What Is Azure Kubernetes Service?

Lecture: What Is Kubernetes?

A basic understanding of Kubernetes will help you significantly in this course. In this lesson, we'll go over what problems Kubernetes solves, and why it's important when publishing our application.

Lecture: Managed Kubernetes

In this lesson, we'll discuss the differences between a self-hosted Kubernetes cluster and a managed Kubernetes cluster. This will allow us to see the benifits of AKS, in taking reposibility away from the cluster administrator.

Building Your AKS Cluster

Lecture: Container Registries and Container Instances

Azure Container Registry is a private image repository, allowing you to use images stored in that repository for your pods in AKS. Azure Container Instances can be used in conjuction with AKS to build virtual nodes (see the "Virtual Kubelet" lesson in this course). In this lesson, we'll define ACR and ACI and how they are used in the context of AKS.

Commands Issued in This Lesson

```
# Create a resource group
```

```
az group create --name myaks-rg --location eastus

# Create a service principal:
az ad sp create-for-rbac --skip-assignment

# Copy the app ID and password from the output:
{
  "appId": "4631b7ff-bd2a-4269-80b6-218fc694c0a7",
  "password": "5c242d31-3617-4198-b663-f40f4f4a1f05"
}

# Create ACR (must have a unique name):
az acr create --resource-group myaks-rg --name aksdeepdive
--sku Basic --admin-enabled true

# Add permission for our SP to read ACR images:
az role assignment create --assignee "47cf0a9a-ffd6-4630-
a108-65851636af61" --role acrpull --scope '/
subscriptions/....'

# Change directory into cloud drive:
cd $HOME/clouddrive

# Create a Dockerfile:
echo "FROM hello-world" > Dockerfile

# Use ACR tasks to build the image and push it to ACR:
az acr build --image sample/hello-world:v1 --registry
aksdeepdive --file Dockerfile .

# Run the image with a command(cmd) from an image location
to '/dev/null' which is a device file that reports that the
operation succeeded:
az acr run --registry aksdeepdive --cmd '$Registry/sample/
hello-world:v1' /dev/null
```

```
# Push another image to the new ACR:
git clone https://github.com/chadmcrowell/aks-node-
docker.git

cd aks-node-docker/app

# Build and push another image:
az acr build --registry aksdeepdive --image node:v1 .
```

Lecture: Service Principal

A Service Principal is used with an AKS cluster to provide it access to different Azure resources (e.g. to pull images from ACR). In this lesson, we'll go over how to create a service principal using PowerShell and the Azure Portal, which will provide us with multiple ways to troubleshoot if any secrets expire or credentials refresh.

Commands used in this lesson:

```
# Connect to Azure Active Directory
Connect-AzureAD

# List the service principal properties
(Get-AzureADServicePrincipal -filter "DisplayName eq
'azure-cli-2020-02-08-13-26-11'").OAuth2Permissions

# create a new service principal
$sp = New-AzADServicePrincipal -Role Reader

# Convert secret to a binary string
```

```
$BSTR =
[System.Runtime.InteropServices.Marshal]::SecureStringToBST
R($sp.Secret)

# Convert secret into a text string
$UnsecureSecret =
[System.Runtime.InteropServices.Marshal]::PtrToStringAuto($
BSTR)

$UnsecureSecret
```

Lecture: Creating and Accessing an AKS Cluster

Creating an AKS cluster is essential to expound upon the benefit of running Kuberentes in Azure. In this lesson, we will go through the process of creating an AKS cluster in Cloud Shell. Once the cluster has been built, we will run a deployment, and expose that deployment to a load balancer service, which will allow anyone to access our application from the internet.

Commands Used in This Lesson

```
# Create AKS cluster:
az aks create \
--resource-group myaks-rg \
--name myAKSCluster \
--node-count 1 \
--enable-addons monitoring \
--generate-ssh-keys \
--enable-rbac \
--service-principal "c33dfc60-19d6-4d19-9788-217800d87a89"
\
```

```
--client-secret "c780465f-e389-479a-8083-01d0ba3821aa"

# Get the credentials and store in cluster:
az aks get-credentials --resource-group myaks-rg2 --name
myAKSCluster2

# Note: context stored in '~/.kube/config'

# Create a deployment:
kubectl run nodeapp --image=aksdeepdive.azurecr.io/node:v1
--replicas=1 --port=8080

# View the deployments:
kubectl get deploy

# View the pods:
kubectl get po

# View the ReplicaSets:
kubectl get rs

# Write the service YAML to a file:
kubectl expose deploy nodeapp --port=80 --target-port=8080
--dry-run -o yaml > svc.yaml

# Modify the YAML to create a LoadBalancer type service:
yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    run: nodeapp
  name: nodeapp
spec:
  type: LoadBalancer
```

```
  ports:
  - port: 80
    protocol: TCP
    targetPort: 8080
  selector:
    run: nodeapp

# Create the service:
kubectl apply -f svc.yaml

# View the services:
kubectl get svc
```

Credentials and Access

Lecture: Kubernetes Service Accounts

Service Accounts are important for authenticating to the Kubernetes cluster. The default service account is created upon AKS cluster creation, and pods use it to authenticate to the API server via a token. In this lesson, we'll take a look at the way users and pods authenticate to the cluster to make changes to our AKS cluster.

Commands Used in This Lesson

```
# Get cluster list from kubeconfig:
kubectl config get-clusters

# Get the current context:
kubectl config current-context
```

```
# Get all contexts:
kubectl config get-contexts

# Switch between user and admin context:
kubectl config use-context myAKSCluster-admin

# Get cluster administrator credentials:
az aks get-credentials --resource-group myaks-rg --name
myAKSCluster --admin

# View all the service accounts in the default namespace:
kubectl get sa

# View the details of the default service account:
kubectl describe sa default

# View the secrets in the AKS cluster:
kubectl get secrets

# View the secret mounted to a pod:
kubectl exec -it $pod-name sh

# cat /var/run/secrets/kubernetes.io/serviceaccount/token
```

Lecture: Role-Based Access Control (RBAC)

RBAC is how users or pods authenticate to the Kubernetes cluster. In this lesson, we'll go through ClusterRoles and ClusterRoleBindings, which apply to resources at the cluster level. We will also discuss Role and RoleBindings, which are permissions that apply to the resources that reside at the namespace level.

