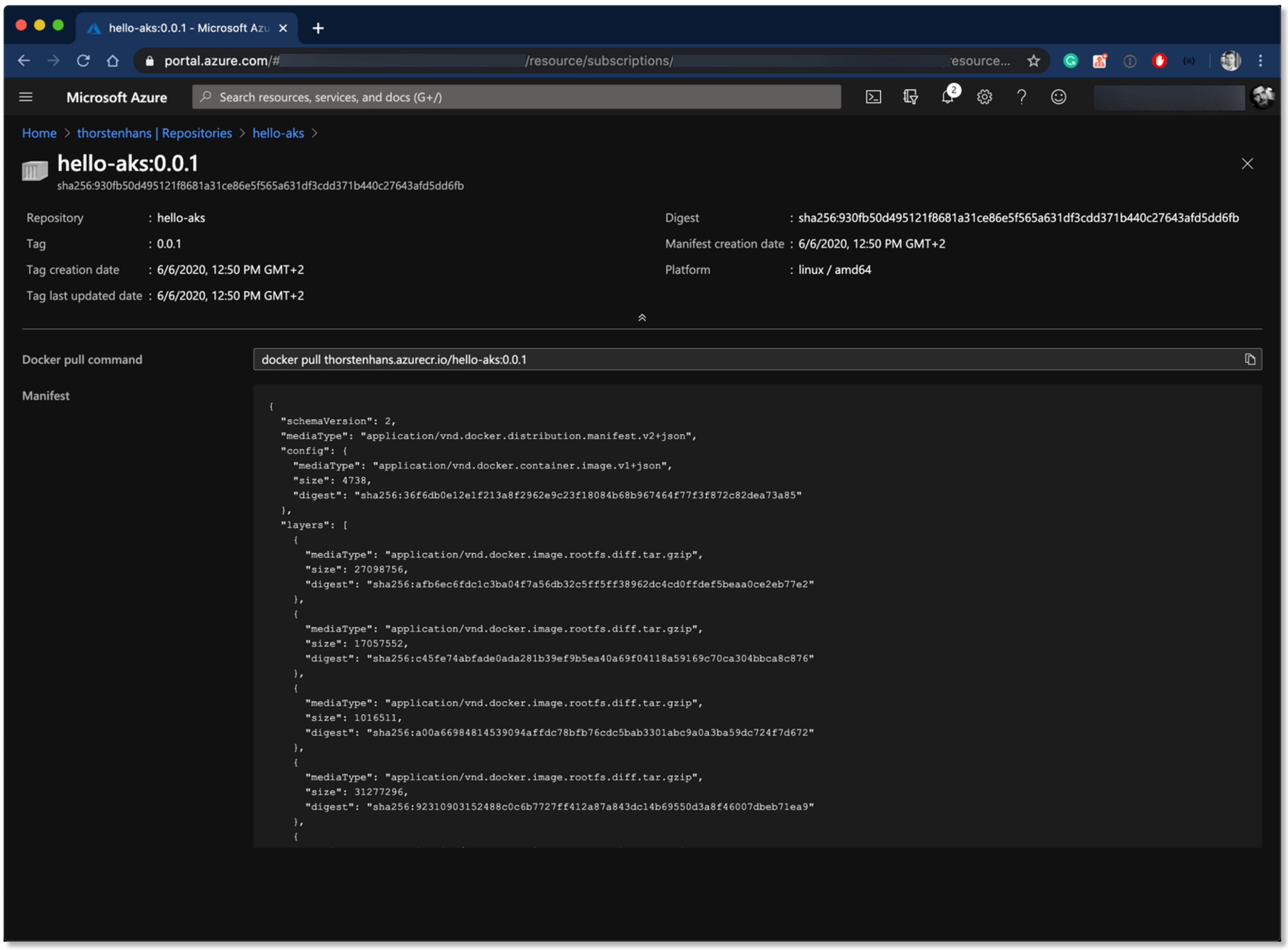
```bash
# authenticate against ACR
az acr login -n thorstenhans
```

Once the login has succeeded, we use standard Docker commands to push our Docker image to ACR. Docker knows where to route the image during `push` operation because of the ACR-specific prefix.

