Contents

Azure Kubernetes Service (AKS)

9/26/2018 • 5 minutes to read • Edit Online

Azure Kubernetes Service (AKS) makes it simple to deploy a managed Kubernetes cluster in Azure. AKS reduces the complexity and operational overhead of managing Kubernetes by offloading much of that responsibility to Azure. As a hosted Kubernetes service, Azure handles critical tasks like health monitoring and maintenance for you. The Kubernetes masters are managed by Azure. You only manage and maintain the agent nodes. As a managed Kubernetes service, AKS is free - you only pay for the agent nodes within your clusters, not for the masters.

You can create an AKS cluster in the Azure portal, with the Azure CLI, or template driven deployment options such as Resource Manager templates and Terraform. When you deploy an AKS cluster, the Kubernetes master and all nodes are deployed and configured for you. Additional features such as advanced networking, Azure Active Directory integration, and monitoring can also be configured during the deployment process.

To get started, complete the AKS quickstart in the Azure portal or with the Azure CLI.

Access, security, and monitoring

For improved security and management, AKS lets you integrate with Azure Active Directory and use Kubernetes role-based access controls. You can also the monitor the health of your cluster and resources.

Identity and security management

To limit access to cluster resources, AKS supports Kubernetes role-based access control (RBAC). RBAC lets you control how can access Kubernetes resources and namespaces, and what permissions that have on those resources. You can also configure an AKS cluster to integrate with Azure Active Directory (AD). With Azure AD integration, Kubernetes access can be configured based on existing identity and group membership. Your existing Azure AD users and groups can be provided access to AKS resources and with an integrated sign-on experience.

To secure your AKS clusters, see Integrate Azure Active Directory with AKS.

Integrated logging and monitoring

To understand how your AKS cluster and deployed applications are performing, Azure Monitor for container health collects memory and processor metrics from containers, nodes, and controllers. Container logs are available, and you can also review the Kubernetes master logs. This monitoring data is stored in an Azure Log Analytics workspace, and is available through the Azure portal, Azure CLI, or a REST endpoint.

For more information, see Monitor Azure Kubernetes Service container health.

Cluster and node

AKS nodes run on Azure virtual machines. You can connect storage to nodes and pods, upgrade cluster components, and use GPUs.

Cluster node and pod scaling

As demand for resources change, the number of cluster nodes or pods that run your services can automatically scale up or down. You can use both the horizontal pod autoscaler or the cluster autoscaler. This approach to scaling lets the AKS cluster automatically adjust to demands and only run the resources needed.

For more information, see Scale an Azure Kubernetes Service (AKS) cluster.

Cluster node upgrades

Azure Kubernetes Service offers multiple Kubernetes versions. As new versions become available in AKS, your

cluster can be upgraded using the Azure portal or Azure CLI. During the upgrade process, nodes are carefully cordoned and drained to minimize disruption to running applications.

To learn more about lifecycle versions, see Supported Kubernetes versions in AKS. For steps on how to upgrade, see Upgrade an Azure Kubernetes Service (AKS) cluster.

GPU enabled nodes

AKS supports the creation of GPU enabled node pools. Azure currently provides single or multiple GPU enabled VMs. GPU enabled VMs are designed for compute-intensive, graphics-intensive, and visualization workloads.

For more information, see Using GPUs on AKS.

Storage volume support

To support application workloads, you can mount storage volumes for persistent data. Both static and dynamic volumes can be used. Depending on how many connected pods are to share the storage, you can use storage backed by either Azure Disks for single pod access, or Azure Files for multiple concurrent pod access.

Get started with dynamic persistent volumes with Azure Disks or Azure Files.

Virtual networks and ingress

An AKS cluster can be deployed into an existing virtual network. In this configuration, every pod in the cluster is assigned an IP address in the virtual network, and can directly communicate with other pods in the cluster, and other nodes in the virtual network. Pods can connect also to other services in a peered virtual network, and to on-premises networks over ExpressRoute or site-to-site (S2S) VPN connections.

For more information, see the AKS networking overview.

Ingress with HTTP application routing

The HTTP application routing add-on makes it easy to access applications deployed to your AKS cluster. When enabled, the HTTP application routing solution configures an ingress controller in your AKS cluster. As applications are deployed, publicly accessible DNS names are auto configured. The HTTP application routing configures a DNS zone and integrates it with the AKS cluster. You can then deploy Kubernetes ingress resources as normal.

To get started with ingress traffic, see HTTP application routing.

Development tooling integration

Kubernetes has a rich ecosystem of development and management tools such as Helm, Draft, and the Kubernetes extension for Visual Studio Code. These tools work seamlessly with AKS.

Additionally, Azure Dev Spaces provides a rapid, iterative Kubernetes development experience for teams. With minimal configuration, you can run and debug containers directly in AKS. To get started, see Azure Dev Spaces.

The Azure DevOps project provides a simple solution for bringing existing code and Git repository into Azure. The DevOps project automatically creates Azure resources such as AKS, a release pipeline in Azure DevOps Services that includes a build pipeline for CI, sets up a release pipeline for CD, and then creates an Azure Application Insights resource for monitoring.

For more information, see Azure DevOps project.

Docker image support and private container registry

AKS supports the Docker image format. For private storage of your Docker images, you can integrate AKS with Azure Container Registry (ACR).

To create private image store, see Azure Container Registry.

Kubernetes certification

Azure Kubernetes Service (AKS) has been CNCF certified as Kubernetes conformant.

Regulatory compliance

Azure Kubernetes Service (AKS) is compliant with SOC, ISO, and PCI DSS.

Next steps

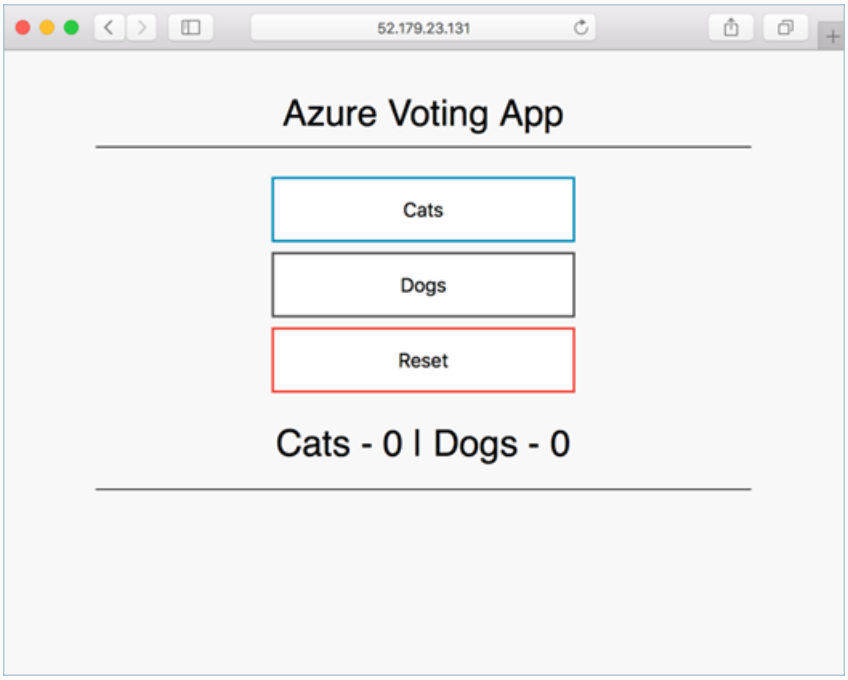
Learn more about deploying and managing AKS with the Azure CLI quickstart.

AKS quickstart

Quickstart: Deploy an Azure Kubernetes Service (AKS) cluster

9/24/2018 • 5 minutes to read • Edit Online

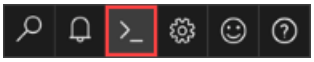
In this quickstart, an AKS cluster is deployed using the Azure CLI. A multi-container application consisting of web front end and a Redis instance is then run on the cluster. Once completed, the application is accessible over the internet.



This quickstart assumes a basic understanding of Kubernetes concepts, for detailed information on Kubernetes see the Kubernetes documentation.

Open Azure Cloud Shell

Azure Cloud Shell is a free, interactive shell that you can use to run the steps in this article. Common Azure tools are preinstalled and configured in Cloud Shell for you to use with your account. Just select the **Copy** button to copy the code, paste it in Cloud Shell, and then press Enter to run it. There are a few ways to open Cloud Shell, you can choose any one of them to open Cloud Shell:

Select **Try It** in the upper-right corner of a code block.	
Open Cloud Shell in your browser.	
Select the **Cloud Shell** button on the menu in the upper-right corner of the Azure portal.	

If you choose to install and use the CLI locally, this quickstart requires that you are running the Azure CLI version 2.0.46 or later. Run `az --version` to find the version. If you need to install or upgrade, see Install Azure

CLI.

Create a resource group

Create a resource group with the az group create command. An Azure resource group is a logical group in which Azure resources are deployed and managed. When you create a resource group, you are asked to specify a location. This location is where your resources run in Azure.

The following example creates a resource group named *myAKSCluster* in the *eastus* location.

```
az group create --name myAKSCluster --location eastus
```

Output:

```
{
  "id": "/subscriptions/00000000-0000-0000-0000-000000000000/resourceGroups/myAKSCluster",
  "location": "eastus",
  "managedBy": null,
  "name": "myAKSCluster",
  "properties": {
    "provisioningState": "Succeeded"
  },
  "tags": null
}
```

Create AKS cluster

Use the az aks create command to create an AKS cluster. The following example creates a cluster named *myAKSCluster* with one node. Azure Monitor for containers is also enabled using the *--enable-addons monitoring* parameter. For more information on enabling the container health monitoring solution, see Monitor Azure Kubernetes Service health.

```
az aks create --resource-group myAKSCluster --name myAKSCluster --node-count 1 --enable-addons monitoring --
generate-ssh-keys
```

After several minutes, the command completes and returns JSON-formatted information about the cluster.

Connect to the cluster

To manage a Kubernetes cluster, use kubectl, the Kubernetes command-line client.

If you're using Azure Cloud Shell, `kubectl` is already installed. If you want to install it locally, use the az aks install-cli command.

```
az aks install-cli
```

To configure `kubectl` to connect to your Kubernetes cluster, use the az aks get-credentials command. This step downloads credentials and configures the Kubernetes CLI to use them.

```
az aks get-credentials --resource-group myAKSCluster --name myAKSCluster
```

To verify the connection to your cluster, use the kubectl get command to return a list of the cluster nodes. It can take a few minutes for the nodes to appear.

```
kubectl get nodes
```

Output:

```
NAME                          STATUS    ROLES     AGE       VERSION
k8s-myAKSCluster-36346190-0   Ready     agent     2m        v1.7.7
```

Run the application

A Kubernetes manifest file defines a desired state for the cluster, including what container images should be running. For this example, a manifest is used to create all objects needed to run the Azure Vote application. This manifest includes two Kubernetes deployments, one for the Azure Vote Python applications, and the other for a Redis instance. Also, two Kubernetes Services are created, an internal service for the Redis instance, and an external service for accessing the Azure Vote application from the internet.

Create a file named `azure-vote.yaml` and copy into it the following YAML code. If you are working in Azure Cloud Shell, this file can be created using vi or Nano as if working on a virtual or physical system.

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: azure-vote-back
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: azure-vote-back
    spec:
      containers:
      - name: azure-vote-back
        image: redis
        ports:
        - containerPort: 6379
          name: redis
---
apiVersion: v1
kind: Service
metadata:
  name: azure-vote-back
spec:
  ports:
  - port: 6379
  selector:
    app: azure-vote-back
---
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: azure-vote-front
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: azure-vote-front
    spec:
      containers:
      - name: azure-vote-front
        image: microsoft/azure-vote-front:v1
        ports:
        - containerPort: 80
        env:
        - name: REDIS
          value: "azure-vote-back"
---
apiVersion: v1
kind: Service
metadata:
  name: azure-vote-front
spec:
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    app: azure-vote-front
```

Use the kubectl apply command to run the application.

```
kubectl apply -f azure-vote.yaml
```

Output:

```
deployment "azure-vote-back" created
service "azure-vote-back" created
deployment "azure-vote-front" created
service "azure-vote-front" created
```

Test the application

As the application is run, a Kubernetes service is created that exposes the application front end to the internet. This process can take a few minutes to complete.

To monitor progress, use the kubectl get service command with the `--watch` argument.

```
kubectl get service azure-vote-front --watch
```

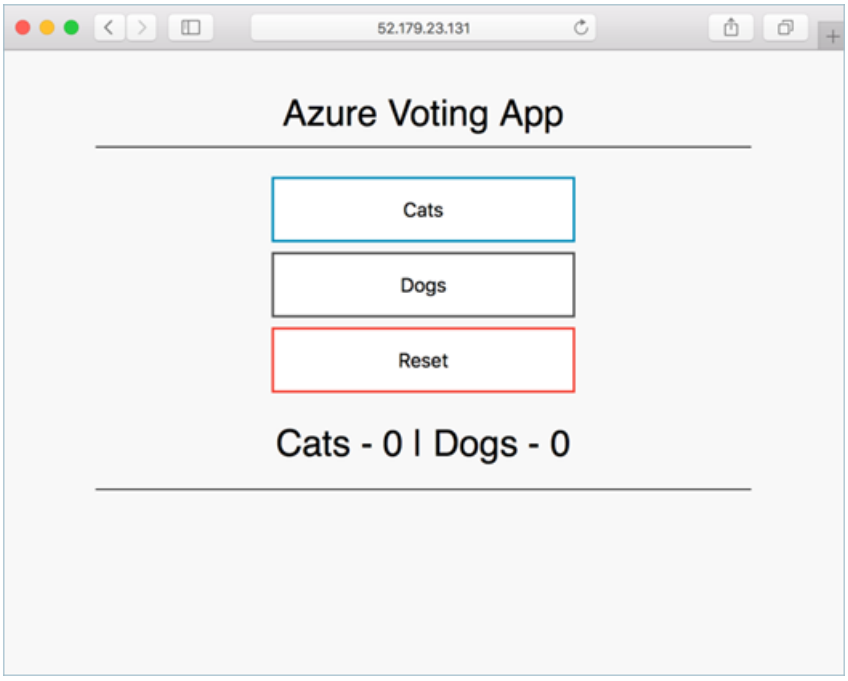
Initially the *EXTERNAL-IP* for the *azure-vote-front* service appears as *pending*.