Commands Used in This Lesson:

```
# View the cluster kubeconfig:
kubectl config view

# View cluster role bindings:
kubectl get clusterrolebinding

# View the cluster role binding for cluster admin:
kubectl describe clusterrolebinding cluser-admin

# View the cluster role for cluster admin:
kubectl describe clusterrole cluser-admin

# View namespaced resources (pods, services, deployments,
etc.):
kubectl api-resources --namespaced=true

# View cluster-level resources (persistent volumes, nodes,
etc.):
kubectl api-resources --namespaced=false

# Get roles in kube-system namespace:
kubectl get roles -n kube-system

# Describe the controller manager role:
kubectl describe role system::leader-locking-kube-
controller-manager -n kube-system

# Get rolebindings in kube-system namespace:
kubectl get rolebinding -n kube-system
```

```
# Describe role binding for controller manager:
kubectl describe rolebinding system::leader-locking-kube-
controller-manager -n kube-system
```

Directory Access

Lecture: Creating an AKS Cluster with Active Directory Integration

The really great thing about Kubernetes on Azure is how easy it is to integrate Active Directory. This fully-integrated identity management tool can be installed with your AKS cluster, and that's exactly what we'll do in this lesson.

```
# Create a variable for your desired AKS cluster name:
aksname = "myAKSCluster"

# Create an Azure AD application to act as an endpoint for
identity requests:
serverApplicationId=$(az ad app create \
    --display-name "${aksname}Server" \
    --identifier-uris "https://${aksname}Server" \
    --query appId -o tsv)

# Update the application group membership claims:
az ad app update --id $serverApplicationId --set
groupMembershipClaims=All

# Create a service principal for the server app:
az ad sp create --id $serverApplicationId

# Get the SP secret:
```

```
serverApplicationSecret=$(az ad sp credential reset \
    --name $serverApplicationId \
    --credential-description "AKSPassword" \
    --query password -o tsv)

# Add permissions for Azure AD to read directory data and
sign in user profiles:
az ad app permission add \
    --id $serverApplicationId \
    --api 00000003-0000-0000-c000-000000000000 \
    --api-permissions e1fe6dd8-
ba31-4d61-89e7-88639da4683d=Scope
06da0dbc-49e2-44d2-8312-53f166ab848a=Scope 7ab1d382-
f21e-4acd-a863-ba3e13f7da61=Role

# Grant permission for the server application.
# Note: You must add this part in the portal.

# Create a Client AD app for client:
clientApplicationId=$(az ad app create \
    --display-name "${aksname}Client" \
    --native-app \
    --reply-urls "https://${aksname}Client" \
    --query appId -o tsv)



# Create an SP for the client:
az ad sp create --id $clientApplicationId

# Get the oAuth2 ID for the server app to allow the
authentication flow between the two app components (not
required):
oAuthPermissionId=$(az ad app show --id
$serverApplicationId --query "oauth2Permissions[0].id" -o
tsv)
```

```
# Add the permissions for the client application and server
application components to use the oAuth2 communication
flow.
# Note: You must do this part in the portal.

# Get the tenant ID:
tenantId=$(az account show --query tenantId -o tsv)

# Create the AKS cluster:
az aks create \
    --resource-group myResourceGroup \
    --name $aksname \
    --node-count 1 \
    --generate-ssh-keys \
    --aad-server-app-id $serverApplicationId \
    --aad-server-app-secret $serverApplicationSecret \
    --aad-client-app-id $clientApplicationId \
    --aad-tenant-id $tenantId \
    --service-principal
"c33dfc60-19d6-4d19-9788-217800d87a89" \
    --client-secret "c780465f-e389-479a-8083-01d0ba3821aa"
```

Lecture: Authenticating to the AKS Cluster Using Azure Active Directory

Now that we've created an AKS cluster with Active Directory Integration, we can create our user groups and users to authenticate to the AKS cluster. In this lesson we'll go through a scenario in which Alice from the dev team needs access to the cluster. As the cluster admin, we'll create the role and role binding to only allow Alice access to the `dev` namespace. We'll test that access by assuming Alice's role.

Commands used in this lesson:

```
# Get the resource ID of the AKS cluster and assign it to
the '$aks_id' variable:
AKS_ID=$(az aks show \
    --resource-group myaks-rg \
    --name myAKSCluster \
    --query id -o tsv)

# Create a dev group in Azure AD:
APPDEV_ID=$(az ad group create --display-name appdev --
mail-nickname appdev --query objectId -o tsv)

# Create an Azure role assignment for the Azure ad group:
az role assignment create \
  --assignee $APPDEV_ID \
  --role "Azure Kubernetes Service Cluster User Role" \
  --scope $AKS_ID

# Create the 'Alice' user:

AKSDEV_ID=$(az ad user create \
  --display-name "Alice Developer" \
  --user-principal-name alice@buildazurepipelines.com \
  --password P@ssw0rd1 \
  --query objectId -o tsv)

# Add the 'Alice' user to the dev group:
az ad group member add --group appdev --member-id
$AKSDEV_ID

# Get admin credentials to create role:
az aks get-credentials --resource-group myaks-rg --name
myAKSCluster --admin
```

```
# Create a 'dev' namespace:
kubectl create namespace dev

# Create a role in the 'dev' namespace:
kubectl create role dev-user-full-access --verb=* --
resource=* -n dev

# Get the group object ID:
az ad group show --group dev --query objectId -o tsv

# Create a role binding:
kubectl create rolebinding dev-group-full-access --
role=dev-user-full-access --group="32b7adef-e03d-4120-ba81-
b231f289e414" -n dev

# Reset the kubeconfig:
az aks get-credentials --resource-group myaks-rg --name
myAKSCluster --overwrite-existing

# Create a deployment in the 'dev' namespace:
kubectl run --generator=run-pod/v1 nginx-dev --image=nginx
-n dev

# View pods created in deployment:
kubectl get pods -n dev

# Try to view pods outside of the 'dev' namespace:
kubectl get pods --all-namespaces
```

```
# Get the resource ID of the AKS cluster and assign it to
the '$aks_id' variable:
AKS_ID=$(az aks show \
    --resource-group myaks-rg \
    --name myAKSCluster \
    --query id -o tsv)
```

```
# Create a dev group in Azure AD:
APPDEV_ID=$(az ad group create --display-name appdev --
mail-nickname appdev --query objectId -o tsv)

# Create an Azure role assignment for the Azure ad group:
az role assignment create \
  --assignee $APPDEV_ID \
  --role "Azure Kubernetes Service Cluster User Role" \
  --scope $AKS_ID

# Create the 'Alice' user:

AKSDEV_ID=$(az ad user create \
  --display-name "Alice Developer" \
  --user-principal-name alice@buildazurepipelines.com \
  --password P@ssw0rd1 \
  --query objectId -o tsv)

# Add the 'Alice' user to the dev group:
az ad group member add --group appdev --member-id
$AKSDEV_ID

# Get admin credentials to create role:
az aks get-credentials --resource-group myaks-rg --name
myAKSCluster --admin

# Create a 'dev' namespace:
kubectl create namespace dev

# Create a role in the 'dev' namespace:
kubectl create role dev-user-full-access --verb=* --
resource=* -n dev

# Get the group object ID:
```

```
az ad group show --group dev --query objectId -o tsv

# Create a role binding:
kubectl create rolebinding dev-group-full-access --
role=dev-user-full-access --group="32b7adef-e03d-4120-ba81-
b231f289e414" -n dev

# Reset the kubeconfig:
az aks get-credentials --resource-group myaks-rg --name
myAKSCluster --overwrite-existing

# Create a deployment in the 'dev' namespace:
kubectl run --generator=run-pod/v1 nginx-dev --image=nginx
-n dev

# View pods created in deployment:
kubectl get pods -n dev

# Try to view pods outside of the 'dev' namespace:
kubectl get pods --all-namespaces
```

Network Models

Lecture: The Kubenet Network Plugin

The kubenet plugin for AKS is the suggested plugin to use because it allows for a greater number of pods to be used in your cluster. In this lesson, we'll go through creating an AKS cluster with the `kubenet` plugin.

