

Please Release Me

Continuous Delivery,
DevOps, ALM and IoT in a
Mostly Microsoft Azure
World

Deploy a Dockerized ASP.NET Core Application to Kubernetes on Azure Using a VSTS CI/CD Pipeline: Part 1

Posted by Graham Smith on February 20, 2018 | 5 Comments

Over the past 18 months or so I've written a handful of blog posts about deploying [Docker containers using Visual Studio Team Services \(VSTS\)](#). The first post covered deploying a container to a Linux VM running Docker and other posts covered deploying containers to a cluster running DC/OS—all running in Microsoft Azure. Fast forward to today and everything looks completely different from when I wrote that first post: Docker is much more mature with features such as multi-stage builds dramatically streamlining the process of building source code and packaging it in to containers, and Kubernetes has emerged as a clear leader in the container orchestration battle and looks set to be a game-changing technology. (If you are new to Kubernetes I have a [Getting Started](#) blog post [here](#) with plenty of useful learning resources and tips for getting started.)

One of the key questions that's been on my mind recently is how to use Kubernetes as part of a CI/CD pipeline, specifically using VSTS to deploy to Microsoft's [Azure Container Service \(AKS\)](#), which is now specifically targeted at managing hosted Kubernetes environments. So in a new series of posts I'm going to be examining that very question, with each post

About the Author

Dr Graham Smith is a former research scientist who got bitten by the programming and database bug so badly that in 2000 he changed careers to become a full-time software developer.

After spending 12 years as a .NET

/ SQL Server software engineer Graham spent three years leading a major CI/CD pipeline implementation using Microsoft technologies followed by three years in senior IT leadership roles leading wider DevOps initiatives. Graham currently works for **DevOpsGroup** as a Senior DevOps



building on previous posts as I drill deeper in to the details. In this post I'm starting as simply as I possibly can whilst still answering the key question of how to use VSTS to deploy to Kubernetes. Consequently I'm ignoring the Kubernetes experience on the development workstation, I only deploy a very simple application to one environment and I'm not looking at scaling or rolling updates. All this will come later, but meantime I hope you'll find that this walkthrough will whet your appetite for learning more about CI/CD and Kubernetes.

Development Workstation Configuration

These are the main tools you'll need on a Windows 10 Pro development workstation (I've documented the versions of certain tools at the time of writing but in general I'm always on the latest version):

- Visual Studio 2017—version 15.5.6 with the ASP.NET and web development workload.
- Docker for Windows—stable channel 17.12.0-ce.
- Windows Subsystem for Linux (WSL)—see [here](#) for installation details. I'm still using [Bash on Ubuntu on Windows](#) that I installed before WSL moved to the Microsoft Store and in this post I assume you are using Ubuntu. The aim of installing WSL is to run Azure CLI, although technically you don't need WSL as Azure CLI will run happily under a Windows command prompt. However using WSL facilitates running Azure CLI commands from a Bash script.
- Azure CLI on Windows Subsystem for Linux—see [here](#) for installation (and subsequent upgrade) instructions. There are several ways to login to Azure from the CLI however I've found that the [interactive log-in](#) works well since once you're logged-in you remain so for quite a long time (many days for me so far). Use `az -v` to check which version you are on (2.0.27 was latest at time of writing).
- kubectl on Azure CLI—the kubectl CLI is used to interact with a Kubernetes cluster. Install using `sudo az aks install-cli`.

Create Services in Microsoft Azure

There are several services you will need to set up in Microsoft Azure:

- Azure Container Registry—see [here](#) for an overview and links to the various methods for creating an ACR. I use the **Standard SKU** for the better performance and increased storage.
- Azure Container Service (AKS) cluster—see [here](#) for more details about AKS and how to create a cluster, however you may find it easier to use my script below. I started off by creating a cluster and then destroying it after each use until I did some tests and found that a one-node cluster was costing pennies per day rather than the pounds per day I had assumed it would cost and now I just keep the cluster running.

Transformation

Consultant and in his spare time is a **CodeClub** volunteer helping to teach kids programming on the Raspberry Pi platform. The opinions expressed here are Graham's and not necessarily those of the DevOpsGroup.