```
NAME               TYPE           CLUSTER-IP   EXTERNAL-IP   PORT(S)        AGE
azure-vote-front   LoadBalancer   10.0.37.27   <pending>     80:30572/TCP   6s
```

Once the *EXTERNAL-IP* address has changed from *pending* to an *IP address*, use `CTRL-C` to stop the kubectl watch process.

```
azure-vote-front   LoadBalancer   10.0.37.27   52.179.23.131   80:30572/TCP   2m
```

Now browse to the external IP address to see the Azure Vote App.



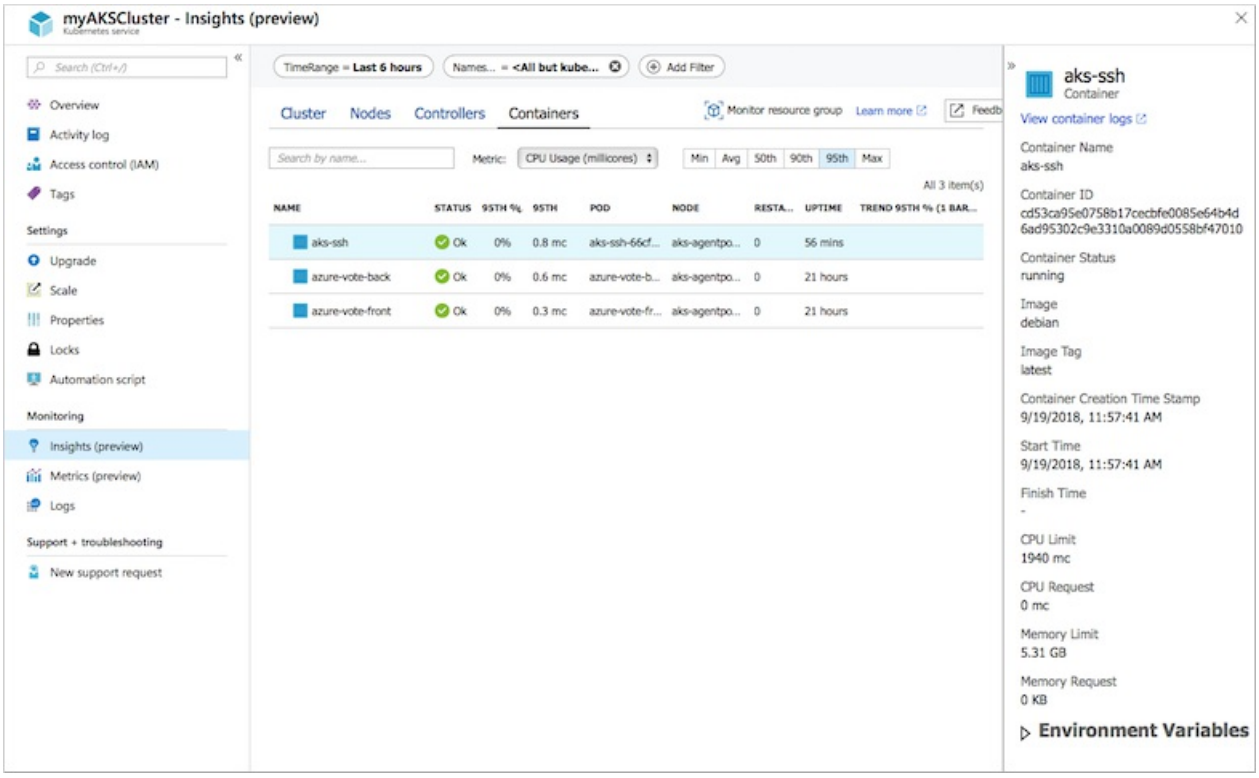
Monitor health and logs

When the AKS cluster was created, monitoring was enabled to capture health metrics for both the cluster nodes and pods. These health metrics are available in the Azure portal. For more information on container health monitoring, see Monitor Azure Kubernetes Service health.

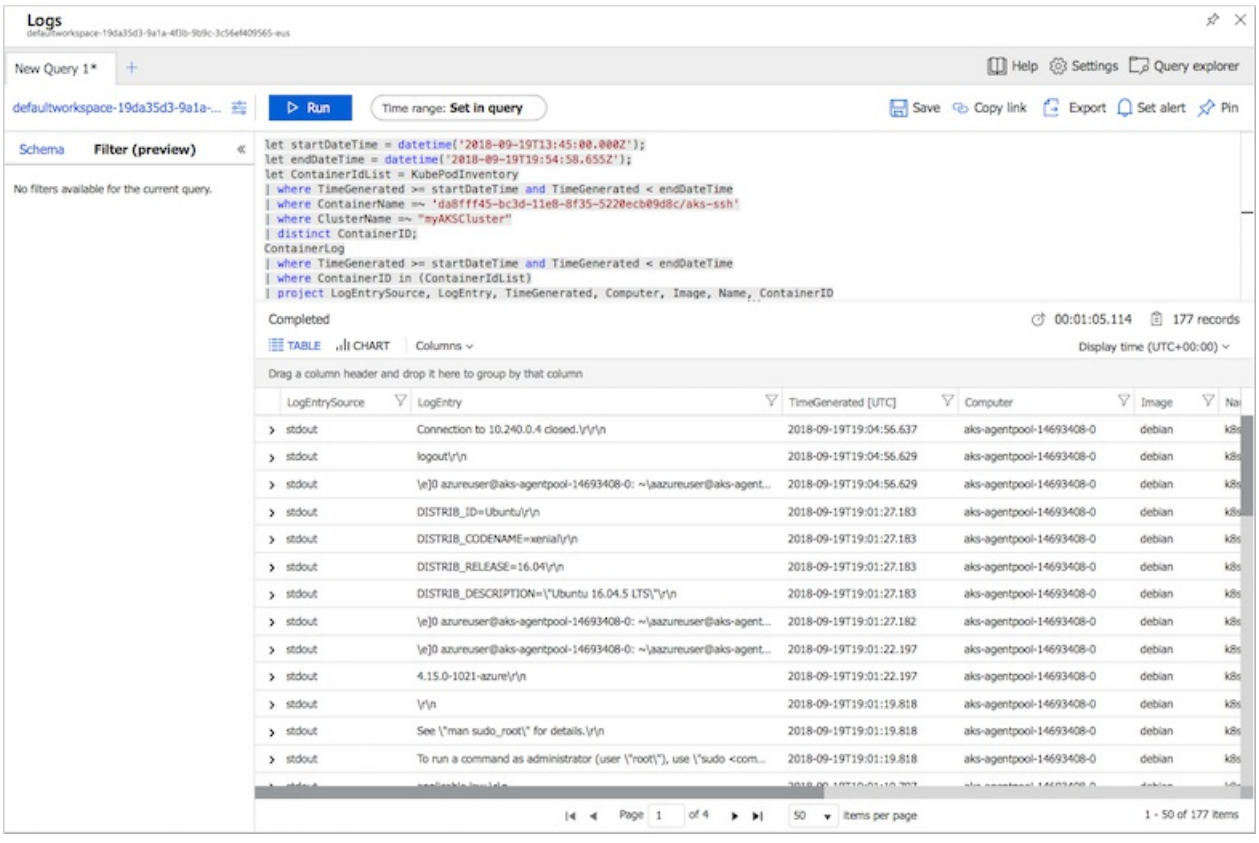
To see current status, uptime, and resource usage for the Azure Vote pods, complete the following steps:

1. Open a web browser to the Azure portal https://portal.azure.com.
2. Select your resource group, such as *myResourceGroup*, then select your AKS cluster, such as *myAKSCluster*.
3. Under **Monitoring** on the left-hand side, choose **Insights (preview)**
4. Across the top, choose to **+ Add Filter**
5. Select *Namespace* as the property, then choose *<All but kube-system>*
6. Choose to view the **Containers**.

The *azure-vote-back* and *azure-vote-front* containers are displayed, as shown in the following example:



To see logs for the `azure-vote-front` pod, select the **View container logs** link on the right-hand side of the containers list. These logs include the *stdout* and *stderr* streams from the container.



Delete cluster

When the cluster is no longer needed, use the az group delete command to remove the resource group, container service, and all related resources.