Commands Used in This Lesson:

```
# Get the subnet ID and store it in the 'SUBNET_ID'
```

```
variable:
SUBNET_ID=$(az network vnet subnet show --resource-group
myaksrg --vnet-name myAKSVnet --name myAKSSubnet --query id
-o tsv)

# Create the AKS cluster and explicitly use the kubenet
network plugin:
az aks create \
    --resource-group myaksrg \
    --name kubenetCluster \
    --node-count 3 \
    --network-plugin kubenet \
    --service-cidr 172.0.0.0/16 \
    --dns-service-ip 172.0.0.10 \
    --pod-cidr 172.244.0.0/16 \
    --docker-bridge-address 172.99.0.1/16 \
    --vnet-subnet-id $SUBNET_ID \
    --service-principal "63e3dd7e-ff67-4bb7-
a0fb-8b077e6acbcd" \
    --client-secret "a040ae44-9036-4ef9-a898-233782ab5406"

# Connect to your cluster:
az aks get-credentials --resource-group myaksrg --name
kubenetCluster

# View the node IP addresses:
kubectl get node -o wide

# Create a pod:
kubectl run --generator=run-pod/v1 nginx-dev --image=nginx

# View the pod IP addresses:
kubectl get po -o wide
```

Lecture: The Azure CNI Network Plugin

The Azure CNI allows for more control over your address spaces but comes with limitations. If you have fewer IP addresses to use for your pods, then you'll have to do a bit more planning up front in order to plan for scale.

Commands Used in This Lesson

```
# Create new resource group:
az group create -l eastus -n myaksrg



# Create vnet:
az network vnet create -g myaksrg -n MyVnet --address-
prefix 10.0.0.0/16 --subnet-name default --subnet-prefix
10.0.0.0/21




# Get subnet resource ID:
az network vnet subnet list --resource-group myaksrg --
vnet-name myVnet --query "[0].id" --output tsv



# Create AKS cluster with Azure CNI:
az aks create \
  --resource-group myaksrg \
  --name azurecniCluster \
  --network-plugin azure \
  --vnet-subnet-id "/subscriptions/07aef18f-f3b6-432a-8e63-
ea7131d5f83b/resourceGroups/myaksrg/providers/
Microsoft.Network/virtualNetworks/MyVnet/subnets/default" \
  --docker-bridge-address 172.17.0.1/16 \
```

```
  --dns-service-ip 10.0.8.10 \
  --service-cidr 10.0.8.0/21 \
  --generate-ssh-keys \
  --service-principal "63e3dd7e-ff67-4bb7-
a0fb-8b077e6acbcd" \
  --client-secret "a040ae44-9036-4ef9-a898-233782ab5406"
```

Lecture: Network Policies

By default, all traffic is allowed from pod to pod. In this lesson, we'll discuss why you need to set a rule to deny all traffic by creating a network policy. Using network policies, you can allow traffic by port, label, or namespace.

Commands Used in This Lesson:

```
# Create the AKS cluster with calico network policies:
az aks create \
--resource-group myaksrg \
--name netPolicyCluster \
--node-count 3 \
--generate-ssh-keys \
--network-policy calico \
--network-plugin kubenet \
--service-principal "b692ba20-d946-4454-975e-a6699d0bf874"
\
--client-secret "cbf13526-b9bf-4785-a16e-7ebff63ee4e4"


# Add the cluster into your current context:
az aks get-credentials --resource-group myaksrg --name
netPolicyCluster
```

```
# Create YAML:
apiVersion: apps/v1
kind: Deployment
metadata:
  name: db
  labels:
    app: db
spec:
  replicas: 1
  selector:
    matchLabels:
      app: db
  template:
    metadata:
      labels:
        app: db
    spec:
      containers:
      - name: couchdb
        image: couchdb:2.3.0
        ports:
        - containerPort: 5984
---
apiVersion: v1
kind: Service
metadata:
  name: db
spec:
  selector:
    app: db
  ports:
    - name: db
      port: 15984
      targetPort: 5984
  type: ClusterIP
```

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api
  labels:
    app: api
spec:
  replicas: 1
  selector:
    matchLabels:
      app: api
  template:
    metadata:
      labels:
        app: api
    spec:
      containers:
      - name: nodebrady
        image: mabenoit/nodebrady
        ports:
        - containerPort: 3000
---
apiVersion: v1
kind: Service
metadata:
  name: api
spec:
  selector:
    app: api
  ports:
    - name: api
      port: 8080
      targetPort: 3000
  type: ClusterIP
```

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web
  labels:
    app: web
spec:
  replicas: 1
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: web
spec:
  selector:
    app: web
  ports:
    - name: web
      port: 80
      targetPort: 80
  type: LoadBalancer
```

```
# Create multi-line deployment for myapp:
kubectl apply -f deployall.yaml

# List the services and pods in the cluster:
kubectl get po,svc

# Reach out to web service:
curl [web-svc-external-ip]

# Test connection from BusyBox pod via curl.
# Open a shell in the container:
kubectl run curlpod --image=radial/busyboxplus:curl --rm
-it --generator=run-pod/v1

# Run 'curl' on the database service (once at the prompt
for container):
curl http://db:15984

# Create a network policy to deny all ingress/egress:
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  - Egress

# Apply the policy after pasting the above into a file
named 'denyall.yaml':
kubectl apply -f denyall.yaml

# Test connection again after deny all policy has been
```

```
applied:
curl --connect-timeout 2 [web-svc-external-ip]

# Create a network policy for the database pods:
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: db-netpol
spec:
  podSelector:
    matchLabels:
      app: db
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: api
    ports:
     - port: 5984
       protocol: TCP

# Apply the policy after pasting the above into a file
named 'db-netpolicy.yaml':
kubectl apply -f db-netpolicy.yaml

# Create a network policy for the API pods:
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: api-netpol
spec:
  podSelector:
    matchLabels:
```

```
      app: api
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: web
    ports:
     - port: 3000
       protocol: TCP
  egress:
  - to:
    - podSelector:
        matchLabels:
          app: db
    ports:
     - port: 5984
       protocol: TCP
  - to:
    - namespaceSelector:
        matchLabels:
          name: kube-system
      podSelector:
        matchLabels:
          k8s-app: kube-dns
    ports:
     - port: 53
       protocol: UDP

# Apply the policy after pasting the above into a file
named 'api-netpolicy.yaml':
kubectl apply -f api-netpolicy.yaml
```

```
# Label the kube-system namespace to apply network policy
to kube-dns pods:
kubectl label ns kube-system name=kube-system

# Test the network policy by applying the label 'app=api'
to the curl pod.
# Open a shell in the container:
kubectl run curlpod --image=radial/busyboxplus:curl --
labels app=api --rm -it --generator=run-pod/v1

# Run 'curl' on the database service (once at the prompt
for container):
curl http://db:15984

# Try to connect to the web service (we should not be able
to):
curl --connect-timeout 2 http://web:80

# Create a network policy for the web pods:
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: web-netpol
spec:
  podSelector:
    matchLabels:
      app: web
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from: []
    ports:
     - port: 80
       protocol: TCP
```

```
  egress:
  - to:
    - podSelector:
        matchLabels:
          app: api
    ports:
     - port: 3000
       protocol: TCP
  - to:
    - namespaceSelector:
        matchLabels:
          name: kube-system
      podSelector:
        matchLabels:
          k8s-app: kube-dns
    ports:
     - port: 53
       protocol: UDP

# Apply the policy after pasting the above into a file
named 'web-netpolicy.yaml':
kubectl apply -f web-netpolicy.yaml

# Test connection from web pods to api pods.
# Open a shell in the container:
kubectl run curlpod --image=radial/busyboxplus:curl --
labels app=web --rm -it --generator=run-pod/v1

# Run 'curl' on the API service (once at the prompt for
container):
curl http://api:8080

# Try to connect to the database service (we should not be
able to):
curl --connect-timeout 2 http://db:15984
```