Simple-Talk Awards 2015/16



Blog Series

Internet of Things
Continuous Delivery with Containers
Continuous Delivery with TFS / VSTS
Continuous Delivery with TFS
Continuous Delivery with VSO
ALM Practices
Getting Started
Tools, Tips and Tricks

- From a WSL Bash prompt run `nano create_k8s_cluster.sh` to bring up the nano editor with a new empty file. Copy and paste (by pressing right mouse key) the following script:

```
#!/bin/bash

azureSubscriptionId="xxx1x111-1x1x-1111-xx1x-x1x1111111"
resourceGroup="k8sResourceGroup"
clusterName="k8sCluster"
location="eastus"

# Useful if you have more than one Aure subscription
az account set --subscription $azureSubscriptionId

# Resource group for cluster - only availble in certain regions at time o
f writing
az group create --location $location --name $resourceGroup

# Create actual cluster
az aks create --resource-group $resourceGroup --name $clusterName --node-
count 1 --generate-ssh-keys

# Creates a config file at ~/.kube on local machine to tell kubectl which
cluster it should work with
az aks get-credentials --resource-group $resourceGroup --name
$clusterName

# Copies config file to a location easily accessible by Notepad
cp ~/.kube/config /mnt/c/Users/Public
```

- Change the variables to your suit your requirements. If you only have one Azure subscription you can delete the lines that set a particular subscription as the default, otherwise use `az account list` to list your subscriptions to find the ID.
 - Exit out of nano making sure you save the changes (Ctrl +X, Y) and then apply permissions to make it executable by running `chmod 700 create_k8s_cluster.sh`.
 - Next run the script using `./create_k8s_cluster.sh`.
 - One the cluster is fully up-and-running you can show the Kubernetes dashboard using `az aks browse --resource-group $resourceGroup --name $clusterName`.
 - You can also start to use the `kubectl` CLI to explore the cluster. Start with `kubectl get nodes` and then have a look at [this cheat sheet](#) for more commands to run.
 - The cluster will probably be running an older version of Kubernetes—you can check and find the procedure for upgrading [here](#).
- Private VSTS Agent on Linux—you can use the hosted agent (called Hosted Linux Preview at time of writing) but I find it runs very slowly and additionally because a new agent is used every time you perform a build it has to pull docker images down each time which adds to the slowness. In a future post I'll cover running a VSTS agent from a Docker image running on the Kubernetes cluster but for now you can create a private Linux agent running on a VM using [these](#) instructions. Although they date back to

Follow via Email

Follow via RSS

RSS - Posts

RSS - Comments

Recent Posts

Create an Azure DevOps

Services Self-Hosted

Agent in Azure Using

Terraform, Cloud-init—

and Azure DevOps

Pipelines!

Deploy a Dockerized

ASP.NET Core

Application to Azure

Kubernetes Service

Using a VSTS CI/CD

Pipeline: Part 4

Upgrade a Dockerized

ASP.NET Core

October 2016 they still work fine (I've checked them and tweaked them slightly).

- Since we will only need this agent to build using Docker you can skip steps 5b, 5c and 5d.
- Install a newer version of Git—I used [these](#) instructions.
- Install docker-compose using [these](#) instructions and choosing the Linux tab.
- Make the `docker-user` a member of the `docker` group by executing `usermod -aG docker ${USER}`.

Create VSTS Endpoints

In order to talk to the various Azure services you will need to create the following endpoints in VSTS (from the cog icon on the toolbar choose **Services > New Service Endpoint**):

- Azure Resource Manager—to point to your MSDN subscription. You'll need to authenticate as part of the process.

- Kubernetes Service Connection—to point to your Kubernetes cluster. You'll need the FQDN to the cluster (prefixed with `https://`) which you can get from the Azure CLI by executing `az aks show --resource-group $resourceGroup --name $clusterName`, passing in your own resource group and cluster names. You'll also need the contents of the kubeconfig file. If you used the script above to create the cluster then the script copied the config file to `C:\Users\Public` and you can use Notepad to copy the contents.

