Please Release Me

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

Search here ...

Deploy a Dockerized ASP.NET Core Application to Kubernetes on Azure Using a VSTS CI/CD Pipeline: Part 2

Posted by Graham Smith on March 21, 2018 | 0 Comments

If you need to provision a new environment for your deployment pipeline, what's your process and how long does it take? For many of us the process probably starts with a request to an infrastructure team for new virtual machines. If the new VMs are in Azure the request might be completed quite quickly; if they are on premises it might take much longer. In both scenarios you might have to justify your request: there will be actual cost in Azure and on premises it's another chunk of the datacentre 'gone'.

With the help of containers and container orchestrators I predict (and sincerely hope) that this sort of pain will become a distant memory for much of the software development community for whom it is currently an issue. The reason is that container orchestration technologies abstract away the virtual (or physical) server layer and allow you to focus on configuring services and how they communicate with each other—all through configuration files. The only time you'd need to think of virtual (or physical) servers is if the cluster running your orchestrator needed more capacity, in which case someone will need to add more nodes. A whole new

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

environment for your pipeline just by doing some work with a configuration file? What's not to like?

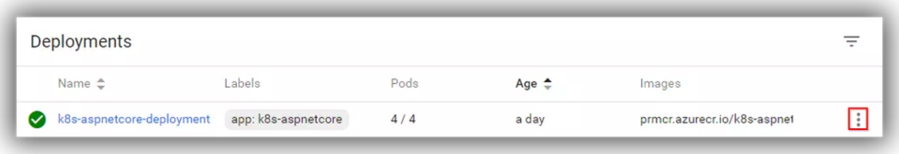
In this blog post I hope to make my prediction come alive by showing you how new environments can be quickly created using Kubernetes running in Microsoft's Azure Container Service (AKS), crucially using declarative configuration files that get deployed as part of a VSTS release pipeline. This post follows directly on from a previous post, both in terms of understanding and also the components that were built in that first post, so if you haven't already done so I recommend working your way through that post before going further.

Housekeeping

In the previous post we deployed to the **default** namespace so it probably makes sense to clean all this up. This can all be done by the command line of course but to mix it up a bit I'll illustrate using the Kubernetes Dashboard. You can start the dashboard using the following command, substituting in the name of your resource group and the name of the cluster:

```
az aks browse --resource-group <resource-group> --name <cluster-name>
```

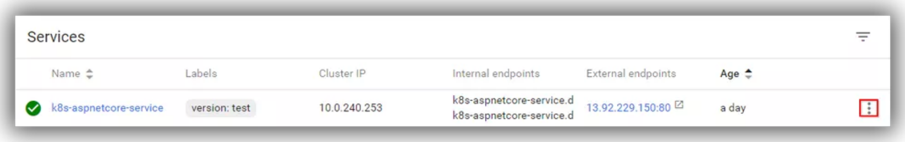
This should open the dashboard in a browser displaying the **default** namespace. Navigate to **Workloads** > **Deployments** and using the hamburger menu delete the deployment:



Deployments

Name	Labels	Pods	Age	Images
✓ k8s-aspnetcore-deployment	app: k8s-aspnetcore	4 / 4	a day	prmcr.azurecr.io/k8s-aspnet

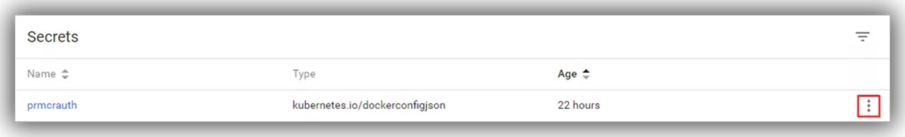
Navigate to **Discovery and Load Balancing** > **Services** and delete the service:



Services

Name	Labels	Cluster IP	Internal endpoints	External endpoints	Age
✓ k8s-aspnetcore-service	version: test	10.0.240.253	k8s-aspnetcore-service.d k8s-aspnetcore-service.d	13.92.229.150:80	a day

Navigate to **Config and Storage** > **Secret** and delete the secret:



Secrets

Name	Type	Age
prmcrauth	kubernetes.io/dockerconfigjson	22 hours

Environments and Namespaces

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

The Kubernetes feature that we'll use to create environments that together form part of our pipeline is Namespaces. You can think of namespaces as a way to divide the Kubernetes cluster in to virtual clusters. Within a namespace resource names need to be unique but they don't have to be across namespaces. This is great because effectively we have network isolation so that across each environment resource names stay the same. Say goodbye to having to append the environment name to all the resources in your environment to make them unique.