Accessing AKS Externally

Lecture: Ingress Traffic

Creating load balancers in Kubernetes is great and all, but what if you want to direct incoming traffic based on URL (e.g., blog.acloud.guru vs. app.acloud.guru)? An Ingress resource in Kubernetes solves this problem. In Azure, a DNS Zone is created automatically to help direct the traffic directly to the deployment you need.

Commands Used in This Lesson

```
# Enable HTTP routing on an existing AKS cluster:
az aks enable-addons --resource-group myResourceGroup --
name myAKSCluster --addons http_application_routing

# Retrieve the DNS zone name. This name is needed to deploy
applications to the AKS cluster:
az aks show --resource-group myResourceGroup --name
myAKSCluster --query
addonProfiles.httpApplicationRouting.config.HTTPApplication
RoutingZoneName -o table

# Create a file named 'samples-http-application-
routing.yaml' and copy in the following YAML (update
<CLUSTER_SPECIFIC_DNS_ZONE>):
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: party-clippy
```

```
spec:
  template:
    metadata:
      labels:
        app: party-clippy
    spec:
      containers:
      - image: r.j3ss.co/party-clippy
        name: party-clippy
        resources:
          requests:
            cpu: 100m
            memory: 128Mi
          limits:
            cpu: 250m
            memory: 256Mi
        tty: true
        command: ["party-clippy"]
        ports:
        - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: party-clippy
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 8080
  selector:
    app: party-clippy
  type: ClusterIP
---
apiVersion: extensions/v1beta1
```

```
kind: Ingress
metadata:
  name: party-clippy
  annotations:
    kubernetes.io/ingress.class: addon-http-application-
routing
spec:
  rules:
  - host: party-
clippy.b631169463fc43d39189.westus2.aksapp.io
    http:
      paths:
      - backend:
          serviceName: party-clippy
          servicePort: 80
        path: /

# Create the resources:
kubectl apply -f samples-http-application-routing.yaml

# Use 'curl' to navigate to the hostname specified in the
host section of the 'samples-http-application-routing.yaml'
file (have to wait 3 min):
$ curl party-clippy.b631169463fc43d39189.westus2.aksapp.io

# If removing HTTP routing:
az aks disable-addons --addons http_application_routing --
name myAKSCluster --resource-group myResourceGroup --no-
wait

# Some resources may still linger. Look for 'addon-http-
application-routing' resources:
kubectl get deployments --namespace kube-system
kubectl get services --namespace kube-system
kubectl get configmaps --namespace kube-system
```

```
kubectl get secrets --namespace kube-system
```

Lecture: Web Application Firewall (WAF)

An Application Gateway in Azure is a resource that provides features and functionality of a firewall. Microsoft has created an Application Gateway called the Application Gateway Ingress Controller (AGIC) that can be deployed with an AKS cluster and still utilizes the traffic filtering, but it does not require additional ports to be open to the cluster. In this lesson we'll go through setting up an Application Gateway Ingress Controller