Application to the Latest

Version of .NET Core

Deploy a Dockerized

ASP.NET Core

Application to Azure

Kubernetes Service

Using a VSTS CI/CD

Pipeline: Part 3

Deploy a Dockerized

ASP.NET Core

Application to Kubernetes

on Azure Using a VSTS

CI/CD Pipeline: Part 2

Tags

Agile ALM **Application**

Insights automated testing

Azure Azure Automation

Azure CLI **Azure**

Resource

Manager

Containers

Continuous
Delivery

Continuous Integration

DevOps **Docker** **Git**

GitHub IIS JSON kubectl

Kubernetes **Linux**

Microsoft **Test**

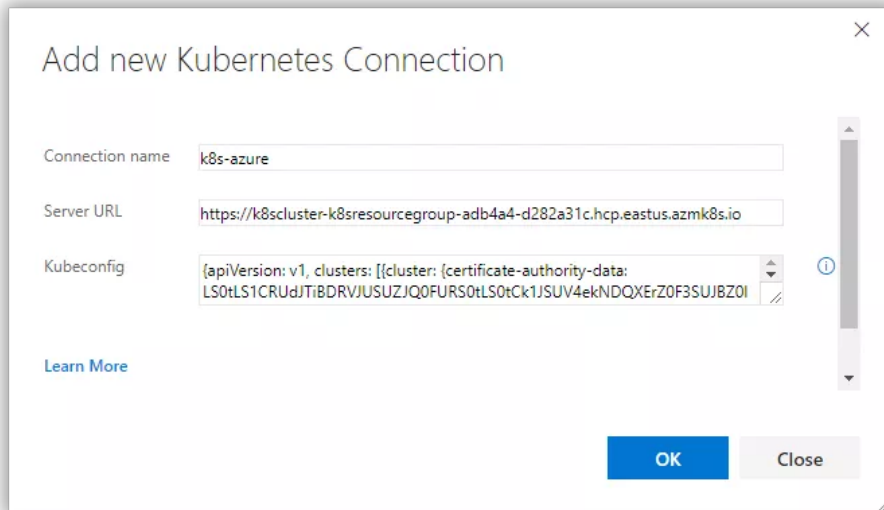
Manager MSDN

PowerShell

PowerShell **DSC**

Python **Raspberry**

Pi **Release**



Configure a CI Build

The first step to deploying containers to a Kubernetes cluster is to configure a CI build that creates a container and then pushes the container to a Docker registry—Azure Container Registry in this case.

Create a Sample App

- Within an existing Team Project create a new Git repository (Code > \$current repository\$ > New repository) called k8s-aspnetcore. Feel free to select the options to add a README and a VisualStudio .gitignore.
- Clone this repo on your development workstation:
 - Open PowerShell at the desired root folder.
 - Copy the URL from the VSTS code view of the new repository.