In this post I'll make a pipeline consisting of two environments. I'm sticking with a convention I established several years ago so I'll be creating DAT (developer automated test) and PRD (production) environments. In a complete pipeline I might also create a DQC (developer quality control) environment to sit between DAT and PRD but that won't really add anything extra to this exercise.

First up is to create the namespaces. There is an argument for saying that namespace creation should be part of the release pipeline however in this post I'm going to create everything manually as I think it helps to understand what's going on. Create a file called **namespaces.yaml** and add the following contents:

```
apiVersion: v1
kind: Namespace
metadata:
  name: dat
---
apiVersion: v1
kind: Namespace
metadata:
  name: prd
```

Note that namespace **name** needs to be in lower case as it needs to be DNS compatible. Open a command prompt at the same location as **namespaces.yaml** and execute the the following command: **kubectl create -f namespaces.yaml**. You should get a message back advising the namespaces have been created and at one level that's all there is to it. However there's a couple of extra bits worth knowing.

When you first start working with **kubectl** at the command line you are working in the **default** namespace. To work with other namespaces needs some configuration.

To return details of the configuration stored in **C:\Users\<username>\.kube\config** use:

```
kubectl config view
```

My cluster returned the following output:

```
apiVersion: v1
clusters:
- cluster:
```

Follow via Email

Email Address

Subscribe

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

```
    certificate-authority-data: REDACTED
    server: https://k8scluster-k8sresourcegroup-adb4a4-d282a31c.hcp.eastus.a
zmk8s.io:443
  name: k8sCluster
contexts:
- context:
    cluster: k8sCluster
    user: clusterUser_k8sResourceGroup_k8sCluster
  name: k8sCluster
current-context: k8sCluster
kind: Config
preferences: {}
users:
- name: clusterUser_k8sResourceGroup_k8sCluster
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED
    token: aa16af4290c8b372d6b8812222dedd69
```

From this output you need to determine your cluster name (which you probably already know) as well as the name of the user. These details are fed in to the following command for creating a new context for an environment (in this case the DAT environment):

```
kubectl config set-context dat --namespace=dat --cluster=k8sCluster --user=c
lusterUser_k8sResourceGroup_k8sCluster
```

To switch to working to this context (and hence the **dat** namespace) use:

```
kubectl config use-context dat
```

To confirm (or check) the current context use:

```
kubectl config current-context
```

To get back to the default namespace use:

```
kubectl config use-context <name-of-cluster>
```

Normally that would be most of what you need to know to work with namespaces, however as of the time of writing there is a bug in the VSTS **Deploy to Kubernetes** task which requires some extra work. The bug may be fixed by the time you read this however it's handy to examine the issue to further understand what is going on behind the scenes.

Each namespace needs to access the Azure Container Registry (ACR) we created in the previous post to pull down images. This is a private registry so we don't want open access and so some form of authentication is required. This is provided by the creation of a Kubernetes secret that holds the authentication details to the ACR. The VSTS **Deploy to Kubernetes** task can create this secret for us however the bug is that it only creates the secret for the **default** namespace and fails to create the secret when a different namespace is specified. The workaround is to create the secret manually in each namespace using the following command:

```
kubectl create secret docker-registry <secret-name> --namespace=<namespace>
--docker-server=<acr-name>.azurecr.io --docker-username=<acr-name> --docker-
```

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

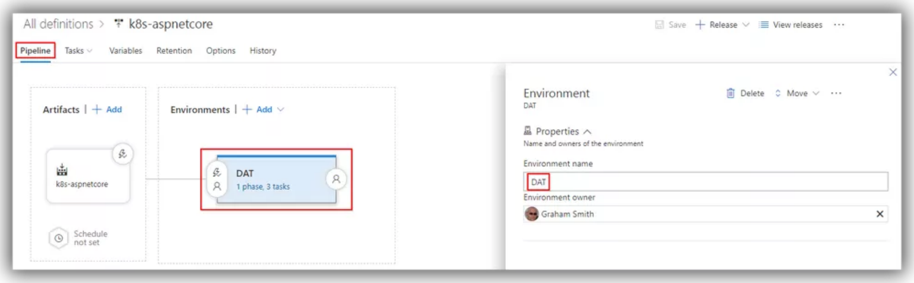
```
password=<acr-admin-password> --docker-email=<any-valid-email-address>
```