Commands Used in This Lesson

```
# Create service principal and store the appID and password
in a variable:
az ad sp create-for-rbac --skip-assignment -o json >
auth.json
appId=$(jq -r ".appId" auth.json)
password=$(jq -r ".password" auth.json)

# Get the objectID from the service principal and store
that in a variable:
objectId=$(az ad sp show --id $appId --query "objectId" -o
tsv)

# Insert the following in a file named 'parameters.json'
which will be used in our ARM template deployment:
cat <<EOF > parameters.json
{
  "aksServicePrincipalAppId": { "value": "$appId" },
  "aksServicePrincipalClientSecret": { "value":
"$password" },
```

```
  "aksServicePrincipalObjectId": { "value": "$objectId" },
  "aksEnableRBAC": { "value": false }
}
EOF

# Download the ARM template:
# To review the template, paste this link into your
browser: https://gist.githubusercontent.com/chadmcrowell/
a283cd7d022dc9eadc72e2f3a76d5f8a/raw/
3bf3ef9202bbfed7bc7c2caea5da7412adb9ec26/aks-app-gateway-
arm-template.json
wget https://gist.githubusercontent.com/chadmcrowell/
a283cd7d022dc9eadc72e2f3a76d5f8a/raw/
3bf3ef9202bbfed7bc7c2caea5da7412adb9ec26/aks-app-gateway-
arm-template.json -O template.json

# Deploy an AKS cluster, along with an Application Gateway,
a Virtual network, a public IP, and managed identity (to
use AAD with pods):
resourceGroupName="myaksrg"
location="westus2"
deploymentName="aks-appgateway"

az group create -n $resourceGroupName -l $location

az deployment group create \
        -g $resourceGroupName \
        -n $deploymentName \
        --template-file template.json \
        --parameters parameters.json

# Download the deployment output into a file named
'deployment-outputs.json':
az group deployment show -g $resourceGroupName -n
$deploymentName --query "properties.outputs" -o json >
```

```
deployment-outputs.json

# Use the 'deployment-outputs.json' created after
deployment to get the cluster name and resource group name:
aksClusterName=$(jq -r ".aksClusterName.value" deployment-
outputs.json)
resourceGroupName=$(jq -r ".resourceGroupName.value"
deployment-outputs.json)

az aks get-credentials --resource-group $resourceGroupName
--name $aksClusterName



# Install AAD Pod Identity
# This will add to the cluster: 3 azure identity CRDs, a
managed identity controller (watches for changes to pods,
identities, bindings), and a node managed identity (fetches
the service principal token and gives it to the pod):
kubectl create -f https://gist.githubusercontent.com/
chadmcrowell/6cea8ac1bf65d38fa1338557c59d0d43/raw/
becd8c4bdc43226e0157e7f6076327b4cb66ff8d/aad-pod-
deploy.yaml

# Install Helm (package manager for k8s):
helm init

# Add the application gateway helm repository:
helm repo add application-gateway-kubernetes-ingress
https://appgwingress.blob.core.windows.net/ingress-azure-
helm-package/
helm repo update

# Create these variables for the Helm chart:
applicationGatewayName=$(jq -r
".applicationGatewayName.value" deployment-outputs.json)
```

```
resourceGroupName=$(jq -r ".resourceGroupName.value"
deployment-outputs.json)
subscriptionId=$(jq -r ".subscriptionId.value" deployment-
outputs.json)
identityClientId=$(jq -r ".identityClientId.value"
deployment-outputs.json)
identityResourceId=$(jq -r ".identityResourceId.value"
deployment-outputs.json)

# Create the Helm chart:
# To review this chart, paste this link in your browser:
https://gist.githubusercontent.com/chadmcrowell/
e9c3945ec6e6230902a9cbff69d0d857/raw/
9f080c160f4736adbe2f1de4b6e907680231e994/appgw-helm-
chart.yam
wget https://gist.githubusercontent.com/chadmcrowell/
e9c3945ec6e6230902a9cbff69d0d857/raw/
9f080c160f4736adbe2f1de4b6e907680231e994/appgw-helm-
chart.yaml -O helm-config.yaml

# Replace the values in the Helm chart with your deployment
output:
sed -i "s|<subscriptionId>|${subscriptionId}|g" helm-
config.yaml
sed -i "s|<resourceGroupName>|${resourceGroupName}|g" helm-
config.yaml
sed -i "s|<applicationGatewayName>|$
{applicationGatewayName}|g" helm-config.yaml
sed -i "s|<identityResourceId>|${identityResourceId}|g"
helm-config.yaml
sed -i "s|<identityClientId>|${identityClientId}|g" helm-
config.yaml

# Install the app gateway via Helm chart:
helm install -f helm-config.yaml application-gateway-
```

```
kubernetes-ingress/ingress-azure

# Install sample app now that we have the Application
Gateway and AKS installed:
curl https://gist.githubusercontent.com/chadmcrowell/
9d171392eb6906133ea4342b695b0871/raw/
f46b34273b44ef8b5b89332f4adf79d7f247b0a0/app-gateway-
deploy.yaml -o aspnetapp.yaml

kubectl apply -f aspnetapp.yaml
```

Storage Types

Lecture: Persistent Volumes and Persistent Volume Claims - PART 1

In this lesson we'll talk about the benefit of persistent volumes. I will demonstrate how emptyDir volumes, although simple to create, will not provide consistent storage to pods as they move across nodes. Finally, I will demonstrate the use case for a ConfigMap and Secret, and we'll create them together.

Commands Used in This Lesson

```
# Create the YAML for our emptyDir volume pod:
kubectl run localvolpod --image=busybox --restart=Never -o
yaml --dry-run -- /bin/sh -c 'echo hello;sleep 3600' >
localvolpod.yaml

apiVersion: v1
kind: Pod
metadata:
```

```
  labels:
    run: busybox
  name: busybox
spec:
  containers:
  - args:
    - /bin/sh
    - -c
    - echo hello;sleep 3600
    image: busybox
    name: busybox
    volumeMounts:
    - mountPath: /tmp/data
      name: local-vol
  volumes:
  - name: local-vol
    emptyDir: {}

# Create the local volume pod:
kubectl apply -f localvolpod.yaml

# List pods:
kubectl get po

# Describe the pod:
kubectl describe po

# Connect to the pod:
kubectl exec -it localvolpod sh

# Create a file in the emptyDir from the container:
echo "sometext" > file.txt

# Find which node the pod is on:
kubectl get po -o wide
```

```
# List the nodes:
kubectl get no –o wide

# Scale the AKS cluster:
az aks scale ––name clus1 ––node–count 2 ––resource–group
aksrg

# Delete the pod:
kubectl delete –f localvolpod.yaml

# Cordon the node:
kubectl cordon aks–node1

# Verify that the pod has been scheduled to a different
node:
kubectl get po –o wide

# Verify that the file no longer exists in the emptyDir
volume:
kubectl exec –it localvolpod sh
ls /tmp/data



# Create a ConfigMap with the value 'mykey' and 'myvalue'
example:
kubectl create configmap myconfigmap ––from–
literal=mykey=myValue

# Create a file named 'configmappod.yaml':
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: configmappod
```

```
  name: configmappod
spec:
  containers:
  - args:
    - /bin/sh
    - -c
    - echo $(cat /etc/config/mykey);sleep 3600
    image: busybox
    name: configmappod
    volumeMounts:
    - mountPath: /etc/config
      name: configmap-vol
  volumes:
  - name: configmap-vol
    configMap:
      name: myconfigmap

# Create the ConfigMap pod:
kubectl apply -f configmappod.yaml

# Create a secret that will pass an encrypted password to a
pod:
kubectl create secret generic user-pass --from-
literal=username=devuser --from-literal=password='S!
B\*d$zDsb'

# Create a file named 'secretpod.yaml':
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: secretpod
  name: secretpod
spec:
  containers:
```

```
  - image: redis
    name: secretpod
    volumeMounts:
    - mountPath: /etc/secret
      name: secret-vol
      readOnly: true
  volumes:
  - name: secret-vol
    secret:
      secretName: user-pass

# Create the pod that mounts a secret to a pod using a
volume:
kubectl apply -f secretpod.yaml
```