Management
Release
Management
for Visual
Studio Selenium

SonarQube SQL Server

SQL Server Reporting Services

SSDT Team
Foundation
Server

Test Impact

Analysis TFBUILD Version

Control Virtual Machine

Visual Studio

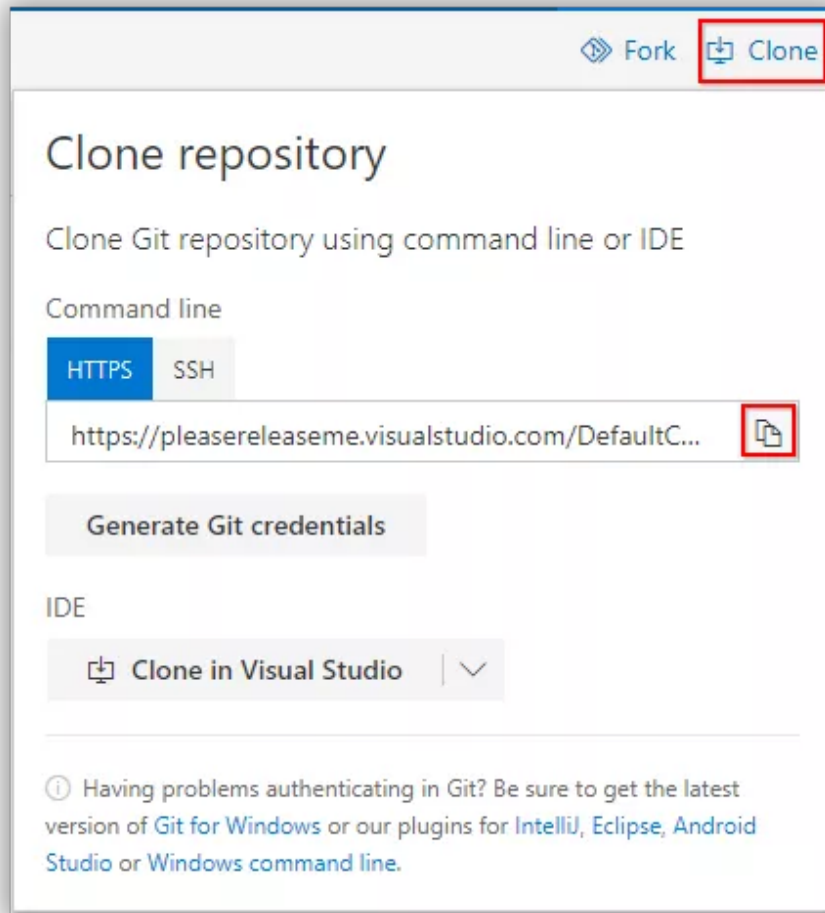
Visual Studio

Online Visual
Studio Team
Services vsts

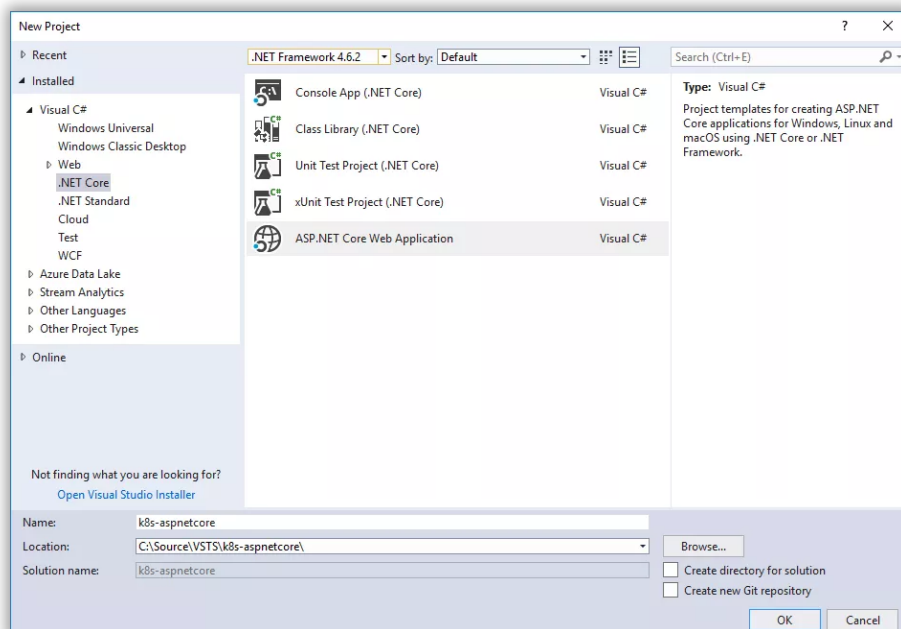
Windows Server Windows

Subsystem for Linux

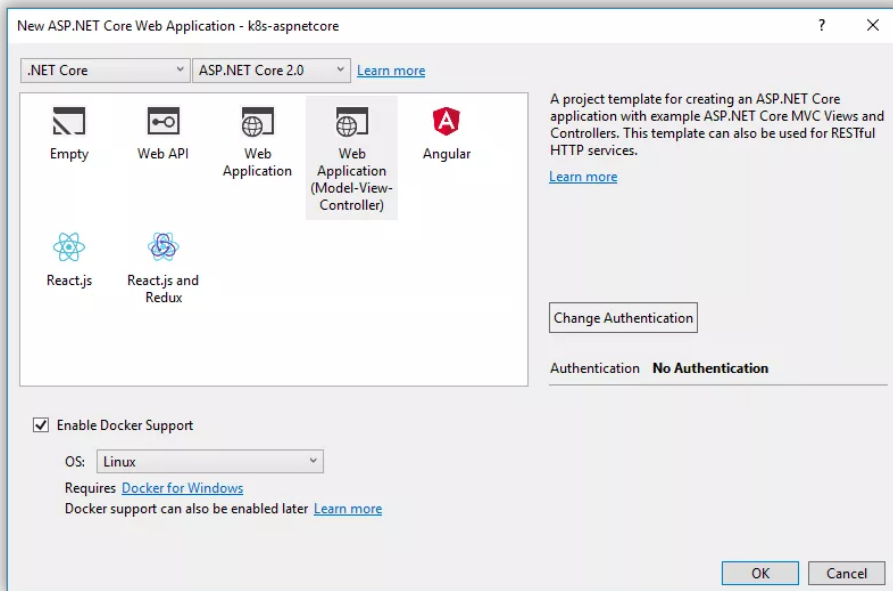
WordPress



- At the PowerShell prompt execute git clone along with the pasted URL.
- Make sure Docker for Windows is running.
- In Visual Studio create an ASP.NET Core Web Application in the folder the git clone command created.



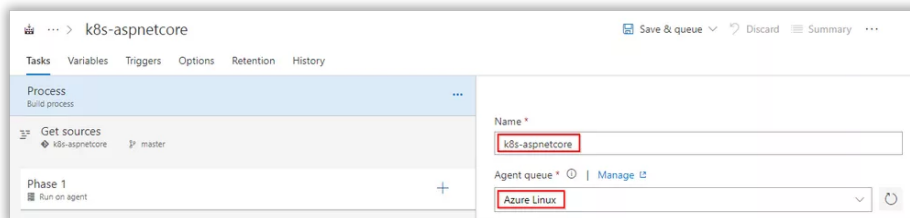
- Choose an MVC app and enable Docker support for Linux.



- You should now be able to run your application using the green Docker run button on the Standard toolbar. What is interesting here is that the build process is using a multi-stage Dockerfile, ie the tooling to build the application is running from a Docker container. See Steve Lasker's post [here](#) for more details.