```
az group delete --name myAKSCluster --yes --no-wait
```

> **NOTE**
> When you delete the cluster, the Azure Active Directory service principal used by the AKS cluster is not removed. For steps on how to remove the service principal, see AKS service principal considerations and deletion.

Get the code

In this quickstart, pre-created container images have been used to create a Kubernetes deployment. The related application code, Dockerfile, and Kubernetes manifest file are available on GitHub.

https://github.com/Azure-Samples/azure-voting-app-redis

Next steps

In this quickstart, you deployed a Kubernetes cluster and deployed a multi-container application to it.

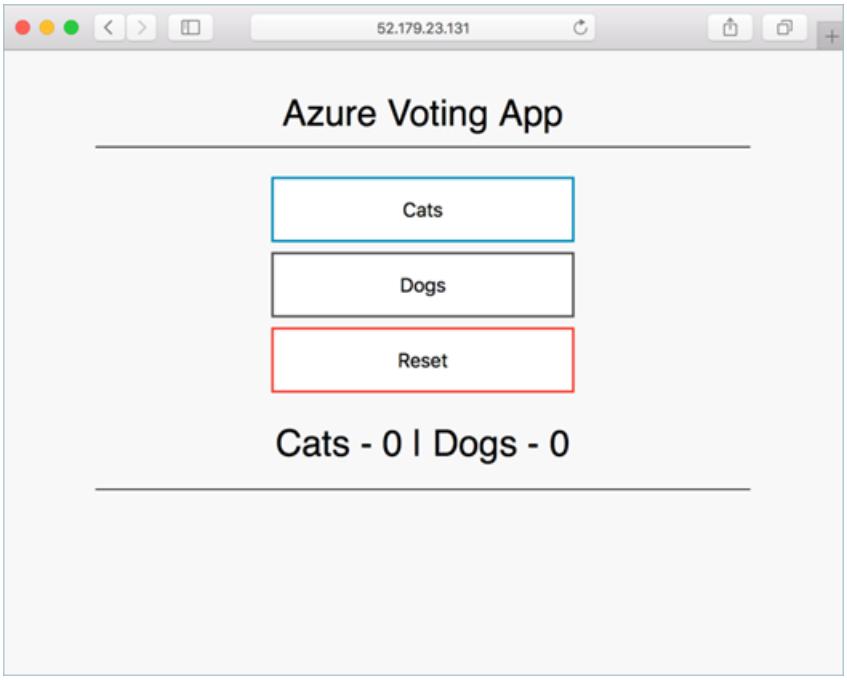
To learn more about AKS, and walk through a complete code to deployment example, continue to the Kubernetes cluster tutorial.

AKS tutorial

Quickstart: Deploy an Azure Kubernetes Service (AKS) cluster

9/24/2018 • 6 minutes to read • Edit Online

In this quickstart, you deploy an AKS cluster using the Azure portal. A multi-container application consisting of web front end and a Redis instance is then run on the cluster. Once completed, the application is accessible over the internet.



This quickstart assumes a basic understanding of Kubernetes concepts. For detailed information on Kubernetes, see the Kubernetes documentation.

Sign in to Azure

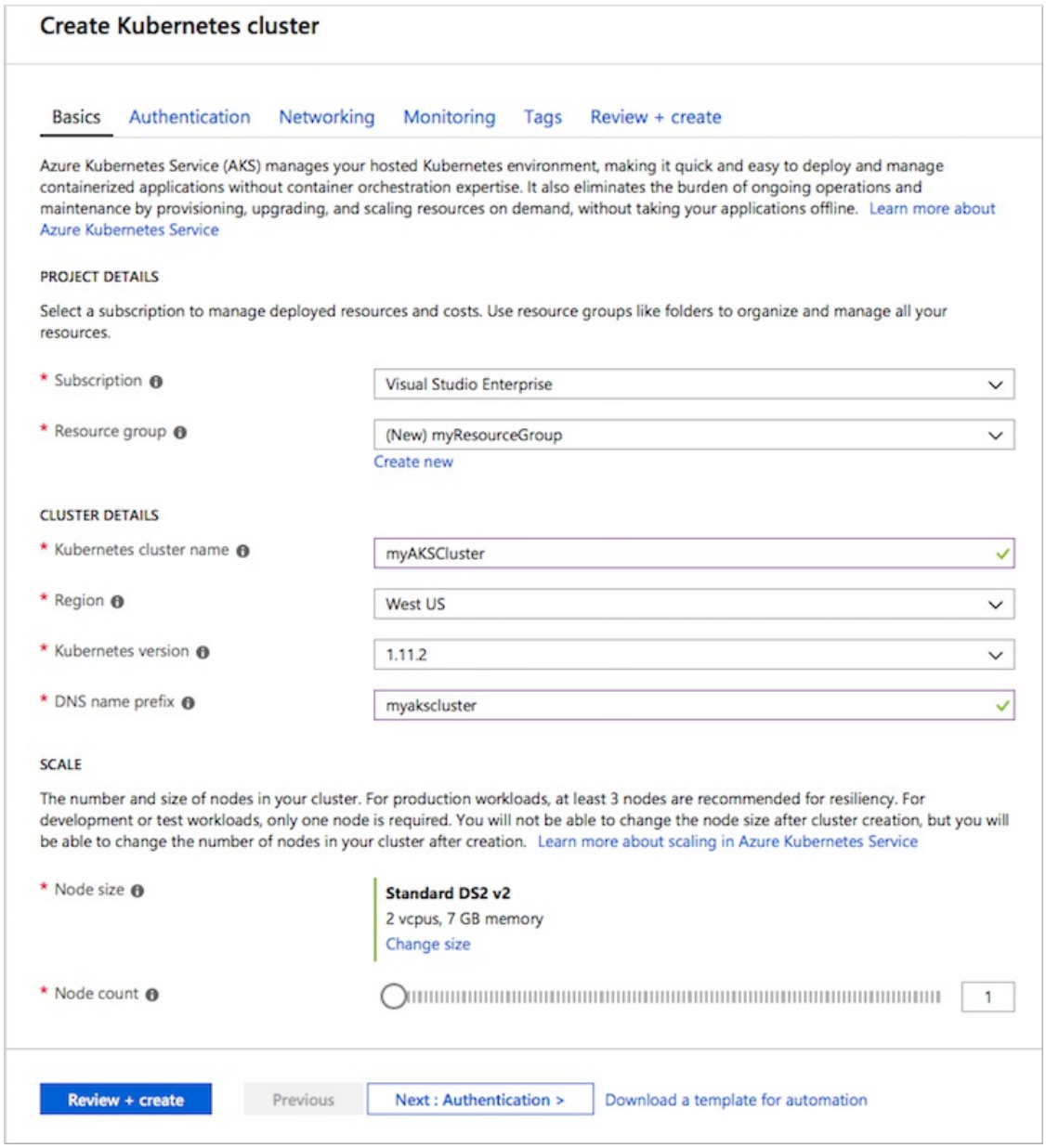
Sign in to the Azure portal at http://portal.azure.com.

Create an AKS cluster

In the top left-hand corner of the Azure portal, select **Create a resource** > **Kubernetes Service**.

To create an AKS cluster, complete the following steps:

1. **Basics** - Configure the following options:
   - *PROJECT DETAILS*: Select an Azure subscription, then select or create an Azure resource group, such as *myResourceGroup*. Enter a **Kubernetes cluster name**, such as *myAKSCluster*.
   - *CLUSTER DETAILS*: Select a region, Kubernetes version, and DNS name prefix for the AKS cluster.
   - *SCALE*: Select a VM size for the AKS nodes. The VM size **cannot** be changed once an AKS cluster has been deployed.
     - Select the number of nodes to deploy into the cluster. For this quickstart, set **Node count** to *1*. Node count **can** be adjusted after the cluster has been deployed.



Select **Next: Authentication** when complete.

2. **Authentication**: Configure the following options:

   - Create a new service principal or *Configure* to use an existing one. When using an existing SPN, you need to provide the SPN client ID and secret.
   - Enable the option for Kubernetes role-based access controls (RBAC). These controls provide more fine-grained control over access to the Kubernetes resources deployed in your AKS cluster.

   Select **Next: Networking** when complete.

3. **Networking**: Configure the following networking options, which should be set as default:

   - **Http application routing** - Select **Yes** to configure an integrated ingress controller with automatic public DNS name creation. For more information on Http routing, see, AKS HTTP routing and DNS.
   - **Network configuration** - Select the **Basic** network configuration using the kubenet Kubernetes plugin, rather than advanced networking configuration using Azure CNI. For more information on networking options, see AKS networking overview.

   Select **Next: Monitoring** when complete.

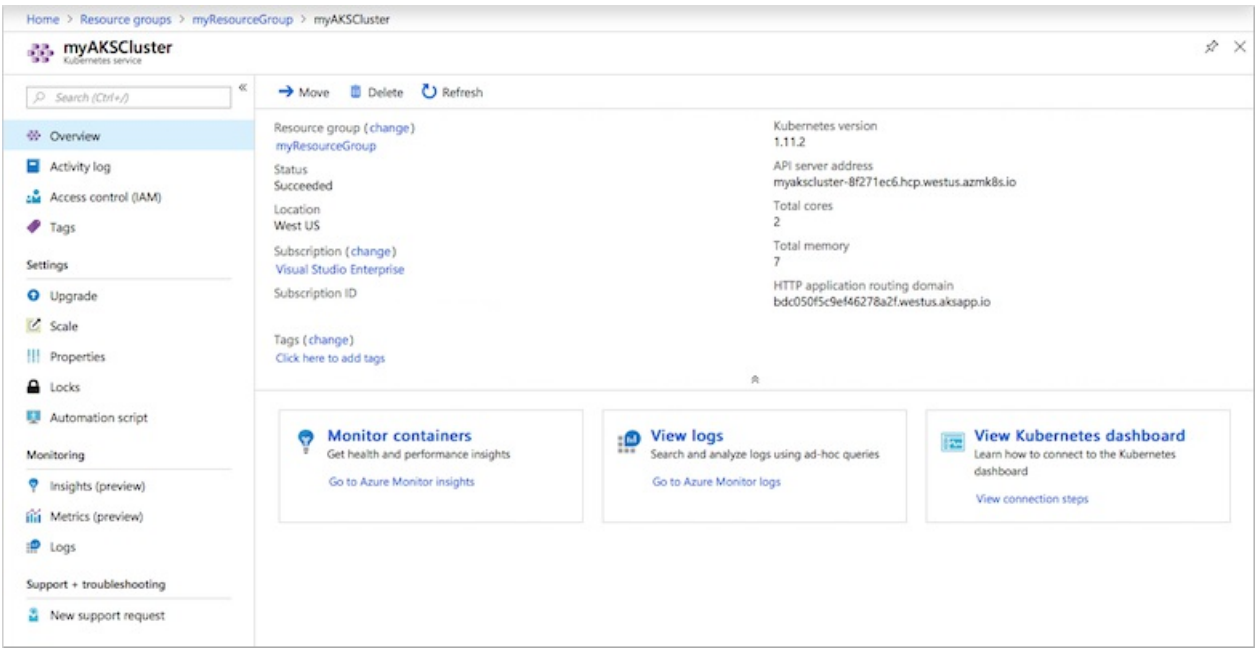
4. When deploying an AKS cluster, Azure Monitor for containers can be configured to monitor the health of the AKS cluster and pods running on the cluster. For more information on container health monitoring, see

Monitor Azure Kubernetes Service health.

Select **Yes** to enable container monitoring and select an existing Log Analytics workspace, or create a new one.

Select **Review + create** and then **Create** when ready.

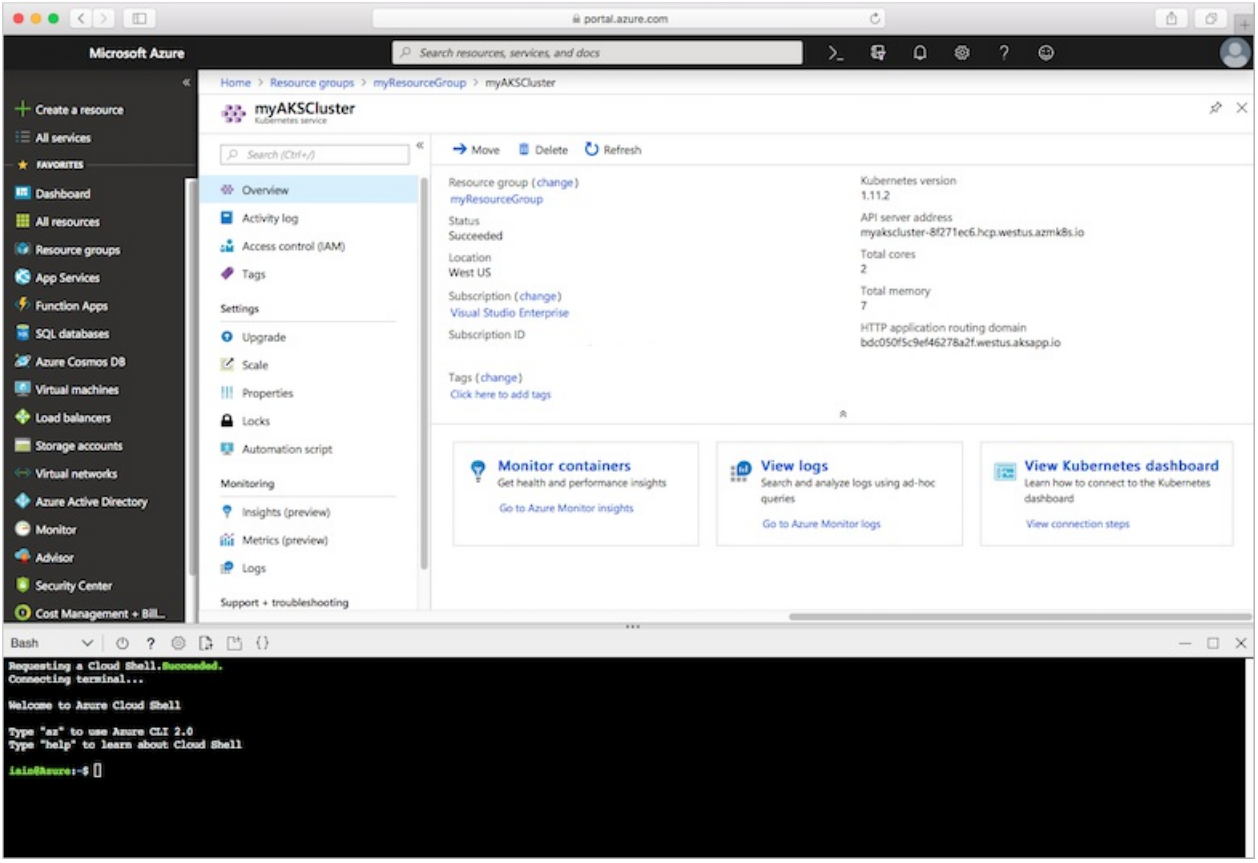
It takes a few minutes to create the AKS cluster and to be ready for use. Browse to the AKS cluster resource group, such as *myResourceGroup*, and select the AKS resource, such as *myAKSCluster*. The AKS cluster dashboard is shown, as in the following example screenshot:



Connect to the cluster

To manage a Kubernetes cluster, use kubectl, the Kubernetes command-line client. The `kubectl` client is pre-installed in the Azure Cloud Shell.

Open Cloud Shell using the button on the top right-hand corner of the Azure portal.



Use the az aks get-credentials command to configure `kubectl` to connect to your Kubernetes cluster. The following example gets credentials for the cluster name *myAKSCluster* in the resource group named *myResourceGroup*:

```
az aks get-credentials --resource-group myResourceGroup --name myAKSCluster
```

To verify the connection to your cluster, use the kubectl get command to return a list of the cluster nodes.

```
kubectl get nodes
```

The following example output shows the single node created in the previous steps.

```
NAME                       STATUS    ROLES     AGE       VERSION
aks-agentpool-14693408-0   Ready     agent     10m       v1.11.2
```

Run the application

Kubernetes manifest files define a desired state for a cluster, including what container images should be running. In this quickstart, a manifest is used to create all the objects needed to run a sample Azure Vote application. These objects include two Kubernetes deployments - one for the Azure Vote front end, and the other for a Redis instance. Also, two Kubernetes Services are created - an internal service for the Redis instance, and an external service for accessing the Azure Vote application from the internet.

Create a file named `azure-vote.yaml` and copy into it the following YAML code. If you are working in Azure Cloud Shell, create the file using `vi` or `Nano`, as if working on a virtual or physical system.

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: azure-vote-back
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: azure-vote-back
    spec:
      containers:
      - name: azure-vote-back
        image: redis
        ports:
        - containerPort: 6379
          name: redis
---
apiVersion: v1
kind: Service
metadata:
  name: azure-vote-back
spec:
  ports:
  - port: 6379
  selector:
    app: azure-vote-back
---
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: azure-vote-front
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: azure-vote-front
    spec:
      containers:
      - name: azure-vote-front
        image: microsoft/azure-vote-front:v1
        ports:
        - containerPort: 80
        env:
        - name: REDIS
          value: "azure-vote-back"
---
apiVersion: v1
kind: Service
metadata:
  name: azure-vote-front
spec:
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    app: azure-vote-front
```

Use the kubectl apply command to run the application.

```
kubectl create -f azure-vote.yaml
```

The following example output shows the Kubernetes resources created on your AKS cluster:

```
deployment "azure-vote-back" created
service "azure-vote-back" created
deployment "azure-vote-front" created
service "azure-vote-front" created
```

Test the application

As the application is run, a Kubernetes service is created to expose the application to the internet. This process can take a few minutes to complete.

To monitor progress, use the kubectl get service command with the `--watch` argument.

```
kubectl get service azure-vote-front --watch
```

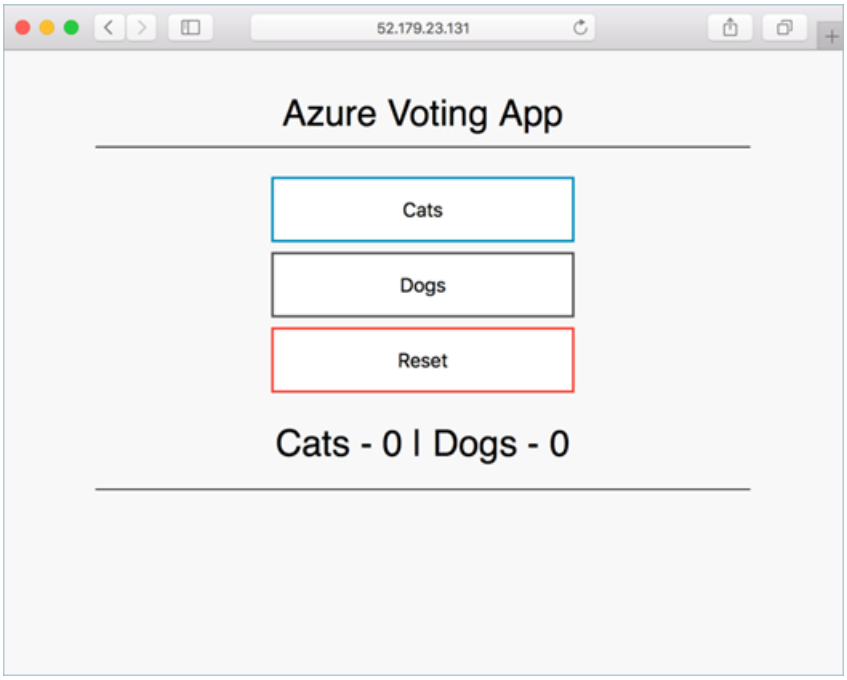
Initially, the *EXTERNAL-IP* for the *azure-vote-front* service appears as *pending*.

```
NAME              TYPE           CLUSTER-IP   EXTERNAL-IP   PORT(S)        AGE
azure-vote-front  LoadBalancer   10.0.37.27   <pending>     80:30572/TCP   6s
```

Once the *EXTERNAL-IP* address has changed from *pending* to an *IP address*, use `CTRL-C` to stop the kubectl watch process.

```
azure-vote-front   LoadBalancer   10.0.37.27   52.179.23.131   80:30572/TCP   2m
```

Open a web browser to the external IP address of your service to see the Azure Vote App, as shown in the following example:



Monitor health and logs

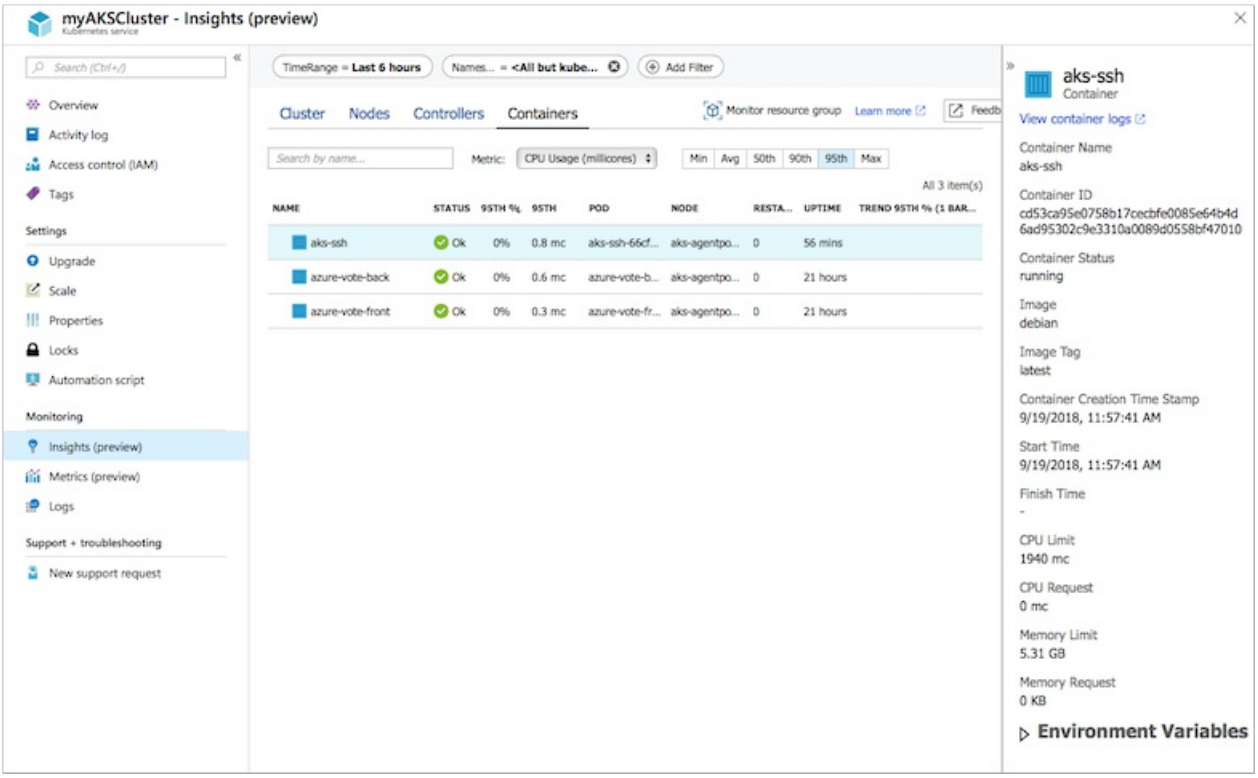
When you created the cluster, container insights monitoring was enabled. This monitoring feature provides health metrics for both the AKS cluster and pods running on the cluster. For more information on container health monitoring, see Monitor Azure Kubernetes Service health.

It may take a few minutes for this data to populate in the Azure portal. To see current status, uptime, and resource

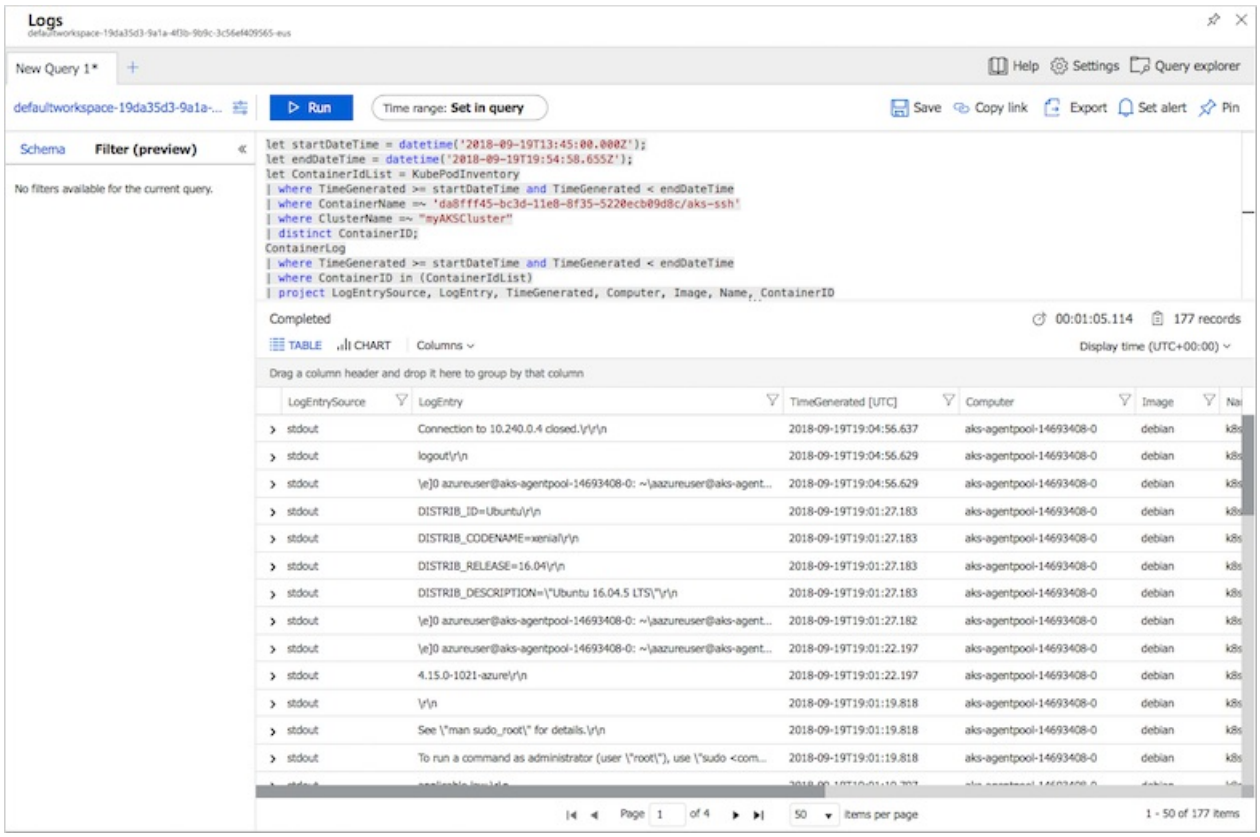
usage for the Azure Vote pods, browse back to the AKS resource in the Azure portal, such as *myAKSCluster*. You can then access the health status as follows:

1. Under **Monitoring** on the left-hand side, choose **Insights (preview)**
2. Across the top, choose to **+ Add Filter**
3. Select *Namespace* as the property, then choose *<All but kube-system>*
4. Choose to view the **Containers**.

The *azure-vote-back* and *azure-vote-front* containers are displayed, as shown in the following example:



To see logs for the `azure-vote-front` pod, select the **View container logs** link on the right-hand side of the containers list. These logs include the *stdout* and *stderr* streams from the container.



Delete cluster

When the cluster is no longer needed, delete the cluster resource, which deletes all associated resources. This operation can be completed in the Azure portal by selecting the **Delete** button on the AKS cluster dashboard. Alternatively, the az aks delete command can be used in the Cloud Shell:

```
az aks delete --resource-group myResourceGroup --name myAKSCluster --no-wait
```

> **NOTE**
>
> When you delete the cluster, the Azure Active Directory service principal used by the AKS cluster is not removed. For steps on how to remove the service principal, see AKS service principal considerations and deletion.

Get the code

In this quickstart, pre-created container images have been used to create a Kubernetes deployment. The related application code, Dockerfile, and Kubernetes manifest file are available on GitHub.

https://github.com/Azure-Samples/azure-voting-app-redis

Next steps

In this quickstart, you deployed a Kubernetes cluster and deployed a multi-container application to it.

To learn more about AKS, and walk through a complete code to deployment example, continue to the Kubernetes cluster tutorial.

AKS tutorial

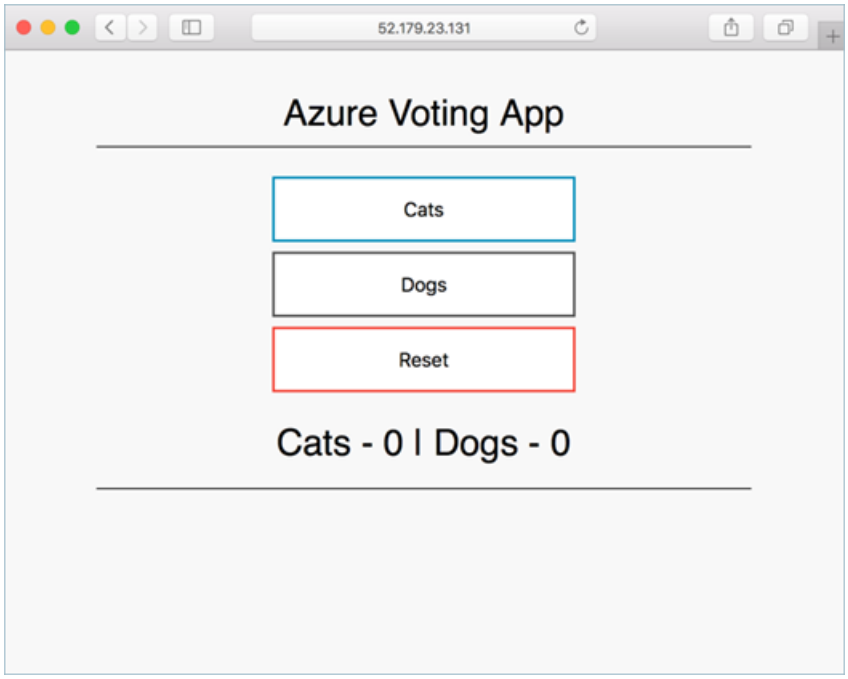
Tutorial: Prepare an application for Azure Kubernetes Service (AKS)

9/18/2018 • 3 minutes to read • Edit Online

In this tutorial, part one of seven, a multi-container application is prepared for use in Kubernetes. Existing development tools such as Docker Compose are used to locally build and test an application. You learn how to:

- Clone a sample application source from GitHub
- Create a container image from the sample application source
- Test the multi-container application in a local Docker environment

Once completed, the following application runs in your local development environment:



In subsequent tutorials, the container image is uploaded to an Azure Container Registry, and then deployed into an AKS cluster.

Before you begin

This tutorial assumes a basic understanding of core Docker concepts such as containers, container images, and `docker` commands. For a primer on container basics, see Get started with Docker.

To complete this tutorial, you need a local Docker development environment running Linux containers. Docker provides packages that configure Docker on a Mac, Windows, or Linux system.

Azure Cloud Shell does not include the Docker components required to complete every step in these tutorials. Therefore, we recommend using a full Docker development environment.

Get application code

The sample application used in this tutorial is a basic voting app. The application consists of a front-end web component and a back-end Redis instance. The web component is packaged into a custom container image. The Redis instance uses an unmodified image from Docker Hub.

Use git to clone the sample application to your development environment:

```
git clone https://github.com/Azure-Samples/azure-voting-app-redis.git
```

Change directories so that you are working from the cloned directory.

```
cd azure-voting-app-redis
```

Inside the directory is the application source code, a pre-created Docker compose file, and a Kubernetes manifest file. These files are used throughout the tutorial set.

Create container images

Docker Compose can be used to automate building container images and the deployment of multi-container applications.

Use the sample `docker-compose.yaml` file to create the container image, download the Redis image, and start the application:

```
docker-compose up -d
```

When completed, use the docker images command to see the created images. Three images have been downloaded or created. The *azure-vote-front* image contains the front-end application and uses the `nginx-flask` image as a base. The `redis` image is used to start a Redis instance.

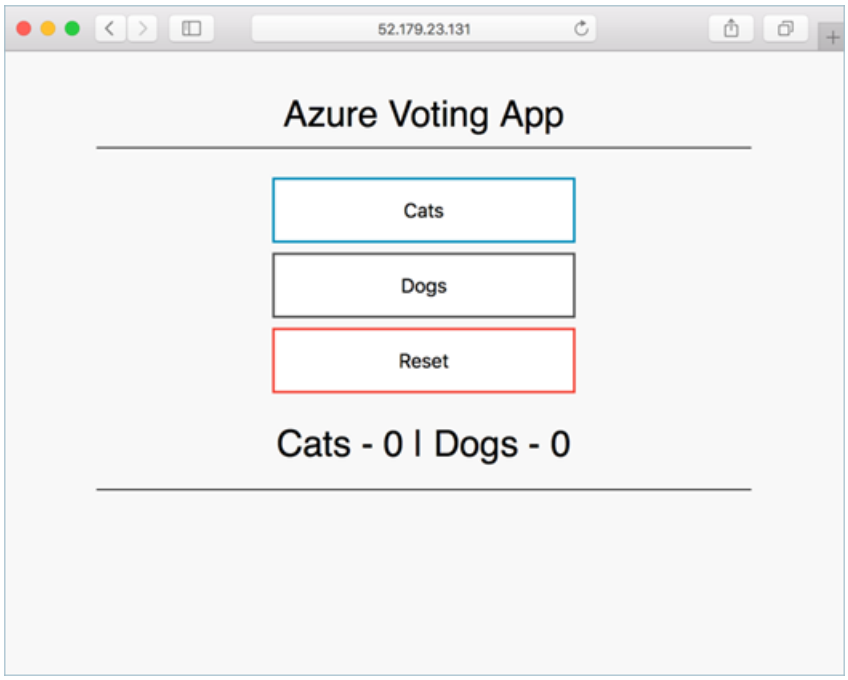
```
$ docker images

REPOSITORY                    TAG        IMAGE ID            CREATED             SIZE
azure-vote-front              latest     9cc914e25834        40 seconds ago      694MB
redis                         latest     a1b99da73d05        7 days ago          106MB
tiangolo/uwsgi-nginx-flask    flask      788ca94b2313        9 months ago        694MB
```

Run the docker ps command to see the running containers:

```
$ docker ps

CONTAINER ID        IMAGE             COMMAND                  CREATED             STATUS              PORTS
NAMES
82411933e8f9        azure-vote-front  "/usr/bin/supervisord"   57 seconds ago      Up 30 seconds
443/tcp, 0.0.0.0:8080->80/tcp   azure-vote-front
b68fed4b66b6        redis             "docker-entrypoint..."   57 seconds ago      Up 30 seconds
0.0.0.0:6379->6379/tcp          azure-vote-back
```

Test application locally

To see the running application, enter http://localhost:8080 in a local web browser. The sample application loads, as shown in the following example:



Clean up resources

Now that the application's functionality has been validated, the running containers can be stopped and removed. Do not delete the container images - in the next tutorial, the *azure-vote-front* image is uploaded to an Azure Container Registry instance.