Lecture: Persistent Volumes and Persistent Volume Claims - PART 2

```
# NOTE: The disk must reside in the node group
# get the nodegroup rg name
NODE_GROUP=$(az aks show --resource-group myResourceGroup
--name myAKSCluster --query nodeResourceGroup -o tsv)
AKS_CLUSTER=clus1

# create an azure disk
az disk create --resource-group $NODE_GROUP --name
myAKSDisk --size-gb 10 --query id --output tsv

# copy URI and paste in this YAML
kubectl run secretpod --image=nginx:1.15.5 --restart=Never
-o yaml --dry-run > azurediskpod.yaml

# create a file named azure-disk-pod.yaml and insert the
```

```
following:
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - image: nginx:1.15.5
    name: mypod
    resources:
      requests:
        cpu: 100m
        memory: 128Mi
      limits:
        cpu: 250m
        memory: 256Mi
    volumeMounts:
      - name: azure
        mountPath: /mnt/azure
  volumes:
      - name: azure
        azureDisk:
          kind: Managed
          diskName: myAKSDisk
          diskURI: /subscriptions/<subscriptionID>/
resourceGroups/MC_myAKSCluster_myAKSCluster_eastus/
providers/Microsoft.Compute/disks/myAKSDisk

# create the azure disk pod
kubectl apply -f azure-disk-pod.yaml

# check to see the pod running
kubectl get po -w

# connect directly to the pod and write some data
```

```
kubectl exec -it mypod sh
> cd /mnt/azure
> pwd
> echo "sometext" > file1.txt
> exit

# see which node the pod is on
kubectl get po -o wide

# uncordon the node
kubectl uncordon aks-nodepool1-24331151-vmss00000

# delete the pod
kubectl delete po mypod

# create a new pod with the same yaml
kubectl apply -f azure-disk-pod.yaml

# cordon the node to make sure the pod lands on a different
node
kubectl cordon aks-nodepool1-24331151-vmss000001

# connect to the pod again
kubectl exec -it mypod sh
> cd /mnt/azure
> ls

# create storage account
az storage account create -n aksdeepdive20834 -g aksrg -l
eastus --sku Standard_LRS

# Export the connection string as an environment variable,
this is used when creating the Azure file share
CONNECTION_STRING=$(az storage account show-connection-
string -n aksdeepdive20834 -g aksrg -o tsv)
```

```
# Create the file share
az storage share create -n fileshare1 --connection-string
$AZURE_STORAGE_CONNECTION_STRING

# Get storage account key
STORAGE_KEY=$(az storage account keys list --resource-group
aksrg --account-name aksdeepdive20834 --query "[0].value"
-o tsv)

# create a secret to store the storage account key
kubectl create secret generic azure-secret --from-
literal=azurestorageaccountname=aksdeepdive20834 --from-
literal=azurestorageaccountkey=$STORAGE_KEY

# create a file named pv.yaml with the following contents:
apiVersion: v1
kind: PersistentVolume
metadata:
  name: fileshare-pv
  labels:
    usage: fileshare-pv
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Retain
  azureFile:
    secretName: azure-secret
    shareName: fileshare1
    readOnly: false

# create the persistent volume (pv)
kubectl apply -f pv.yaml
```

```
# create a file named pvc.yaml with the following contents:
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: fileshare-pvc
  annotations:
    volume.beta.kubernetes.io/storage-class: ""
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 10Gi
  selector:
    matchLabels:
      usage: fileshare-pv

# create the persistent volume claim (PVC)
kubectl apply -f pvc.yaml

# view the persistent volumes in order to see the bound
status
kubectl get pv

# view the persistent volume claims in order to see the
bound status
kubectl get pvc

# create a file named pvcpod.yaml with the following
contents:
apiVersion: v1
kind: Pod
metadata:
 name: pvcpod
```

```
spec:
 containers:
  - image: nginx
    name: pvcpod
    volumeMounts:
      - name: azure
        mountPath: /usr/share/nginx/html
 volumes:
  - name: azure
    persistentVolumeClaim:
      claimName: fileshare-pvc

# create the pod with attached pvc
kubectl apply -f pvcpod.yaml

# view the storage classes available in your cluster
kubectl get sc

# create a file named sc.yaml with the following contents
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: azurefile
provisioner: kubernetes.io/azure-file
mountOptions:
  - dir_mode=0777
  - file_mode=0777
  - uid=0
  - gid=0
  - mfsymlinks
  - cache=strict
parameters:
  skuName: Premium_LRS

# create the storage class
```

```
kubectl apply -f sc.yaml

# create a file named pvc1.yaml with the following
contents:
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: azurefile
spec:
  accessModes:
    - ReadWriteMany
  storageClassName: azurefile
  resources:
    requests:
      storage: 100Gi

# view the pvc creation status
kubectl get pvc

# view the pvc bound to the pv
kubectl get pv

# create a file named pvcpod1.yaml with the following
contents:
apiVersion: v1
kind: Pod
metadata:
 name: mypod1
spec:
 containers:
  - image: nginx:1.15.5
    name: mypod1
    resources:
      requests:
        cpu: 100m
```

```
        memory: 128Mi
      limits:
        cpu: 250m
        memory: 256Mi
    volumeMounts:
      - name: volume
        mountPath: "/mnt/azure"
 volumes:
  - name: volume
    persistentVolumeClaim:
      claimName: azurefile

# create the pod with pvc attached
kubectl apply -f pvcpod1.yaml
```

Lecture: Static and Dynamic Volumes

Static volumes are created by the cluster administrator and cannot be changed after creation time. Dynamic volumes are more flexible, because you can request any size volume upon PVC creation. Remember, with Azure Disks, you can only mount a volume to one pod at a time. With Azure Files, you can mount the file share to as many pods as you want.

AKS Workloads

Lecture: Node Sizing

The performance of your cluster comes down to VM size in Azure. In this lesson, we'll go through the process of creating an additional node pool with a high-speed disk, which will improve the performance of the applications running inside of our AKS

cluster

Commands Used in This Lesson

```
# List the available VM SKUs:
az aks create -l eastus -s  # hit [TAB] twice here

# Create a new node pool:
#   - For Linux node pools, the length must be between 1
and 12 characters. For Windows node pools,
#     the length must be between 1 and 6 characters.
#   - If no size is specified, Standard_DS2_v3 is applied
for Windows node pools and Standard_DS2_v2 for Linux node
pools.
#   - If no kubernetes-version is specified, it defaults to
the same version as the control plane.
az aks nodepool add \
    --resource-group aksrg \
    --cluster-name clus1 \
    --name fseriespool \
    --node-count 3 \
    --node-vm-size Standard_F8s_v2 \
    --kubernetes-version 1.15.7

# List node pools:
az aks nodepool list --resource-group aksrg --cluster-name
clus1

# Upgrade the node pool:
#   - The version must have the same major version as the
control plane.
#   - You can change the version of the control plan in the
portal k8s > version.
#   - The node pool minor version must be within two minor
```

```
versions of the control plane version.
#   - You cannot scale the cluster and upgrade the nodes at
the same time.
az aks nodepool upgrade \
    --resource-group aksrg \
    --cluster-name clus1 \
    --name fseriespool \
    --kubernetes-version 1.15.7 \
    --no-wait
```

Lecture: Azure Site Recovery

Being able to restore you cluster in place or to another cluster is crucial to disaster recovery. In this lesson we'll go through best practices for apps running in Kubernetes and how you can create full and incremental snapshots in Azure.