- In the root of the repository folder create a folder named `k8s-config`, which we'll use later to store Kubernetes configuration files. In Visual Studio create a **New Solution Folder** with the same name and back in the file system folder create empty files named `service.yaml` and `deployment.yaml`. In Visual Studio add these files as existing items to the newly created solution folder.
- The final step here is to commit the code and sync it with VSTS.

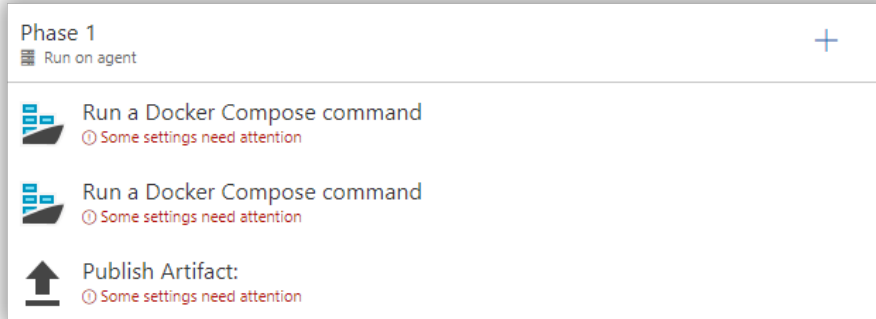
Create a VSTS Build

- In VSTS create a new build based on the repository created above and start with an empty process.
- After the wizard stage of the setup supply an appropriate name for the build and select the **Agent queue** created above if you are using the recommended private agent or **Hosted Linux Preview** if not.



- Go ahead and perform a **Save & queue** to make sure this initial configuration succeeds.

- In the **Phase 1** panel use + to add two **Docker Compose** tasks and one **Publish Build Artifacts** task.