Stop and remove the container instances and resources with the docker-compose down command:

```
docker-compose down
```

When the local application has been removed, you have a Docker image that contains the Azure Vote application, *azure-front-front*, for use with the next tutorial.

Next steps

In this tutorial, an application was tested and container images created for the application. You learned how to:

- Clone a sample application source from GitHub
- Create a container image from the sample application source
- Test the multi-container application in a local Docker environment

Advance to the next tutorial to learn how to store container images in Azure Container Registry.

Push images to Azure Container Registry

Tutorial: Deploy and use Azure Container Registry

8/14/2018 • 4 minutes to read • Edit Online

Azure Container Registry (ACR) is an Azure-based private registry for Docker container images. A private container registry lets you securely build and deploy your applications and custom code. In this tutorial, part two of seven, you deploy an ACR instance and push a container image to it. You learn how to:

- Create an Azure Container Registry (ACR) instance
- Tag a container image for ACR
- Upload the image to ACR
- View images in your registry

In subsequent tutorials, this ACR instance is integrated with a Kubernetes cluster in AKS, and an application is deployed from the image.

Before you begin

In the previous tutorial, a container image was created for a simple Azure Voting application. If you have not created the Azure Voting app image, return to Tutorial 1 – Create container images.

This tutorial requires that you are running the Azure CLI version 2.0.44 or later. Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI.

Create an Azure Container Registry

To create an Azure Container Registry, you first need a resource group. An Azure resource group is a logical container into which Azure resources are deployed and managed.

Create a resource group with the az group create command. In the following example, a resource group named *myResourceGroup* is created in the *eastus* region:

```
az group create --name myResourceGroup --location eastus
```

Create an Azure Container Registry instance with the az acr create command and provide your own registry name. The registry name must be unique within Azure, and contain 5-50 alphanumeric characters. In the rest of this tutorial, `<acrName>` is used as a placeholder for the container registry name. The *Basic* SKU is a cost-optimized entry point for development purposes that provides a balance of storage and throughput.

```
az acr create --resource-group myResourceGroup --name <acrName> --sku Basic
```

Log in to the container registry

To use the ACR instance, you must first log in. Use the az acr login command and provide the unique name given to the container registry in the previous step.

```
az acr login --name <acrName>
```

The command returns a *Login Succeeded* message once completed.

Tag a container image

To see a list of your current local images, use the docker images command:

```
$ docker images

REPOSITORY                    TAG                 IMAGE ID            CREATED             SIZE
azure-vote-front              latest              4675398c9172        13 minutes ago      694MB
redis                         latest              a1b99da73d05        7 days ago          106MB
tiangolo/uwsgi-nginx-flask    flask               788ca94b2313        9 months ago        694MB
```

To use the *azure-vote-front* container image with ACR, the image needs to be tagged with the login server address of your registry. This tag is used for routing when pushing container images to an image registry.

To get the login server address, use the az acr list command and query for the *loginServer* as follows:

```
az acr list --resource-group myResourceGroup --query "[].{acrLoginServer:loginServer}" --output table
```

Now, tag your local *azure-vote-front* image with the *acrloginServer* address of the container registry. To indicate the image version, add *:v1* to the end of the image name:

```
docker tag azure-vote-front <acrLoginServer>/azure-vote-front:v1
```

To verify the tags are applied, run docker images again. An image is tagged with the ACR instance address and a version number.

```
$ docker images

REPOSITORY                                          TAG                 IMAGE ID            CREATED             SIZE
azure-vote-front                                    latest              eaf2b9c57e5e        8 minutes ago       716
MB
mycontainerregistry.azurecr.io/azure-vote-front     v1                  eaf2b9c57e5e        8 minutes ago       716
MB
redis                                               latest              a1b99da73d05        7 days ago
106MB
tiangolo/uwsgi-nginx-flask                          flask               788ca94b2313        8 months ago        694
MB
```

Push images to registry

You can now push the *azure-vote-front* image to your ACR instance. Use docker push and provide your own *acrLoginServer* address for the image name as follows:

```
docker push <acrLoginServer>/azure-vote-front:v1
```

It may take a few minutes to complete the image push to ACR.

List images in registry

To return a list of images that have been pushed to your ACR instance, use the az acr repository list command. Provide your own `<acrName>` as follows:

```
az acr repository list --name <acrName> --output table
```

The following example output lists the *azure-vote-front* image as available in the registry:

```
Result
----------------
azure-vote-front
```

To see the tags for a specific image, use the az acr repository show-tags command as follows:

```
az acr repository show-tags --name <acrName> --repository azure-vote-front --output table
```

The following example output shows the *v1* image tagged in a previous step:

```
Result
--------
v1
```

You now have a container image that is stored in a private Azure Container Registry instance. This image is deployed from ACR to a Kubernetes cluster in the next tutorial.

Next steps

In this tutorial, you created an Azure Container Registry and pushed an image for use in an AKS cluster. You learned how to:

- Create an Azure Container Registry (ACR) instance
- Tag a container image for ACR
- Upload the image to ACR
- View images in your registry

Advance to the next tutorial to learn how to deploy a Kubernetes cluster in Azure.

Deploy Kubernetes cluster

Tutorial: Deploy an Azure Kubernetes Service (AKS) cluster

8/14/2018 • 3 minutes to read • Edit Online

Kubernetes provides a distributed platform for containerized applications. With AKS, you can quickly provision a production ready Kubernetes cluster. In this tutorial, part three of seven, a Kubernetes cluster is deployed in AKS. You learn how to:

- Create a service principal for resource interactions
- Deploy a Kubernetes AKS cluster
- Install the Kubernetes CLI (kubectl)
- Configure kubectl to connect to your AKS cluster

In subsequent tutorials, the Azure Vote application is deployed to the cluster, scaled, and updated.

Before you begin

In previous tutorials, a container image was created and uploaded to an Azure Container Registry instance. If you have not done these steps, and would like to follow along, return to Tutorial 1 – Create container images.

This tutorial requires that you are running the Azure CLI version 2.0.44 or later. Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI.

Create a service principal

To allow an AKS cluster to interact with other Azure resources, an Azure Active Directory service principal is used. This service principal can be automatically created by the Azure CLI or portal, or you can pre-create one and assign additional permissions. In this tutorial, you create a service principal, grant access to the Azure Container Registry (ACR) instance created in the previous tutorial, then create an AKS cluster.

Create a service principal using the az ad sp create-for-rbac command. The `--skip-assignment` parameter limits any additional permissions from being assigned.

```
az ad sp create-for-rbac --skip-assignment
```

The output is similar to the following example:

```
{
  "appId": "e7596ae3-6864-4cb8-94fc-20164b1588a9",
  "displayName": "azure-cli-2018-06-29-19-14-37",
  "name": "http://azure-cli-2018-06-29-19-14-37",
  "password": "52c95f25-bd1e-4314-bd31-d8112b293521",
  "tenant": "72f988bf-86f1-41af-91ab-2d7cd011db48"
}
```

Make a note of the *appId* and *password*. These values are used in the following steps.

Configure ACR authentication

To access images stored in ACR, you must grant the AKS service principal the correct rights to pull images from

ACR.

First, get the ACR resource ID using az acr show. Update the `<acrName>` registry name to that of your ACR instance and the resource group where the ACR instance is located.

```
az acr show --resource-group myResourceGroup --name <acrName> --query "id" --output tsv
```

To grant the correct access for the AKS cluster to use images stored in ACR, create a role assignment using the az role assignment create command. Replace `<appId` > and `<acrId>` with the values gathered in the previous two steps.

```
az role assignment create --assignee <appId> --scope <acrId> --role Reader
```

Create a Kubernetes cluster

AKS clusters can use Kubernetes role-based access controls (RBAC). These controls let you define access to resources based on roles assigned to users. Permissions can be combined if a user is assigned multiple roles, and permissions can be scoped to either a single namespace or across the whole cluster. Kubernetes RBAC is currently in preview for AKS clusters. By default, the Azure CLI automatically enables RBAC when you create an AKS cluster.

Create an AKS cluster using az aks create. The following example creates a cluster named *myAKSCluster* in the resource group named *myResourceGroup*. This resource group was created in the previous tutorial. Provide your own `<appId>` and `<password>` from the previous step where the service principal was created.

```
az aks create \
    --resource-group myResourceGroup \
    --name myAKSCluster \
    --node-count 1 \
    --service-principal <appId> \
    --client-secret <password> \
    --generate-ssh-keys
```

After several minutes, the deployment completes, and returns JSON-formatted information about the AKS deployment.

Install the Kubernetes CLI

To connect to the Kubernetes cluster from your local computer, you use kubectl, the Kubernetes command-line client.

If you use the Azure Cloud Shell, `kubectl` is already installed. You can also install it locally using the az aks install-cli command:

```
az aks install-cli
```

Connect to cluster using kubectl

To configure `kubectl` to connect to your Kubernetes cluster, use az aks get-credentials. The following example gets credentials for the AKS cluster name *myAKSCluster* in the *myResourceGroup*:

```
az aks get-credentials --resource-group myResourceGroup --name myAKSCluster
```

To verify the connection to your cluster, run the kubectl get nodes command:

```
$ kubectl get nodes

NAME                       STATUS    ROLES     AGE       VERSION
aks-nodepool1-66427764-0   Ready     agent     9m        v1.9.9
```

Next steps

In this tutorial, a Kubernetes cluster was deployed in AKS, and you configured `kubectl` to connect to it. You learned how to:

- Create a service principal for resource interactions
- Deploy a Kubernetes AKS cluster
- Install the Kubernetes CLI (kubectl)
- Configure kubectl to connect to your AKS cluster

Advance to the next tutorial to learn how to deploy an application to the cluster.

Deploy application in Kubernetes

Tutorial: Run applications in Azure Kubernetes Service (AKS)

8/14/2018 • 3 minutes to read • Edit Online

Kubernetes provides a distributed platform for containerized applications. You build and deploy your own applications and services into a Kubernetes cluster, and let the cluster manage the availability and connectivity. In this tutorial, part four of seven, a sample application is deployed into a Kubernetes cluster. You learn how to:

- Update a Kubernetes manifest files
- Run an application in Kubernetes
- Test the application

In subsequent tutorials, this application is scaled out and updated.

This tutorial assumes a basic understanding of Kubernetes concepts, for detailed information on Kubernetes see the Kubernetes documentation.

Before you begin

In previous tutorials, an application was packaged into a container image, this image was uploaded to Azure Container Registry, and a Kubernetes cluster was created.

To complete this tutorial, you need the pre-created `azure-vote-all-in-one-redis.yaml` Kubernetes manifest file. This file was downloaded with the application source code in a previous tutorial. Verify that you have cloned the repo, and that you have changed directories into the cloned repo. If you have not done these steps, and would like to follow along, return to Tutorial 1 – Create container images.

This tutorial requires that you are running the Azure CLI version 2.0.44 or later. Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI.

Update the manifest file

In these tutorials, an Azure Container Registry (ACR) instance stores the container image for the sample application. To deploy the application, you must update the image name in the Kubernetes manifest file to include the ACR login server name.

Get the ACR login server name using the az acr list command as follows:

```
az acr list --resource-group myResourceGroup --query "[].{acrLoginServer:loginServer}" --output table
```

The sample manifest file from the git repo cloned in the first tutorial uses the login server name of *microsoft*. Open this manifest file with a text editor, such as `vi`:

```
vi azure-vote-all-in-one-redis.yaml
```

Replace *microsoft* with your ACR login server name. The image name is found on line 47 of the manifest file. The following example shows the default image name:

```
containers:
- name: azure-vote-front
  image: microsoft/azure-vote-front:v1
```

Provide your own ACR login server name so that your manifest file looks like the following example:

```
containers:
- name: azure-vote-front
  image: <acrName>.azurecr.io/azure-vote-front:v1
```

Save and close the file.

Deploy the application

To deploy your application, use the kubectl apply command. This command parses the manifest file and creates the defined Kubernetes objects. Specify the sample manifest file, as shown in the following example:

```
kubectl apply -f azure-vote-all-in-one-redis.yaml
```

The Kubernetes objects are created within the cluster, as shown in the following example:

```
$ kubectl apply -f azure-vote-all-in-one-redis.yaml

deployment "azure-vote-back" created
service "azure-vote-back" created
deployment "azure-vote-front" created
service "azure-vote-front" created
```

Test the application

A Kubernetes service is created which exposes the application to the internet. This process can take a few minutes. To monitor progress, use the kubectl get service command with the `--watch` argument:

```
kubectl get service azure-vote-front --watch
```

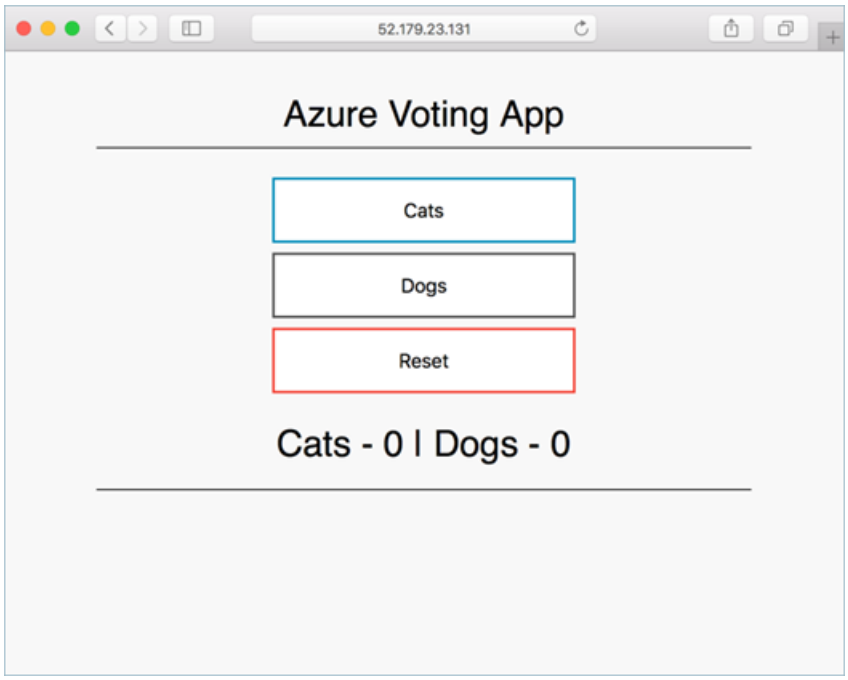
The *EXTERNAL-IP* for the *azure-vote-front* service initially appears as *pending*, as shown in the following example:

```
azure-vote-front   10.0.34.242   <pending>     80:30676/TCP   7s
```

When the *EXTERNAL-IP* address changes from *pending* to an actual public IP address, use `CTRL-C` to stop the kubectl watch process. The following example shows a public IP address is now assigned:

```
azure-vote-front   10.0.34.242   52.179.23.131   80:30676/TCP   2m
```

To see the application in action, open a web browser to the external IP address.