Commands used in this lesson:

```
# get volume name
kubectl get pvc

# get the disk id
az disk list --query '[].id | [?
contains(@,`pvc-8226a8f3-2696-45e5-93aa-6ab34993434d`)]' -o
tsv

# copy the disk id and paste into snapshot create command
az snapshot create \
  --resource-group MC_aksrg_clus1_eastus \
  --name pvcSnapshot \
  --source /subscriptions/07aef18f-f3b6-432a-8e63-
ea7131d5f83b/resourceGroups/MC_AKSRG_CLUS1_EASTUS/
```

```
providers/Microsoft.Compute/disks/kubernetes-dynamic-
pvc-8226a8f3-2696-45e5-93aa-6ab34993434d

# create disk from snapshot
az disk create --resource-group MC_aksrg_clus1_eastus --
name pvcRestored --source pvcSnapshot

# get the id of the disk to use in a pod
az disk show --resource-group MC_aksrg_clus1_eastus --name
pvcRestored --query id -o tsv

# pod yaml
kind: Pod
apiVersion: v1
metadata:
  name: mypodrestored
spec:
  containers:
  - name: mypodrestored
    image: nginx:1.15.5
    resources:
      requests:
        cpu: 100m
        memory: 128Mi
      limits:
        cpu: 250m
        memory: 256Mi
    volumeMounts:
    - mountPath: "/mnt/azure"
      name: volume
  volumes:
    - name: volume
      azureDisk:
        kind: Managed
        diskName: pvcRestored
```

```
        diskURI:

# create pod
kubectl apply -f mypodrestored.yaml

# get the disk name
az disk list

# create incremental snapshots by setting the incremental
property to true
# consist only of all the changes since the last snapshot,
so cost savings
# Incremental snapshots will always use standard HDDs
storage, irrespective of the storage type of the disk,
whereas regular snapshots can use premium SSDs
# only these regions are supported: West Central US, East
US, East US 2, Central US, Canada East, Canada Central,
North Europe, South East Asia
az snapshot create \
-g MC_aksrg_clus1_eastus \
-n aksIncSnapshot \
-l eastus \
--source /subscriptions/07aef18f-f3b6-432a-8e63-
ea7131d5f83b/resourceGroups/MC_AKSRG_CLUS1_EASTUS/
providers/Microsoft.Compute/disks/myAKSDisk \
--incremental
```

Resource Management

Lecture: Pod Resource Requests and Limits

In order for the Kubernetes scheduler to schedule a pod to a node, it has to ensure that the node has enough resources

available. You can reserve CPU and RAM by specifying requests and limits in your pod or deployment YAML.

Commands Used in This Lesson

```
# Describe node to get allocatable and allocated:
kubectl describe no <node-name>

# It's recommended to set requests and limits for all pods
in your cluster.
# Requests are how much the pod needs.
# Limits are the greatest amount of resources the pod can
consume.

# Pod with two containers:
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: db
    image: mysql:8
    env:
    - name: MYSQL_ROOT_PASSWORD
      value: "password"
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
  - name: wp
```

```
    image: wordpress:php7.4
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"

# Get the container logs (f=logs should be streamed):
kubectl logs -f -c wp frontend
kubectl logs -f -c db frontend

# Get the CPU and memory usage:
kubectl top po frontend

# Set a Pod Disruption Budget

# A pod disruption budget sets an absolute minimum number
of pods that need to run for your application to still be
available.
# A PDB selects the pods by specifying the label of the
pod:
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: nginx-pdb
spec:
  minAvailable: 2
  selector:
    matchLabels:
      app: nginx
```

Lecture: Scaling Options

Azure has a built-in pod and cluster autoscaler. This means that when your application demands more resources, those resources will be automatically provisioned. In this lesson we'll go through manually and automatically scaling your cluster.

Commands Used in This Lesson

```
# Manually scale pods:
kubectl get po,deploy

kubectl scale --replicas=5 deployment/nginx

kubectl scale --replicas=3 deployment/nginx

# Verify that version is above 1.10:
az aks show --resource-group aksrg --name clus1 --query
kubernetesVersion --output table

# Adjust the number of pods in a deployment depending on
CPU utilization from the metrics server.
# To use the autoscaler, all containers in your pods, and
your pods, must have CPU requests and limits defined.
# Increase number of pods if pod CPU exceeds 50% of their
requested usage, up to 10:
kubectl autoscale deployment nginx --cpu-percent=50 --min=3
--max=10

# Get horizontal pod autoscaler status:
kubectl get hpa

# Manually scale nodes:
az aks scale --resource-group aksrg --name clus1 --node-
```

```
count 3

# Cluster autoscaler

# This watches for pods in your cluster that can't be
scheduled and scales up.
# It also watches for underutilized nodes to scale down.
# Limitation: The HTTP application routing add-on can't be
used:
az aks nodepool update \
  --cluster-name clus1 \
  --name nodepool1 \
  --resource-group aksrg \
  --enable-cluster-autoscaler \
  --min-count 1 \
  --max-count 5

# Update the cluster autoscaler:
az aks update \
  --resource-group aksrg \
  --name clus1 \
  --update-cluster-autoscaler \
  --min-count 2 \
  --max-count 5

# The cluster is scanned every 10 min to scale up or down.
# If you had workloads that ran every 15 minutes, you might
want to change this.
# Install aks-preview:
az extension add --name aks-preview

# Optionally, update the extension:
az extension update --name aks-preview

# Update the scan interval:
```

```
az aks update \
  --resource-group aksrg \
  --name clus1 \
  --cluster-autoscaler-profile scan-interval=15s

# To reset back to defaults:
az aks update \
  --resource-group aksrg \
  --name clus1 \
  --cluster-autoscaler-profile ""

# Disable cluster autoscaler:
az aks update \
  --resource-group aksrg \
  --name clus1 \
  --disable-cluster-autoscaler
```

Lecture: Kube-Advisor Tool

Since the metrics server requires that all pods have requests and limits, you can use the `kube-advisor` tool to detect if any pods in your entire cluster are missing those requests and limits.
The `kube-advisor` tool can be used for pods running Windows containers as well as Linux.