- If you want to be able to perform a **Save & queue** after configuring each task (recommended) then right-click the second and third tasks and disable them.
- Configure the first Docker Compose task as follows:
 - Display name = Build service images
 - Container Registry Type = Azure Container Registry
 - Azure subscription = [name of Azure Resource Manager endpoint created above]
 - Azure Container Registry = [name of Azure Container Registry created above]
 - Docker Compose File = `**/docker-compose.yml`
 - Project Name = `$(Build.Repository.Name)`
 - Qualify Image Names = checked
 - Action = Build service images
 - Additional Image Tags = `$(Build.BuildId)`
 - Include Latest Tag = checked
- Configure the second Docker Compose task as follows:
 - Display name = Push service images
 - Container Registry Type = Azure Container Registry
 - Azure subscription = [name of Azure Resource Manager endpoint created above]
 - Azure Container Registry = [name of Azure Container Registry created above]
 - Docker Compose File = `**/docker-compose.yml`
 - Project Name = `$(Build.Repository.Name)`
 - Qualify Image Names = checked
 - Action = Push service images
 - Additional Image Tags = `$(Build.BuildId)`
 - Include Latest Tag = checked
- Configure the Publish Build Artifacts task as follows:
 - Display name = Publish k8s config
 - Path to publish = k8s-config (this is the folder we created earlier in the repository root folder)

- Artifact name = k8s-config
- Artifact publish location = Visual Studio Team Services/TFS
- Finally, in the **Triggers** section of the build editor check **Enable continuous integration** so that the build will trigger on a commit from Visual Studio.

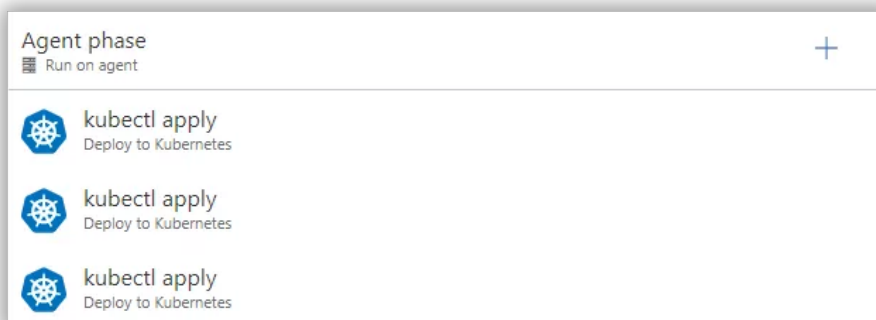
So what does this build do? The first Docker Compose task uses the `docker-compose.yml` file to work out what images need building as specified by `Dockerfile` file(s) for different services. We only have one service (`k8s-aspnetcore`) but there could (and usually would) be more. With the image built on the VSTS agent the second Docker Compose task pushes the image to the Azure Container Registry. If you navigate to this ACR in the Azure portal and drill in to the **Repositories** section you should see your image. The build also publishes the yaml configuration files needed to deploy to the cluster.

Configure a Release Pipeline

We are now ready to configure a release to deploy the image that's hosted in ACR to our Kubernetes cluster. Note that you'll need to complete all of this section before you can perform a release.

Create a VSTS Release Definition

- In VSTS create a new release definition, starting with an empty process and changing the name to `k8s-aspnetcore`.
- In the **Artifacts** panel click on **Add artifact** and wire-up the build we created above.
- With the build now added as an artifact click on the lightning bolt to enable the **Continuous deployment** trigger.
- In the default **Environment 1** click on **1phase, 0 task** and in the **Agent phase** click on **+** to create three **Deploy to Kubernetes** tasks.