Bash Copy

```bash
# push the docker image to ACR
docker push thorstenhans.azurecr.io/hello-aks:0.0.1
```

You can either use the Azure CLI or Azure Portal to verify that our Docker Image has been pushed to ACR. (`az acr repository list -n thorstenhans`)


hello-aks:0.0.1 - Microsoft Azu
portal.azure.com/# /resource/subscriptions/ esource...
Microsoft Azure
Search resources, services, and docs (G+/)
Home > thorstenhans | Repositories > hello-aks >

hello-aks:0.0.1
sha256:930fb50d495121f8681a31ce86e5f565a631df3cdd371b440c27643afd5dd6fb

Repository : hello-aks
Tag : 0.0.1
Tag creation date : 6/6/2020, 12:50 PM GMT+2
Tag last updated date : 6/6/2020, 12:50 PM GMT+2
Digest : sha256:930fb50d495121f8681a31ce86e5f565a631df3cdd371b440c27643afd5dd6fb
Manifest creation date : 6/6/2020, 12:50 PM GMT+2
Platform : linux / amd64

Docker pull command: docker pull thorstenhans.azurecr.io/hello-aks:0.0.1

Manifest

```json
{
  "schemaVersion": 2,
  "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
  "config": {
    "mediaType": "application/vnd.docker.container.image.v1+json",
    "size": 4738,
    "digest": "sha256:36f6db0e12e1f213a8f2962e9c23f18084b68b967464f77f3f872c82dea73a85"
  },
  "layers": [
    {
      "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip",
      "size": 27098756,
      "digest": "sha256:afb6ec6fdc1c3ba04f7a56db32c5ff5ff38962dc4cd0ffdef5beaa0ce2eb77e2"
    },
    {
      "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip",
      "size": 17057552,
      "digest": "sha256:c45fe74abfade0ada281b39ef9b5ea40a69f04118a59169c70ca304bbca8c876"
    },
    {
      "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip",
      "size": 1016511,
      "digest": "sha256:a00a66984814539094affdc78bfb76cdc5bab3301abc9a0a3ba59dc724f7d672"
    },
    {
      "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip",
      "size": 31277296,
      "digest": "sha256:92310903152488c0c6b7727ff412a87a843dc14b69550d3a8f46007dbeb71ea9"
    },
    {
```

Azure Container Registry - Docker Image in ACR

4. Create A Kubernetes Cluster

We use Azure CLI to create a new Azure Kubernetes cluster. For demonstration purposes, our cluster will consist of just a single worker-node. Running a Kubernetes cluster with a single worker-node is not a recommendation. I just do it here to keep the overall Azure costs as low as possible. As part of the creation process, we will also attach our ACR instance to AKS. This is something you can also do with existing AKS clusters using the `az aks update` command.

Bash Copy

```bash
# create the AKS cluster
az aks create -n aks-demo \
  -g hello-aks \
  --enable-managed-identity \
  --attach-acr $ACR_ID \
  --node-count 1
```

When Azure CLI finished provisioning the AKS cluster (this could take some minutes), we need to set up `kubectl`, the Kubernetes CLI, which you use to communicate with any Kubernetes cluster. If you have `kubectl` already installed, you can skip the following command and continue with downloading the cluster configuration for `kubectl`.

Bash Copy

```bash
# Install kubectl
az aks install-cli
```

Once `kubectl` is installed on your machine, we need the configuration for the `aks-demo` cluster. Again Azure CLI offers the corresponding command:

Bash Copy

```bash
# download cluster configuration for kubectl
az aks get-credentials \
  -n aks-demo \
  -g hello-aks

# verify kubectl context
kubectl config get-contexts
# prints all cluster contexts

# if kubectl does not point to aks-demo, switch context
kubectl config use-context aks-demo
```

5. Create Kubernetes Deployment Manifests

In Kubernetes, we don't deploy Docker images directly. The smallest unit of work in Kubernetes is a so-called *Pod*. A pod offers a bunch of configuration opportunities to control the entire lifecycle of a containerized application.

When you dive deeper into Kubernetes, you will learn other building blocks like *ReplicaSet* or *Deployment*, which are wrappers for a *Pod* adding even more capabilities like replications and fine-granular update strategies. But for now, let's start with your first Pod. Create a new file in the main folder with the name `pod.yml` and add the following content. We will discuss the most critical parts in a second.

YAML Copy

```yaml
# pod.yml
apiVersion: v1
kind: Pod
metadata:
  name: first-netcore-app
  labels:
    app: hello-aks
    component: netcore-app
spec:
  containers:
    - image: thorstenhans.azurecr.io/hello-aks:0.0.1
      name: webapi
      ports:
        - containerPort: 80
```

As you can see, we specified two labels for our Pod. `app: hello-aks` and `component: netcore-app`. In Kubernetes, we use labels and label-selectors to couple components loosely. Labels are super-efficient, yet simple. Think of them as post-its that you attach to several things like the Pod in this case.

The `spec` describes the actual properties of the Pod. As you can see, we specify our previously published image and configure port exposure. We instruct Kubernetes to allow network traffic for our Docker container on port `80`.

Besides the Pod, we want to expose our API to the public. To make Pods accessible (either cluster-wide or externally), we have to create a *Service* in Kubernetes. Again in the project root folder, create a new file called `service.yml` and add the following:

YAML Copy

```yaml
# service.yml
apiVersion: v1
kind: Service
metadata:
  labels:
    app: hello-aks
  name: hello-aks
spec:
  ports:
    - port: 8080
      protocol: TCP
      targetPort: 80
  selector:
    app: hello-aks
    component: netcore-app
  type: LoadBalancer
```

Reviewing the Service definition, you should focus on `spec`. The spec defines that we want to create a service of type `LoadBalancer`. This will instruct Azure to allocate a public IP-address and create an Azure Load Balancer and route traffic from the public internet into our Kubernetes cluster.