Commands Used in This Lesson

```
# Create the YAML for a service account and cluster role
binding:
apiVersion: v1
kind: ServiceAccount
metadata:
  name: kube-advisor
  namespace: default
```

```
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: kube-advisor
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: kube-advisor
  namespace: default

# Apply the YAML to create the service account and
ClusterRoleBinding:
kubectl run --rm -i -t kubeadvisor --
image=mcr.microsoft.com/aks/kubeadvisor --restart=Never --
overrides="{ \"apiVersion\": \"v1\", \"spec\":
{ \"serviceAccountName\": \"kube-advisor\" } }"

# Optionally, run this if RBAC is disabled:
kubectl run --rm -i -t kubeadvisor --
image=mcr.microsoft.com/aks/kubeadvisor --restart=Never
```

Advanced Scheduling Features

Lecture: Azure Dev Spaces

If you're working in teams on applications in AKS, Azure Dev Spaces is perfect for you. You can create custom endpoints for each team member, where they can work on their code, independent of the production cluster. When they've perfected the service in AKS, they can check their code back into source

control, and continue to be a dev rockstar! No additional clusters or components are needed, just the Azure Dev Spaces extension in VS Code.

```
# Create username and password variables:
adminUsername="azureuser"
adminPassword=$(openssl rand -base64 16)


# Insert the following in a file named 'parameters.json'
which will be used in our ARM template deployment:
cat <<EOF > parameters.json
{
  "adminUsername": { "value": "$adminUsername" },
  "adminPassword": { "value": "$adminPassword" }
}
EOF

# Download the ARM template:
wget https://gist.githubusercontent.com/chadmcrowell/
375e9e3c20cfe048dc4e83250b833684/raw/
d5a1e832a12a4ee7798c956a55533b12a5a24580/win-vm-deploy.json
-O template.json

# Create the resource group and deploy the ARM template:
resourceGroupName="vmdeploy-rg"
location="eastus"
deploymentName="win-vm-deploy"

az group create -n $resourceGroupName -l $location

az deployment group create \
        -g $resourceGroupName \
        -n $deploymentName \
        --template-file template.json \
```

```
        --parameters parameters.json

# RDP into VM and install .net core sdk.
# Create a new project folder.
# Open VSCode, log into Azure:
az login
# Install Azure Dev Spaces VSCode extension.
# Install C# VSCode extension.
# Restart VSCode.

# Create new webapp:
dotnet new webapp

# Enable Dev Spaces:
az aks use-dev-spaces -g aksrg3 -n clus3

# With the project open in VSCode, click View > Command
Palette and select "Azure Dev Spaces: Prepare configuration
files for Azure Dev Spaces"
# When prompted, choose Yes to enable a public endpoint.
# This prepares your project to run in Azure Dev Spaces by
generating a Dockerfile and Helm chart. It also generates a
'.vscode' directory with debugging configuration at the
root of your project.

# Click on the debugger button and click RUN.
# This command builds and runs your service in Azure Dev
Spaces in debugging mode.
# The Terminal window at the bottom shows the build output
and URLs for your service running in Azure Dev Spaces. The
Debug Console shows the log output.
# Change the heading in '\Pages\index.cshtml'.
# Restart the debugger.
```

Lecture: Virtual Kubelet

Scale more rapidly with Virtual Nodes in Kubernetes. The difference between virtual nodes and node pools is the dynamic nature of pods, and not having to build the static infrastructure in advance (the nodes). In this lesson, I will show you how to enable the virtual node addon in AKS and create a deployment that uses a node selector to always choose that virtual node for scheduling.

Commands Used in This Lesson

```
# Enable virtual node:
az aks enable-addons --addons virtual-node --name
virtualKubeCluster --resource-group myaksrg5 --subnet-name
virtualnodesubnet

# Create a deployment with a node selector for the virtual
kubelet:
apiVersion: apps/v1
kind: Deployment
metadata:
  name: aci-helloworld
spec:
  replicas: 1
  selector:
    matchLabels:
      app: aci-helloworld
  template:
    metadata:
      labels:
        app: aci-helloworld
    spec:
      containers:
      - name: aci-helloworld
        image: microsoft/aci-helloworld
```

```
        ports:
        - containerPort: 80
      nodeSelector:
        kubernetes.io/role: agent
        beta.kubernetes.io/os: linux
        type: virtual-kubelet
      tolerations:
      - key: virtual-kubelet.io/provider
        operator: Exists
      - key: azure.com/aci
        effect: NoSchedule

kubectl apply aci-helloworld.yaml


# Scale the deployment:
kubectl scale deploy aci-helloworld --replicas 10
```

Key Metrics

Lecture: Azure Monitor for Containers

Collecting metrics data about our cluster is important for many reasons, the first of which is forecasting. Using metric data, you'll know how to change your cluster and prepare for increases in capacity or performance. In this lesson, we'll talk about Azure Monitor for Containers, which is a component of the Log Analytics workspace, allowing you to capture detailed information about your control plane, nodes, pods and containers in your AKS cluster.

Commands Used in This Lesson

```
# Enable container monitoring on a new cluster:
az aks create \
  --resource-group aksrg10 \
  --name clus10 \
  --node-count 3 \
  --enable-addons monitoring \
  --generate-ssh-keys \
  --service-principal "bb01b96f-eb15-404d-b9b7-
d469a09063d2" \
  --client-secret "9fd44037-e555-4e99-80ed-e739924a5b37"

# Enable monitoring on an existing cluster:
az aks enable-addons -a monitoring -n clus9 -g aksrg9

# List the log analytics workspaces in my subscription:
az resource list --resource-type
Microsoft.OperationalInsights/workspaces -o json

# Choose an existing workspace for an existing cluster:
az aks enable-addons -a monitoring -n clus9 -g aksrg9 --
workspace-resource-id "<workspace-id>"
```

Lecture: Cluster Metrics

There are many ways of collecting metric data about your cluster. The container monitoring features in AKS allow you to start at the controller level, and drill down to a conatiner level, to get as much data as possible. In this lesson, we'll take a look at each of the categories of metrics and explore how to detect errors in your cluster.

Collecting Container Logs

Lecture: Application Insights

In this lesson, to gain a better forecast of our logs over time, we will go over some popular queries that return cluster and container logs.

```
# List all container lifecycle information:
ContainerInventory
| project Computer, Name, Image, ImageTag, ContainerState,
CreatedTime, StartedTime, FinishedTime
| render table

# Kubernetes events:
KubeEvents | where not(isempty(Namespace))
| sort by TimeGenerated desc
| render table

# Container CPU:
Perf
| where ObjectName == "K8SContainer" and CounterName ==
"cpuUsageNanoCores"
| summarize AvgCPUUsageNanoCores = avg(CounterValue) by
bin(TimeGenerated, 30m), InstanceName

# Container memory:
Perf
| where ObjectName == "K8SContainer" and CounterName ==
"memoryRssBytes"
| summarize AvgUsedRssMemoryBytes = avg(CounterValue) by
bin(TimeGenerated, 30m), InstanceName
```

Lecture: View Live Logs

Live logs allow you to see a real-time event stream of container logs. It is similar to `kubectl logs` or `kubectl get events`, but allows you to alert or report on data from within the Azure portal. The data is never stored, so this tool is more for a current stream view versus a log collection mechanism.