If the application did not load, it might be due to an authorization problem with your image registry. To view the status of your containers, use the `kubectl get pods` command. If the container images cannot be pulled, see allow access to Container Registry with a Kubernetes secret.

Next steps

In this tutorial, the Azure vote application was deployed to a Kubernetes cluster in AKS. You learned how to:

- Update a Kubernetes manifest files
- Run an application in Kubernetes
- Test the application

Advance to the next tutorial to learn how to scale a Kubernetes application and the underlying Kubernetes infrastructure.

Scale Kubernetes application and infrastructure

Tutorial: Scale applications in Azure Kubernetes Service (AKS)

8/14/2018 • 3 minutes to read • Edit Online

If you've been following the tutorials, you have a working Kubernetes cluster in AKS and you deployed the Azure Voting app. In this tutorial, part five of seven, you scale out the pods in the app and try pod autoscaling. You also learn how to scale the number of Azure VM nodes to change the cluster's capacity for hosting workloads. You learn how to:

- Scale the Kubernetes nodes
- Manually scale Kubernetes pods that run your application
- Configure autoscaling pods that run the app front-end

In subsequent tutorials, the Azure Vote application is updated to a new version.

Before you begin

In previous tutorials, an application was packaged into a container image, this image uploaded to Azure Container Registry, and a Kubernetes cluster created. The application was then run on the Kubernetes cluster. If you have not done these steps, and would like to follow along, return to the Tutorial 1 – Create container images.

This tutorial requires that you are running the Azure CLI version 2.0.38 or later. Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI.

Manually scale pods

When the Azure Vote front-end and Redis instance were deployed in previous tutorials, a single replica was created. To see the number and state of pods in your cluster, use the kubectl get command as follows:

```
kubectl get pods
```

The following example output shows one front-end pod and one back-end pod:

```
NAME                               READY     STATUS    RESTARTS   AGE
azure-vote-back-2549686872-4d2r5   1/1       Running   0          31m
azure-vote-front-848767080-tf34m   1/1       Running   0          31m
```

To manually change the number of pods in the *azure-vote-front* deployment, use the kubectl scale command. The following example increases the number of front-end pods to *5*:

```
kubectl scale --replicas=5 deployment/azure-vote-front
```

Run kubectl get pods again to verify that Kubernetes creates the additional pods. After a minute or so, the additional pods are available in your cluster:

```
$ kubectl get pods

                                  READY     STATUS    RESTARTS   AGE
azure-vote-back-2606967446-nmpcf  1/1       Running   0          15m
azure-vote-front-3309479140-2hfh0 1/1       Running   0          3m
azure-vote-front-3309479140-bzt05 1/1       Running   0          3m
azure-vote-front-3309479140-fvcvm 1/1       Running   0          3m
azure-vote-front-3309479140-hrbf2 1/1       Running   0          15m
azure-vote-front-3309479140-qphz8 1/1       Running   0          3m
```

Autoscale pods

Kubernetes supports horizontal pod autoscaling to adjust the number of pods in a deployment depending on CPU utilization or other select metrics. The Metrics Server is used to provide resource utilization to Kubernetes. To install the Metrics Server, clone the `metrics-server` GitHub repo and install the example resource definitions. To view the contents of these YAML definitions, see Metrics Server for Kuberenetes 1.8+.

```
git clone https://github.com/kubernetes-incubator/metrics-server.git
kubectl create -f metrics-server/deploy/1.8+/
```

To use the autoscaler, your pods must have CPU requests and limits defined. In the `azure-vote-front` deployment, the front-end container requests 0.25 CPU, with a limit of 0.5 CPU. The settings look like:

```
resources:
  requests:
     cpu: 250m
  limits:
     cpu: 500m
```

The following example uses the kubectl autoscale command to autoscale the number of pods in the *azure-vote-front* deployment. If CPU utilization exceeds 50%, the autoscaler increases the pods up to a maximum of 10 instances:

```
kubectl autoscale deployment azure-vote-front --cpu-percent=50 --min=3 --max=10
```

To see the status of the autoscaler, use the `kubectl get hpa` command as follows:

```
$ kubectl get hpa

NAME               REFERENCE                     TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
azure-vote-front   Deployment/azure-vote-front   0% / 50%   3         10        3          2m
```

After a few minutes, with minimal load on the Azure Vote app, the number of pod replicas decreases automatically to three. You can use `kubectl get pods` again to see the unneeded pods being removed.

Manually scale AKS nodes

If you created your Kubernetes cluster using the commands in the previous tutorial, it has one node. You can adjust the number of nodes manually if you plan more or fewer container workloads on your cluster.

The following example increases the number of nodes to three in the Kubernetes cluster named *myAKSCluster*. The command takes a couple of minutes to complete.

```
az aks scale --resource-group=myResourceGroup --name=myAKSCluster --node-count 3
```

The output is similar to:

```
"agentPoolProfiles": [
  {
    "count": 3,
    "dnsPrefix": null,
    "fqdn": null,
    "name": "myAKSCluster",
    "osDiskSizeGb": null,
    "osType": "Linux",
    "ports": null,
    "storageProfile": "ManagedDisks",
    "vmSize": "Standard_D2_v2",
    "vnetSubnetId": null
  }
```

Next steps

In this tutorial, you used different scaling features in your Kubernetes cluster. You learned how to:

- Scale the Kubernetes nodes
- Manually scale Kubernetes pods that run your application
- Configure autoscaling pods that run the app front-end

Advance to the next tutorial to learn how to update application in Kubernetes.

Update an application in Kubernetes

Tutorial: Update an application in Azure Kubernetes Service (AKS)

8/14/2018 • 3 minutes to read • Edit Online

After an application has been deployed in Kubernetes, it can be updated by specifying a new container image or image version. When doing so, the update is staged so that only a portion of the deployment is concurrently updated. This staged update enables the application to keep running during the update. It also provides a rollback mechanism if a deployment failure occurs.

In this tutorial, part six of seven, the sample Azure Vote app is updated. You learn how to:

- Update the front-end application code
- Create an updated container image
- Push the container image to Azure Container Registry
- Deploy the updated container image

Before you begin

In previous tutorials, an application was packaged into a container image, the image uploaded to Azure Container Registry (ACR), and a Kubernetes cluster created. The application was then run on the Kubernetes cluster.

An application repository was also cloned that includes the application source code, and a pre-created Docker Compose file used in this tutorial. Verify that you have created a clone of the repo, and that you have changed directories into the cloned directory. If you haven't completed these steps, and want to follow along, return to Tutorial 1 – Create container images.

This tutorial requires that you are running the Azure CLI version 2.0.44 or later. Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI.

Update an application

Let's make a change to the sample application, then update the version already deployed to your AKS cluster. The sample application source code can be found inside of the *azure-vote* directory. Open the *config_file.cfg* file with an editor, such as `vi`:

```
vi azure-vote/azure-vote/config_file.cfg
```

Change the values for *VOTE1VALUE* and *VOTE2VALUE* to different colors. The following example shows the updated color values:

```
# UI Configurations
TITLE = 'Azure Voting App'
VOTE1VALUE = 'Blue'
VOTE2VALUE = 'Purple'
SHOWHOST = 'false'
```

Save and close the file.

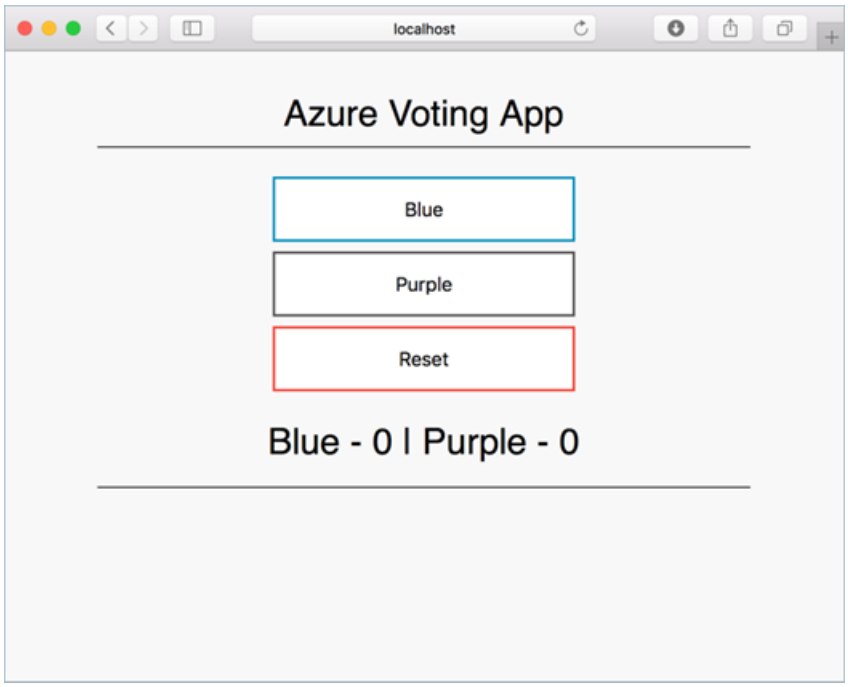
Update the container image

To re-create the front-end image and test the updated application, use docker-compose. The `--build` argument is used to instruct Docker Compose to re-create the application image:

```
docker-compose up --build -d
```

Test the application locally

To verify that the updated container image shows your changes, open a local web browser to http://localhost:8080.



The updated color values provided in the *config_file.cfg* file are displayed on your running application.

Tag and push the image

To correctly use the updated image, tag the *azure-vote-front* image with the login server name of your ACR registry. Get the login server name with the az acr list command:

```
az acr list --resource-group myResourceGroup --query "[].{acrLoginServer:loginServer}" --output table
```

Use docker tag to tag the image. Replace `<acrLoginServer>` with your ACR login server name or public registry hostname, and update the image version to *:v2* as follows:

```
docker tag azure-vote-front <acrLoginServer>/azure-vote-front:v2
```

Now use docker push to upload the image to your registry. Replace `<acrLoginServer>` with your ACR login server name. If you experience issues pushing to your ACR registry, ensure that you have run the az acr login command.

```
docker push <acrLoginServer>/azure-vote-front:v2
```

Deploy the updated application

To ensure maximum uptime, multiple instances of the application pod must be running. Verify the number of running front-end instances with the kubectl get pods command:

```
$ kubectl get pods

NAME                                READY     STATUS    RESTARTS   AGE
azure-vote-back-217588096-5w632     1/1       Running   0          10m
azure-vote-front-233282510-b5pkz    1/1       Running   0          10m
azure-vote-front-233282510-dhrtr    1/1       Running   0          10m
azure-vote-front-233282510-pqbfk    1/1       Running   0          10m
```

If you do not have multiple front-end pods, scale the *azure-vote-front* deployment as follows:

```
kubectl scale --replicas=3 deployment/azure-vote-front
```

To update the application, use the kubectl set command. Update `<acrLoginServer>` with the login server or host name of your container registry, and specify the *v2* application version:

```
kubectl set image deployment azure-vote-front azure-vote-front=<acrLoginServer>/azure-vote-front:v2
```

To monitor the deployment, use the kubectl get pod command. As the updated application is deployed, your pods are terminated and re-created with the new container image.

```
kubectl get pods
```

The following example output shows pods terminating and new instances running as the deployment progresses:

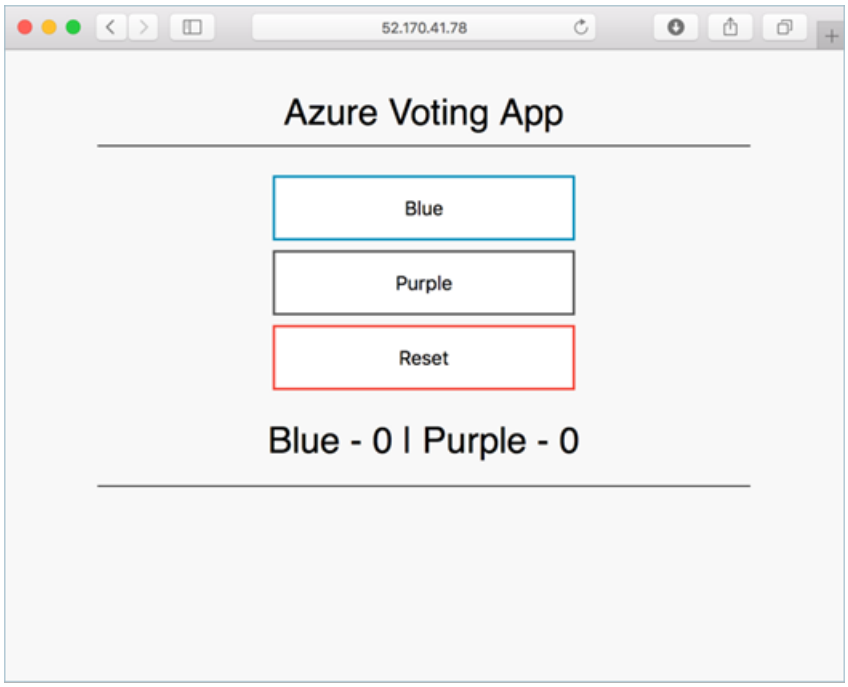
```
$ kubectl get pods

NAME                                READY     STATUS        RESTARTS   AGE
azure-vote-back-2978095810-gq9g0    1/1       Running       0          5m
azure-vote-front-1297194256-tpjlg   1/1       Running       0          1m
azure-vote-front-1297194256-tptnx   1/1       Running       0          5m
azure-vote-front-1297194256-zktw9   1/1       Terminating   0          1m
```

Test the updated application

To view the update application, first get the external IP address of the `azure-vote-front` service:

```
kubectl get service azure-vote-front
```

Now open a local web browser to the IP address.



Next steps

In this tutorial, you updated an application and rolled out this update to a Kubernetes cluster. You learned how to:

- Update the front-end application code
- Create an updated container image
- Push the container image to Azure Container Registry
- Deploy the updated container image

Advance to the next tutorial to learn how to upgrade an AKS cluster to a new version of Kubernetes.

Upgrade Kubernetes

Tutorial: Upgrade Kubernetes in Azure Kubernetes Service (AKS)

10/9/2018 • 2 minutes to read • Edit Online

As part of the application and cluster lifecycle, you may wish to upgrade to the latest available version of Kubernetes and use new features. An Azure Kubernetes Service (AKS) cluster can be upgraded using the Azure CLI. To minimize disruption to running applications, Kubernetes nodes are carefully cordoned and drained during the upgrade process.