- Configure the first **Deploy to Kubernetes** task as follows:
 - Display name = Create Service
 - Kubernetes Service Connection = [name of Kubernetes Service Connection endpoint created above]
 - Command = apply
 - Use Configuration files = checked

- Configuration File = `$(System.DefaultWorkingDirectory)/k8s-aspnetcore/k8s-config/service.yaml`
- Container Registry Type = Azure Container Registry
- Azure subscription = [name of Azure Resource Manager endpoint created above]
- Azure Container Registry = [name of Azure Container Registry created above]
- Secret name [any secret word of your choosing, to be used consistently across all tasks]
- Configure the second Deploy to Kubernetes task as follows:
 - Display name = Create Deployment
 - Kubernetes Service Connection = [name of Kubernetes Service Connection endpoint created above]
 - Command = apply
 - Use Configuration files = checked
 - Configuration File = `$(System.DefaultWorkingDirectory)/k8s-aspnetcore/k8s-config/deployment.yaml`
 - Container Registry Type = Azure Container Registry
 - Azure subscription = [name of Azure Resource Manager endpoint created above]
 - Azure Container Registry = [name of Azure Container Registry created above]
 - Secret name [any secret word of your choosing, to be used consistently across all tasks]
- Configure the third Deploy to Kubernetes task as follows:
 - Display name = Update with Latest Image
 - Kubernetes Service Connection = [name of Kubernetes Service Connection endpoint created above]
 - Command = set
 - Arguments = `image deployment/k8s-aspnetcore-deployment k8s-aspnetcore=$yourAcrNameHere$.azurecr.io/k8s-aspnetcore:$(Build.BuildId)`
 - Container Registry Type = Azure Container Registry
 - Azure subscription = [name of Azure Resource Manager endpoint created above]
 - Azure Container Registry = [name of Azure Container Registry created above]
 - Secret name [any secret word of your choosing, to be used consistently across all tasks]
- Make sure you save the release but don't bother testing it out just yet as it won't work.

Create the Kubernetes configuration

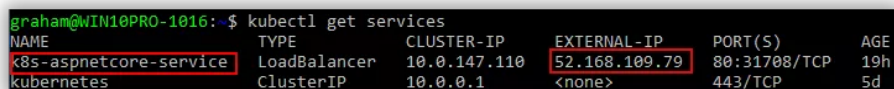
- In Visual Studio paste the following code in to the **service.yaml** file created above.

```
apiVersion: v1
kind: Service
metadata:
  name: k8s-aspnetcore-service
  labels:
    version: test
spec:
  selector:
    app: k8s-aspnetcore
  ports:
    - port: 80
  type: LoadBalancer
```

- Paste the following code in to the **deployment.yaml** file created above. The code is for my ACR so you will need to amend accordingly.

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: k8s-aspnetcore-deployment
spec:
  replicas: 4
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
  minReadySeconds: 5
  template:
    metadata:
      labels:
        app: k8s-aspnetcore
    spec:
      containers:
        - name: k8s-aspnetcore
          image: prmcra.azurecr.io/k8s-aspnetcore
          ports:
            - containerPort: 80
          imagePullSecrets:
            - name: prmk8s
```

- You can now commit these changes and then head over to VSTS to check that the release was successful.
- If the release was successful you should be able to see the ASP.NET Core website in your browser. You can find the IP address by executing **kubectl get services** from wherever you installed kubectl.



```
graham@WIN10PRO-1016: $ kubectl get services
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
k8s-aspnetcore-service	LoadBalancer	10.0.147.110	52.168.109.79	80:31708/TCP	19h
kubernetes	ClusterIP	10.0.0.1	<none>	443/TCP	5d

- Another command you might try running is **kubectl describe deployment \$nameOfYourDeployment**, where **\$nameOfYourDeployment** is the `metadata > name` in **deployment.yaml**. A quick tip here is that if you only have one deployment you only need to type the first letter of it.

- It's worth noting that splitting the **service** and **deployment** configurations in to separate files isn't necessarily a [best practice](#) however I'm doing it here to try and help clarify what's going on.

In terms of a very high level explanation of what we've just configured in the release pipeline, for a simple application such as an ASP.NET Core website we need to deploy two key objects:

1. A Kubernetes [Service](#) which (in our case) is configured with an external IP address and acts as an abstraction layer for Pods which are killed off and recreated every time a new release is triggered. This is handled by the first **Deploy to Kubernetes** task.
2. A Kubernetes [Deployment](#) which describes the nature of the deployment—number of Pods (via Replica Sets), how they will be upgraded and so on. This is handled by the second **Deploy to Kubernetes** task.