In the above command **secret-name** is any arbitrary name you choose for the secret, **namespace** is the namespace in which to create the secret, **acr-name** is the name of your ACR, **acr-admin-password** is the password from the **Access** keys panel of your ACR and **any-valid-email-address** is just that. You'll need to run this command for each namespace of course. One final thing: you'll need to make sure that in the codebase the **imagePullSecrets** name in **deployment.yaml** matches the name of the secret you just created.

Amend the VSTS Pipeline to Support Multiple Environments

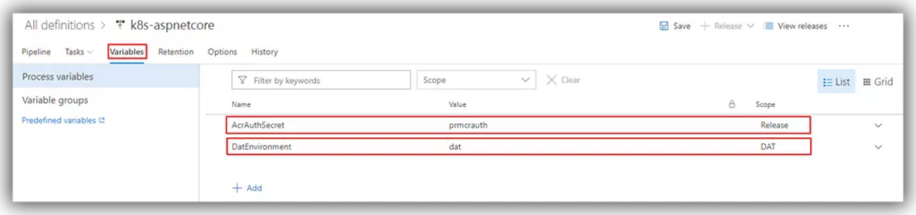
In this section we amend the release pipeline that was built in the previous post to support multiple environments.

1. In the **Pipeline** tab rename **Environment 1** to **DAT**:



Environment name: DAT
Environment owner: Graham Smith

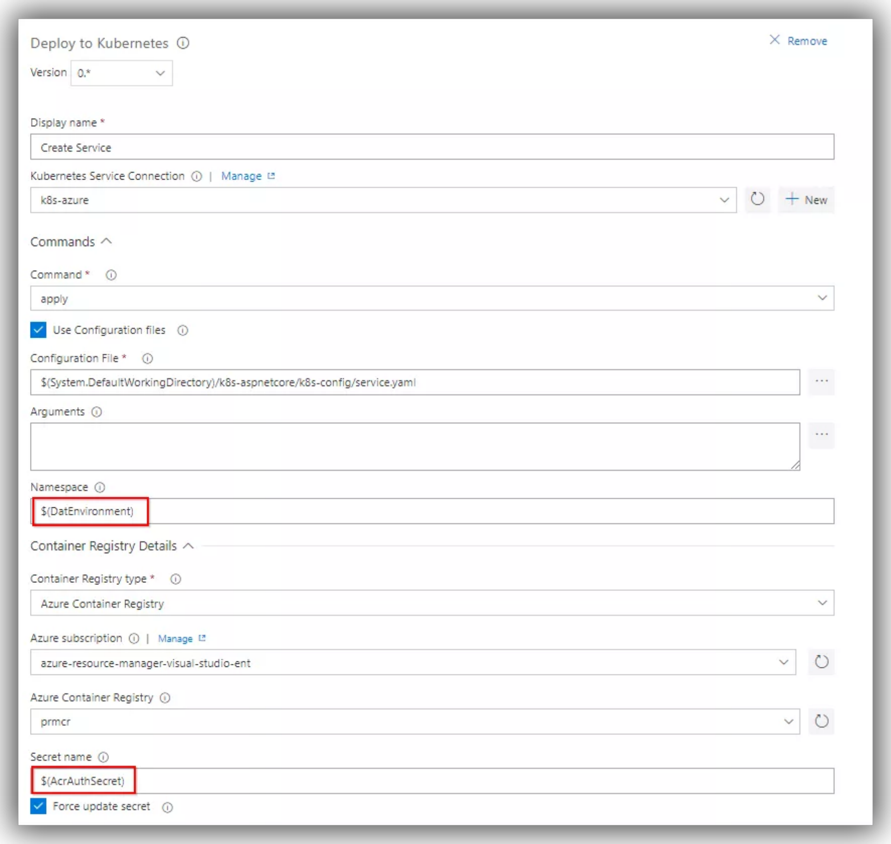
2. In the **Variables** tab create a variable to hold the name of the secret created above to authenticate with ACR. Create a second variable for the **DAT** environment namespace and change its scope to **DAT**. Remember that the value needs to be lower case:



Name	Value	Scope
AcrAuthSecret	prmcrauth	Release
DatEnvironment	dat	DAT

3. In the **Tasks** tab amend all three **Deploy to Kubernetes** tasks so that the **Namespace** field contains the **$(DatEnvironment)** variable. At the same time ensure that **Secret name** field matches the name of the secret variable created above:

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

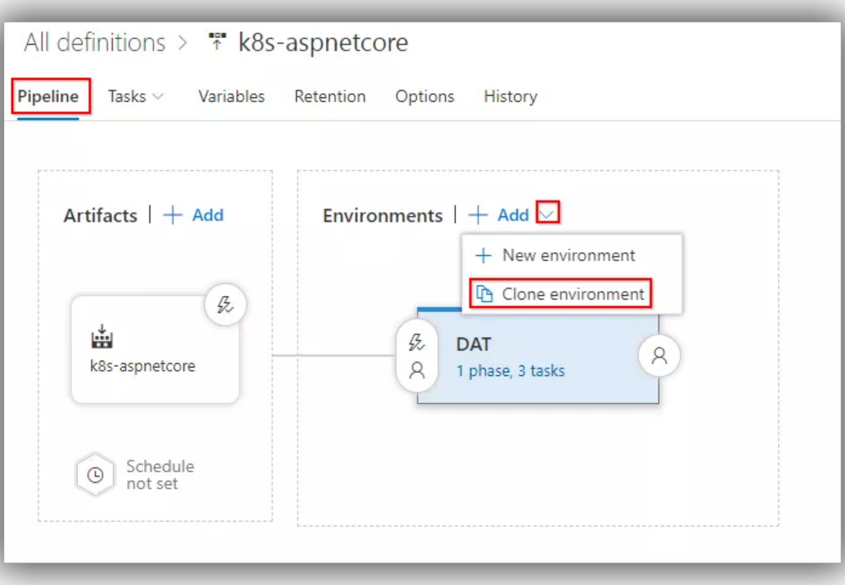

Deploy to Kubernetes
Version 0.*
Display name * Create Service
Kubernetes Service Connection k8s-azure
Commands
Command * apply
✓ Use Configuration files
Configuration File * $(System.DefaultWorkingDirectory)/k8s-aspnetcore/k8s-config/service.yaml
Arguments
Namespace $(DatEnvironment)
Container Registry Details
Container Registry type * Azure Container Registry
Azure subscription azure-resource-manager-visual-studio-ent
Azure Container Registry prmcr
Secret name $(AcrAuthSecret)
✓ Force update secret

4. In order to test that deploying to DAT works, either trigger a build or, if you updated **deployment.yaml** above on your workstation commit your code. If the deployment was successful find the external IP address of the **LoadBalancer** by executing **kubectl get services --namespace=dat** and paste in to a browser to confirm that the ASP.NET Core website is running.

Amend the VSTS Pipeline to Support a New Environment

Now for the fun bit where we see just how easy it is to configure a new, network-isolated environment.

1. In the **Pipeline** tab use the arrow next to **Environments** > **Add** to show and then select **Clone environment**:



2. Rename the cloned environment to **PRD**. Create a new variable (ie **PrdEnvironment**) scoped to PRD to hold the **prd** namespace and amend each of the three **Deploy to Kubernetes** tasks so that the **Namespace** field contains the **$(PrdEnvironment)** variable.
3. Trigger a build and check the deployment was successful by executing **kubectl get services --namespace=prd** to get the external IP address of the **LoadBalancer** which you can paste in to a browser to confirm that the ASP.NET Core website is running.

And That's It!

Yep—that really is all there is to it! Okay, this is just a trivial example, however even with more services the procedure would be the same. Granted, in a more complex application there might be some environment variables or secrets that might change but even so, it's just configuration.

I'm thrilled by the power that Kubernetes gives to developers—no more thinking about VMs or tin, no more having to append resources with environment names, and the ability to create a new environment in the blink of an eye—wow!

There's lots more I'm planning to cover in the deployment pipeline space however next time I'll be looking at the development inner loop and the options for running Kubernetes whilst developing code.

Cheers—Graham

Share this:

Share 0 Share Tweet G+ 7

Save Email Print

Posted in Continuous Delivery with Containers

Tags: kubectl Kubernetes namespaces Visual Studio Team Services VSTS