Kubernetes supports different types of services. In real-world projects, you typically don't use `LoadBalancer`; instead, you use an Ingress controller (like, for example, NGINX Ingress) that controls inbound traffic. Cluster internal services would be of type `NodePort` or `ClusterIP`.

The `spec.selector` part defines where traffic should be routed to. This configuration will route all traffic appearing on port `8080` to port `80` to ALL artifacts inside of the cluster having the labels `app: hello-aks` and `component:netcore-app` associated.

You just described that network traffic would be routed dynamically into your cluster without coupling resources tightly together. **How cool is that?**

Having everything in place, we can move to the final step and deploy everything to AKS.

6. Deploy Your Application

Deploying artifacts to Kubernetes is straight forward; we use `kubectl apply` to deploy stuff to Kubernetes.

Bash Copy

```bash
# deploy artifacts to AKS
kubectl apply -f pod.yml
kubectl apply -f service.yml
```

Kubernetes will now spin up the Service and pull our Docker image from ACR to instantiate the Pod. Especially creating the service can take some time because the public IP has to be allocated and the Load Balancer has to be provisioned. You can use `kubectl` to watch for all services and wait until the external IP is associated.

Bash Copy

```bash
# wait for service to receive its external IP
kubectl get svc -w

NAME         TYPE           CLUSTER-IP     EXTERNAL-IP     PORT(S)
hello-aks    LoadBalancer   10.0.139.177   51.105.199.90   8080:31924/TCP
kubernetes   ClusterIP      10.0.0.1       <none>          443/TCP
```

Once you see the external IP assigned to the service, you can stop watching the services using `[CTRL]+C` and fire up a request to get weather data from our .NET Core application using `curl`:

Bash Copy

```bash
# get weather data from ASP.NET Core API in AKS
curl http://51.105.199.90:8080/weatherforecast | jq

[
    {
        "date": "2020-06-07T10:18:16.0014666+00:00",
        "temperatureC": -11,
        "temperatureF": 13,
        "summary": "Sweltering"
    },
    {
        "date": "2020-06-08T10:18:16.0014937+00:00",
        "temperatureC": 34,
        "temperatureF": 93,
        "summary": "Hot"
    },
    {
        "date": "2020-06-09T10:18:16.0015069+00:00",
        "temperatureC": 28,
        "temperatureF": 82,
        "summary": "Chilly"
    },
    {
        "date": "2020-06-10T10:18:16.0015195+00:00",
        "temperatureC": 41,
        "temperatureF": 105,
        "summary": "Warm"
    },
    {
        "date": "2020-06-11T10:18:16.0015348+00:00",
        "temperatureC": -4,
        "temperatureF": 25,
        "summary": "Warm"
    }
]
```

You should also check your Pod using `kubectl get pods`. Here you should append `-o wide` to get the IP address of the Pod. To see the link between the Service and the Pod, you can look into endpoints using `kubectl get endpoints`.

Bash Copy

```
# get pods and endpoints
kubectl get pods -o wide
NAME                READY   STATUS    RESTARTS   AGE     IP
first-netcore-app   1/1     Running   0          6m58s   10.244.0.10

kubectl get endpoints
NAME         ENDPOINTS          AGE
hello-aks    10.244.0.10:80     15m
kubernetes   20.50.227.85:443   21m
```

Recognize that the endpoint points to the IP address and port of the Pod.

Conclusion

It's a wrap. Six steps to get your .NET Core Web API up and running in Kubernetes. This is a good starting point. You can dive deeper from this point and adopt features like Deployments, ConfigMaps, and Secrets in Kubernetes. Additionally, you should look into things like health checks, resource-requests, and -limits to unleash the power of Kubernetes and start making your containerized application rock-solid.

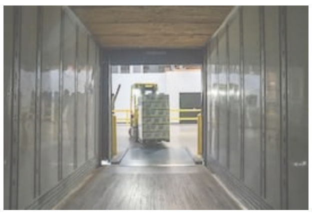
The entire code of this article can be found in the repository on GitHub at https://github.com/ThorstenHans/six-steps-to-aks.

RELATED POSTS

AKS cluster auto-scaler inside out

3 Ways to integrate ACR with AKS

AKS and ACR Integration - Revisited


The state of Helm 3 - Hands-On!

Follow my blog via mail

Enter your email address to follow my blog and receive notifications of new posts by email.

mahesh.dotnethelper.com

Subscribe

THORSTEN HANS © 2020. POWERED BY JEKYLL & NETLIFY.