In this tutorial, part seven of seven, a Kubernetes cluster is upgraded. You learn how to:

- Identify current and available Kubernetes versions
- Upgrade the Kubernetes nodes
- Validate a successful upgrade

Before you begin

In previous tutorials, an application was packaged into a container image, this image uploaded to Azure Container Registry, and a Kubernetes cluster created. The application was then run on the Kubernetes cluster. If you have not done these steps, and would like to follow along, return to the Tutorial 1 – Create container images.

This tutorial requires that you are running the Azure CLI version 2.0.44 or later. Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI.

Get available cluster versions

Before you upgrade a cluster, use the az aks get-upgrades command to check which Kubernetes releases are available for upgrade:

```
az aks get-upgrades --resource-group myResourceGroup --name myAKSCluster --output table
```

In the following example, the current version is *1.9.6*, and the available versions are shown under the *Upgrades* column.

```
Name     ResourceGroup    MasterVersion    NodePoolVersion    Upgrades
-------  ---------------  ---------------  -----------------  ----------------------
default  myResourceGroup  1.9.9            1.9.9              1.10.3, 1.10.5, 1.10.6
```

Upgrade a cluster

Use the az aks upgrade command to upgrade the AKS cluster. The following example upgrades the cluster to Kubernetes version *1.10.6*.

> **NOTE**
>
> You can only upgrade one minor version at a time. For example, you can upgrade from *1.9.6* to *1.10.3*, but cannot upgrade from *1.9.6* to *1.11.x* directly. To upgrade from *1.9.6* to *1.11.x*, first upgrade from *1.9.6* to *1.10.3*, then perform another upgrade from *1.10.3* to *1.11.x*.

```
az aks upgrade --resource-group myResourceGroup --name myAKSCluster --kubernetes-version 1.10.6
```

The following condensed example output shows the *kubernetesVersion* now reports *1.10.6*:

```
{
  "agentPoolProfiles": [
    {
      "count": 3,
      "maxPods": 110,
      "name": "nodepool1",
      "osType": "Linux",
      "storageProfile": "ManagedDisks",
      "vmSize": "Standard_DS1_v2",
    }
  ],
  "dnsPrefix": "myAKSClust-myResourceGroup-19da35",
  "enableRbac": false,
  "fqdn": "myaksclust-myresourcegroup-19da35-bd54a4be.hcp.eastus.azmk8s.io",
  "id": "/subscriptions/<Subscription
ID>/resourcegroups/myResourceGroup/providers/Microsoft.ContainerService/managedClusters/myAKSCluster",
  "kubernetesVersion": "1.10.6",
  "location": "eastus",
  "name": "myAKSCluster",
  "type": "Microsoft.ContainerService/ManagedClusters"
}
```

Validate an upgrade

Confirm that the upgrade was successful using the az aks show command as follows:

```
az aks show --resource-group myResourceGroup --name myAKSCluster --output table
```

The following example output shows the AKS cluster runs *KubernetesVersion 1.10.6*:

```
Name          Location    ResourceGroup    KubernetesVersion    ProvisioningState    Fqdn
------------  ----------  ---------------  -------------------  -------------------  -------------------------
--------------------------------------
myAKSCluster  eastus      myResourceGroup  1.10.6               Succeeded            myaksclust-
myresourcegroup-19da35-bd54a4be.hcp.eastus.azmk8s.io
```

Delete the cluster

As this is the last part of the tutorial series, you may want to delete the AKS cluster. As the Kubernetes nodes run on Azure virtual machines (VMs), they continue to incur charges even if you don't use the cluster. Use the az group delete command to remove the resource group, container service, and all related resources.

```
az group delete --name myResourceGroup --yes --no-wait
```

NOTE

When you delete the cluster, the Azure Active Directory service principal used by the AKS cluster is not removed. For steps on how to remove the service principal, see AKS service principal considerations and deletion.

Next steps

In this tutorial, you upgraded Kubernetes in an AKS cluster. You learned how to:

- Identify current and available Kubernetes versions
- Upgrade the Kubernetes nodes
- Validate a successful upgrade

Follow this link to learn more about AKS.

AKS overview

Quotas and region availability for Azure Kubernetes Service (AKS)

8/2/2018 • 2 minutes to read • Edit Online

All Azure services include certain default limits and quotas for resources and features. The following sections detail the default resource limits for several Azure Kubernetes Service (AKS) resources, as well as the availability of the AKS service in Azure regions.

Service quotas and limits

RESOURCE	DEFAULT LIMIT
Max clusters per subscription	100
Max nodes per cluster	100
Max pods per node: Basic networking with Kubenet	110
Max pods per node: Advanced networking with Azure CNI	Azure CLI deployment: 30[1] Resource Manager template: 30[1] Portal deployment: 30

[1] This value is configurable at cluster deployment when deploying an AKS cluster with the Azure CLI or a Resource Manager template.

Provisioned infrastructure

All other network, compute, and storage limitations apply to the provisioned infrastructure. See Azure subscription and service limits for the relevant limits.

Region availability

Azure Kubernetes Service (AKS) is available in the following regions:

- Australia East
- Canada Central
- Canada East
- Central US
- East US
- East US2
- Japan East
- North Europe
- Southeast Asia
- UK South
- West Europe
- West US
- West US 2

Next steps

Certain default limits and quotas can be increased. To request an increase of one or more resources that support such an increase, submit an Azure support request (select "Quota" for **Issue type**).

Migrating from Azure Container Service (ACS) to Azure Kubernetes Service (AKS)

8/29/2018 • 5 minutes to read • Edit Online

The goal of this document is to help you plan and execute a successful migration between Azure Container Service with Kubernetes (ACS) and Azure Kubernetes Service (AKS). This guide details the differences between ACS and AKS, provides an overview of the migration process, and should help you make key decisions.

Differences between ACS and AKS

ACS and AKS differ in some key areas that impact migration. You should review and plan to address the following differences before any migration.

- AKS nodes use Managed Disks
  - Unmanaged disks will need to be converted before they can be attached to AKS nodes
  - Custom `StorageClass` objects for Azure disks will need to be changed from `unmanaged` to `managed`
  - Any `PersistentVolumes` will need to use `kind: Managed`
- AKS currently supports only one agent pool
- Windows Server-based nodes are currently in private preview
- Check the list of AKS supported regions
- AKS is a managed service with a hosted Kubernetes control plane. You may need to modify your applications if you've previously modified the configuration of your ACS masters

Differences between Kubernetes versions

If you're migrating to a newer version of Kubernetes (ex: 1.7.x to 1.9.x), there are a few changes to the k8s API that will require your attention.

- Migrate a ThirdPartyResource to CustomResourceDefinition
- Workloads API changes in versions 1.8 and 1.9.

Migration considerations

Agent Pools

While AKS manages the Kubernetes control plane, you still define the size and number of nodes you want to include in your new cluster. Assuming you want a 1:1 mapping from ACS to AKS, you'll want to capture your existing ACS node information. You'll use this data when creating your new AKS cluster.

Example:

NAME	COUNT	VM SIZE	OPERATING SYSTEM
agentpool0	3	Standard_D8_v2	Linux
agentpool1	1	Standard_D2_v2	Windows

Because additional virtual machines will be deployed into your subscription during migration, you should verify that your quotas and limits are sufficient for these resources. You can learn more by reviewing Azure subscription and service limits. To check your current quotas, go to the subscriptions blade in the Azure portal, select your

subscription, then select `Usage + quotas`.

Networking

For complex applications, you'll typically migrate over time rather than all at once. That means that the old and new environments may need to communicate over the network. Applications that were previously able to use `ClusterIP` services to communicate may need to be exposed as type `LoadBalancer` and secured appropriately.

To complete the migration, you'll want to point clients to the new services running on AKS. The recommended way to redirect traffic is by updating DNS to point to the Load Balancer that sits in front of your AKS cluster.

Stateless Applications

Stateless application migration is the most straightforward case. You'll apply your YAML definitions to the new cluster, validate that everything is working as expected, and redirect traffic to make your new cluster active.

Stateful Applications

Migrating stateful applications requires careful planning to avoid data loss or unexpected downtime.

Highly Available Applications

Some stateful applications can be deployed in a high availability configuration and can copy data across replicas. If this describes your current deployment, it may be possible to create a new member on the new AKS cluster, and migrate with minimal impact to downstream callers. The migration steps for this scenario generally are:

1. Create a new secondary replica on AKS
2. Wait for data to replicate
3. Fail over to make secondary replica the new primary
4. Point traffic to the AKS cluster

Migrating Persistent Volumes

There are several factors to consider if you're migrating existing Persistent Volumes to AKS. Generally, the steps involved are:

1. (Optional) Quiesce writes to the application (requires downtime)
2. Snapshot disks
3. Create new Managed Disks from snapshots
4. Create Persistent Volumes in AKS
5. Update Pod specifications to use existing volumes rather than PersistentVolumeClaims (static provisioning)
6. Deploy application to AKS
7. Validate
8. Point traffic to the AKS cluster

> **Important**: If you choose not to quiesce writes, you'll need to replicate data to the new deployment, as you'll be missing data that was written since the disk snapshot

Open-source tools exist that can help you create Managed Disks and migrate volumes between Kubernetes clusters.

- noelbundick/azure-cli-disk-extension - copy and convert disks across Resource Groups and Azure regions
- yaron2/azure-kube-cli - enumerate ACS Kubernetes volumes and migrate them to an AKS cluster

Azure Files

Unlike disks, Azure Files can be mounted to multiple hosts concurrently. Neither Azure nor Kubernetes prevents you from creating a Pod in your AKS cluster that is still being used by your ACS cluster. To prevent data loss and unexpected behavior, you should ensure that both clusters aren't writing to the same files at the same time.

If your application can host multiple replicas pointing to the same file share, you can follow the stateless migration

steps and deploy your YAML definitions to your new cluster.

If not, one possible migration approach involves the following steps:

1. Deploy your application to AKS with a replica count of 0
2. Scale the application on ACS to 0 (requires downtime)
3. Scale the application on AKS up to 1
4. Validate
5. Point traffic to the AKS cluster

In cases where you'd like to start with an empty share, then make a copy of the source data, you can use the `az storage file copy` commands to migrate your data.

Deployment Strategy

The recommended method is to use your existing CI/CD pipeline to deploy a known-good configuration to AKS. You'll clone your existing deploy tasks, and ensure that your `kubeconfig` points to the new AKS cluster.

In cases where that's not possible, you'll need to export resource definition from ACS, and then apply them to AKS. You can use `kubectl` to export objects.

```
kubectl get deployment -o=yaml --export > deployments.yaml
```

There are also several open-source tools that can help, depending on your needs:

- heptio/ark - requires k8s 1.7
- yaron2/azure-kube-cli
- mhausenblas/reshifter

Migration steps

1. Create an AKS cluster

You can follow the docs to create an AKS cluster via the Azure portal, Azure CLI, or Resource Manager template.

> You can find sample Azure Resource Manager templates for AKS at the Azure/AKS repository on GitHub

2. Modify applications

Make any necessary modifications to your YAML definitions. Ex: replacing `apps/v1beta1` with `apps/v1` for `Deployments`

3. (Optional) Migrate volumes

Migrate volumes from your ACS cluster to your AKS cluster. More details can be found in the Migrating Persistent Volumes section.

4. Deploy applications

Use your CI/CD system to deploy applications to AKS or use kubectl to apply the YAML definitions.

5. Validate

Validate that your applications are working as expected and that any migrated data has been copied over.

6. Redirect traffic

Update DNS to point clients to your AKS deployment.

7. Post-migration tasks

If you migrated volumes and chose not to quiesce writes, you'll need to copy that data to the new cluster.

Supported Kubernetes versions in Azure Kubernetes Service (AKS)

9/24/2018 • 2 minutes to read • Edit Online

The Kubernetes community releases minor versions roughly every three months. These releases include new features and improvements. Patch releases are more frequent (sometimes weekly) and are only intended for critical bug fixes in a minor version. These patch releases include fixes for security vulnerabilities or major bugs impacting a large number of customers and products running in production based on Kubernetes.

A new Kubernetes minor version is made available in acs-engine on day one. The AKS Service Level Objective (SLO) targets releasing the minor version for AKS clusters within 30 days, subject to the stability of the release.

Kubernetes version support policy

AKS supports four minor versions of Kubernetes:

- The current minor version that is released upstream (n)
- Three previous minor versions. Each supported minor version also supports two stable patches.

For example, if AKS introduces *1.11.x* today, support is also provided for *1.10.a + 1.10.b*, *1.9.c + 1.9d*, *1.8.e + 1.8f* (where the lettered patch releases are two latest stable builds).

When a new minor version is introduced, the oldest minor version and patch releases supported are retired. 15 days before the release of the new minor version and upcoming version retirement, an announcement is made through the Azure update channels. In the example above where *1.11.x* is released, the retired versions are *1.7.g + 1.7.h*.

When you deploy an AKS cluster in the portal or with the Azure CLI, the cluster is always set to the n-1 minor version and latest patch. For example, if AKS supports *1.11.x*, *1.10.a + 1.10.b*, *1.9.c + 1.9d*, *1.8.e + 1.8f*, the default version for new clusters is *1.10.b*.

FAQ

What happens when a customer upgrades a Kubernetes cluster with a minor version that is not supported?

If you are on the *n-4* version, you are out of the SLO. If your upgrade from version n-4 to n-3 succeeds, then you are back in the SLO. For example:

- If the supported AKS versions are *1.10.a + 1.10.b*, *1.9.c + 1.9d*, *1.8.e + 1.8f* and you are on *1.7.g* or *1.7.h*, you are out of the SLO.
- If the upgrade from *1.7.g* or *1.7.h* to *1.8.e* or *1.8.f* succeeds, you are back in the SLO.

Upgrades to versions older than *n-4* are not supported. In such cases, we recommend customers create new AKS clusters and redeploy their workloads.

What happens when a customer scales a Kubernetes cluster with a minor version that is not supported?

For minor versions not supported by AKS, scaling in or out continues to work without any issues.

Can a customer stay on a Kubernetes version forever?

Yes. However, if the cluster is not on one of the versions supported by AKS, the cluster is out of the AKS SLO.

Azure does not automatically upgrade your cluster or delete it.

What version does the master support if the agent cluster is not in one of the supported AKS versions?

The master is automatically updated to the latest supported version.