On first deployment these two objects are all that is needed to perform a release. However, because of the declarative nature of these objects they do nothing on subsequent release if they haven't changed. This is where the third **Deploy to Kubernetes** task comes in to play—ensuring that after the first release subsequent releases do cause the container to be updated.

Wrapping Up

That concludes our initial look at CI/CD with VSTS and Azure Container Service (AKS)! As I mentioned at the beginning of the post I've purposely tried to keep this walkthrough as simple as possible, so watch out for the next installment where I'll build on what I've covered here.

Cheers—Graham

Share this:

Share 5

Share

Tweet

G+

38

▲ ▼

Save 1

Email

Print

Posted in **Continuous Delivery with Containers**

Tags: [Azure CLI](#) [Docker](#) [kubectl](#) [Kubernetes](#)
[Visual Studio Team Services](#) [VSTS](#)
[Windows Subsystem for Linux](#)

5 Comments **pleasereleaseme** **Login** ▾ **Recommend** **Tweet** **Share****Sort by Best** ▾

Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS 

Name

**virasana** • 6 months ago

Hi Graham,

When I run my release, on the task "Create Service", I get:

Error loading config file

"/tmp/kubectITask/1526733061812/config": [pos 210]: json: error
decoding base64 binary

'LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUV4ekNC
illegal base64 data at input byte 135

It doesn't pick up the config file - I suspect it might be finding an
empty one, or possibly an encoding error? It's there (and non-
empty) on the Git repository, and it's there in the artifact when I
browse using the ellipsis on the Create Service task.

In the "Create Service" task, I am using:

\$(System.DefaultWorkingDirectory)/_MyArtifact/k8s-
config/service.yaml

Any ideas?

Many Thanks!

^ | ▾ • Reply • Share ›

**Graham Smith** Mod ➔ virasana • 6 months ago

Hi Jean-Pierre

Could possibly be an (illegal) tab character. I've been
bitten by this a few times and have taken to turning on
'show whitespace' in VS when working with JSON. I'm
away from my workstation right now but I think the option
is somewhere on the Edit menu.

Cheers and good to hear from you - Graham

^ | ▾ • Reply • Share ›



SuperJohn140 • 7 months ago

when i try and create this in kubernetes the deployment results in this error:

"Failed to pull image "pmcr.azurecr.io/k8s-aspnet...": rpc error: code = Unknown desc = Error response from daemon: Get https://pmcr.azurecr.io/v2... unauthorized: authentication required

Error syncing pod"

should "pmcr" be my acr?

and do i need to modify the secret name in the deployment.yaml?

i have mine very close but now im just stuck on this authorized pull

^ | v • Reply • Share ›



Graham Smith Mod ➔ SuperJohn140 • 7 months ago

Hi John

The pmcr.azurecr.io bit is the FQDN of the Azure Container Registry. This is an ACR that I've created so the pmcr bit is DNS unique (and private) to me. So yes, you need to create your own ACR and replace the pmcr bit with the name of your ACR.

The secret name can be the same as my example but beware a possible bug (which existed at the time of writing the post) if using a non-default namespace which I discuss in part 2. Do read about that to understand what's going on behind the scenes.

Good luck and do post back if you still have problems or even better with the good news that you have it working.

Cheers - Graham

1 ^ | v • Reply • Share ›



SuperJohn140 ➔ Graham Smith • 6 months ago

Hi Graham

Currently I'm working in the default namespace but will make sure to read part 2.

I made those changes but i had to do some more work with the secret. i don't think mine is using the secret i provided in all my kubernetes release tasks.

I generated my own using kubectl.

kubectl create secret docker-registry


```
acrconnection —docker-  
server=https://myacr.azurecr.io —docker-  
username=/*my acr access username*/ —docker-  
password=/*my acr access password*/ —docker-  
email=/*some email*/
```

An issue i had with this is that the documentation i was reading didn't explicitly say to get the username and password from my ACR and not use my docker credentials (or maybe i just didn't read into it enough) so i messed up the first time.

Thanks

^ | v • Reply • Share ›

Powered by WordPress, Theme by Andrew Dyer