Next steps

For information on how to upgrade your cluster, see Upgrade an Azure Kubernetes Service (AKS) cluster.

Scale an Azure Kubernetes Service (AKS) cluster

5/10/2018 • 2 minutes to read • Edit Online

It is easy to scale an AKS cluster to a different number of nodes. Select the desired number of nodes and run the `az aks scale` command. When scaling down, nodes will be carefully cordoned and drained to minimize disruption to running applications. When scaling up, the `az` command waits until nodes are marked `Ready` by the Kubernetes cluster.

Scale the cluster nodes

Use the `az aks scale` command to scale the cluster nodes. The following example scales a cluster named *myAKSCluster* to a single node.

```
az aks scale --name myAKSCluster --resource-group myResourceGroup --node-count 1
```

Output:

```
{
  "id": "/subscriptions/<Subscription
ID>/resourcegroups/myResourceGroup/providers/Microsoft.ContainerService/managedClusters/myAKSCluster",
  "location": "eastus",
  "name": "myAKSCluster",
  "properties": {
    "accessProfiles": {
      "clusterAdmin": {
        "kubeConfig": "..."
      },
      "clusterUser": {
        "kubeConfig": "..."
      }
    },
    "agentPoolProfiles": [
      {
        "count": 1,
        "dnsPrefix": null,
        "fqdn": null,
        "name": "myAKSCluster",
        "osDiskSizeGb": null,
        "osType": "Linux",
        "ports": null,
        "storageProfile": "ManagedDisks",
        "vmSize": "Standard_D2_v2",
        "vnetSubnetId": null
      }
    ],
    "dnsPrefix": "myK8sClust-myResourceGroup-4f48ee",
    "fqdn": "myk8sclust-myresourcegroup-4f48ee-406cc140.hcp.eastus.azmk8s.io",
    "kubernetesVersion": "1.7.7",
    "linuxProfile": {
      "adminUsername": "azureuser",
      "ssh": {
        "publicKeys": [
          {
            "keyData": "..."
          }
        ]
      }
    },
    "provisioningState": "Succeeded",
    "servicePrincipalProfile": {
      "clientId": "e70c1c1c-0ca4-4e0a-be5e-aea5225af017",
      "keyVaultSecretRef": null,
      "secret": null
    }
  },
  "resourceGroup": "myResourceGroup",
  "tags": null,
  "type": "Microsoft.ContainerService/ManagedClusters"
}
```


Next steps

Learn more about deploying and managing AKS with the AKS tutorials.

AKS Tutorial

Upgrade an Azure Kubernetes Service (AKS) cluster

8/17/2018 • 2 minutes to read • Edit Online

Azure Kubernetes Service (AKS) makes it easy to perform common management tasks including upgrading Kubernetes clusters.

Upgrade an AKS cluster

Before upgrading a cluster, use the `az aks get-upgrades` command to check which Kubernetes releases are available for upgrade.

```
az aks get-upgrades --name myAKSCluster --resource-group myResourceGroup --output table
```

Output:

```
Name     ResourceGroup    MasterVersion    NodePoolVersion    Upgrades
-------  ---------------  ---------------  -----------------  -------------------
default  mytestaks007     1.8.10           1.8.10             1.9.1, 1.9.2, 1.9.6
```

We have three versions available for upgrade: 1.9.1, 1.9.2 and 1.9.6. We can use the `az aks upgrade` command to upgrade to the latest available version. During the upgrade process, AKS will add a new node to the cluster, then carefully cordon and drain one node at a time to minimize disruption to running applications.

> **NOTE**
> When upgrading an AKS cluster, Kubernetes minor versions cannot be skipped. For example, upgrades between 1.8.x -> 1.9.x or 1.9.x -> 1.10.x are allowed, however 1.8 -> 1.10 is not. To upgrade, from 1.8 -> 1.10, you need to upgrade first from 1.8 -> 1.9 and then another do another upgrade from 1.9 -> 1.10

```
az aks upgrade --name myAKSCluster --resource-group myResourceGroup --kubernetes-version 1.9.6
```

Output:

```
{
  "id": "/subscriptions/<Subscription
ID>/resourcegroups/myResourceGroup/providers/Microsoft.ContainerService/managedClusters/myAKSCluster",
  "location": "eastus",
  "name": "myAKSCluster",
  "properties": {
    "accessProfiles": {
      "clusterAdmin": {
        "kubeConfig": "..."
      },
      "clusterUser": {
        "kubeConfig": "..."
      }
    },
    "agentPoolProfiles": [
      {
        "count": 1,
        "dnsPrefix": null,
        "fqdn": null,
        "name": "myAKSCluster",
        "osDiskSizeGb": null,
        "osType": "Linux",
        "ports": null,
        "storageProfile": "ManagedDisks",
        "vmSize": "Standard_D2_v2",
        "vnetSubnetId": null
      }
    ],
    "dnsPrefix": "myK8sClust-myResourceGroup-4f48ee",
    "fqdn": "myk8sclust-myresourcegroup-4f48ee-406cc140.hcp.eastus.azmk8s.io",
    "kubernetesVersion": "1.9.6",
    "linuxProfile": {
      "adminUsername": "azureuser",
      "ssh": {
        "publicKeys": [
          {
            "keyData": "..."
          }
        ]
      }
    },
    "provisioningState": "Succeeded",
    "servicePrincipalProfile": {
      "clientId": "e70c1c1c-0ca4-4e0a-be5e-aea5225af017",
      "keyVaultSecretRef": null,
      "secret": null
    }
  },
  "resourceGroup": "myResourceGroup",
  "tags": null,
  "type": "Microsoft.ContainerService/ManagedClusters"
}
```


Confirm that the upgrade was successful with the `az aks show` command.

```
az aks show --name myAKSCluster --resource-group myResourceGroup --output table
```

Output:

```
Name          Location    ResourceGroup    KubernetesVersion    ProvisioningState    Fqdn
------------  ----------  ---------------  -------------------  -------------------  -------------------------
--------------------------------------
myAKSCluster  eastus     myResourceGroup  1.9.6                Succeeded            myk8sclust-
myresourcegroup-3762d8-2f6ca801.hcp.eastus.azmk8s.io
```

Next steps

Learn more about deploying and managing AKS with the AKS tutorials.

AKS Tutorial

Integrate Azure Active Directory with Azure Kubernetes Service

10/2/2018 • 5 minutes to read • Edit Online

Azure Kubernetes Service (AKS) can be configured to use Azure Active Directory (AD) for user authentication. In this configuration, you can log into an AKS cluster using your Azure Active Directory authentication token. Additionally, cluster administrators are able to configure Kubernetes role-based access control (RBAC) based on a users identity or directory group membership.

This article shows you how to deploy the prerequisites for AKS and Azure AD, then how to deploy an Azure AD-enabled cluster and create a simple RBAC role in the AKS cluster.

The following limitations apply:

- Existing non-RBAC enabled AKS clusters cannot currently be updated for RBAC use.
- *Guest* users in Azure AD, such as if you are using a federated login from a different directory, are not supported.

Authentication details

Azure AD authentication is provided to AKS clusters with OpenID Connect. OpenID Connect is an identity layer built on top of the OAuth 2.0 protocol. For more information on OpenID Connect, see the Open ID connect documentation.

From inside of the Kubernetes cluster, Webhook Token Authentication is used to verify authentication tokens. Webhook token authentication is configured and managed as part of the AKS cluster. For more information on Webhook token authentication, see the webhook authentication documentation.

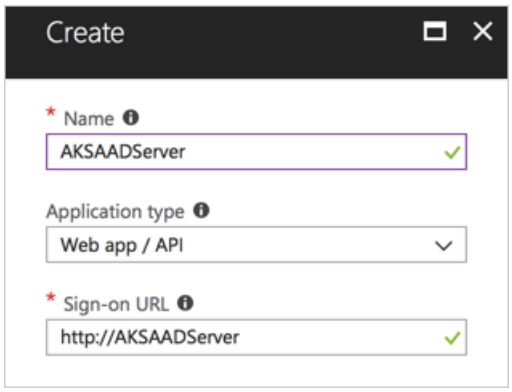
> **NOTE**
> When configuring Azure AD for AKS authentication, two Azure AD application are configured. This operation must be completed by an Azure tenant administrator.

Create server application

The first Azure AD application is used to get a users Azure AD group membership.

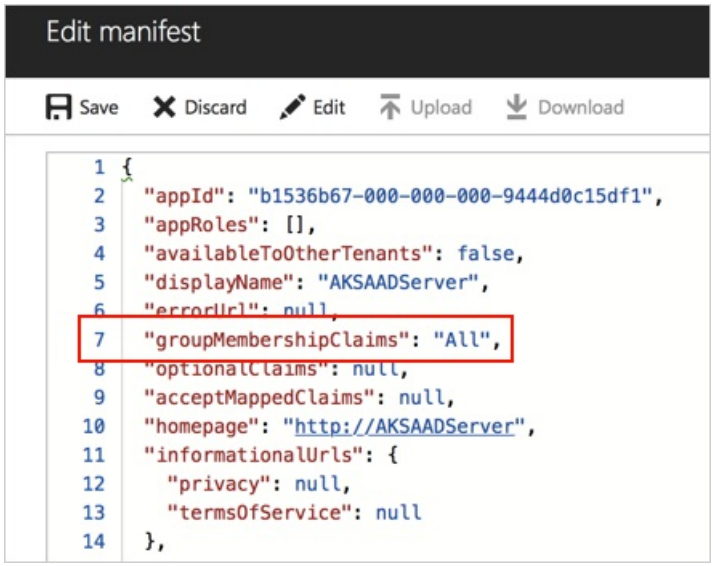
1. Select **Azure Active Directory** > **App registrations** > **New application registration**.

   Give the application a name, select **Web app / API** for the application type, and enter any URI formatted value for **Sign-on URL**. Select **Create** when done.



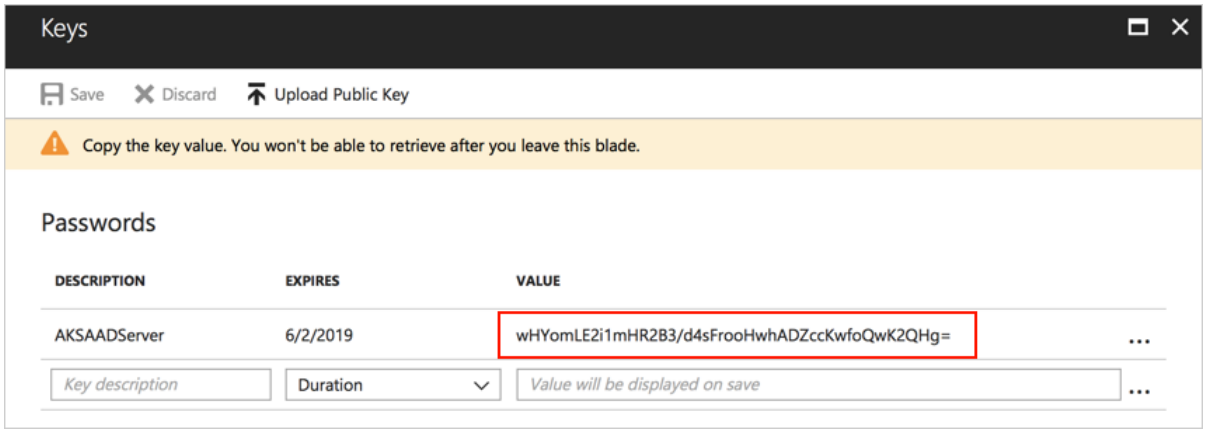
2. Select **Manifest** and edit the `groupMembershipClaims` value to `"All"`.

   Save the updates once complete.

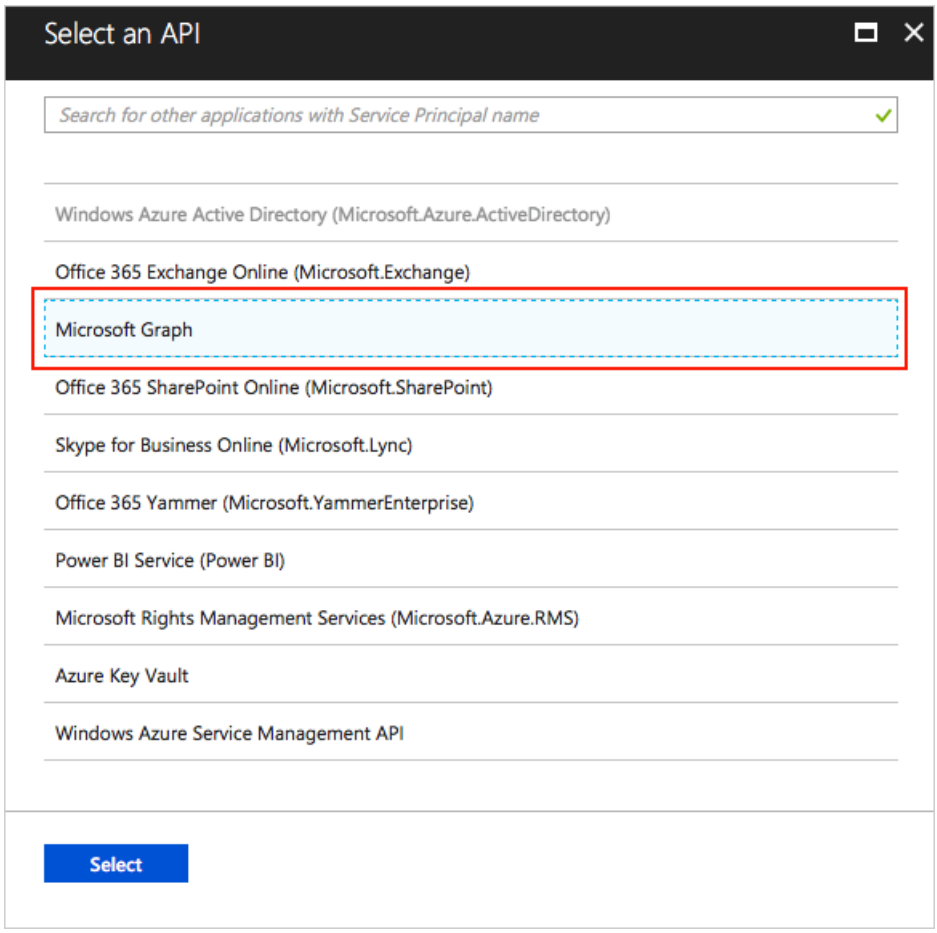


3. Back on the Azure AD application, select **Settings** > **Keys**.

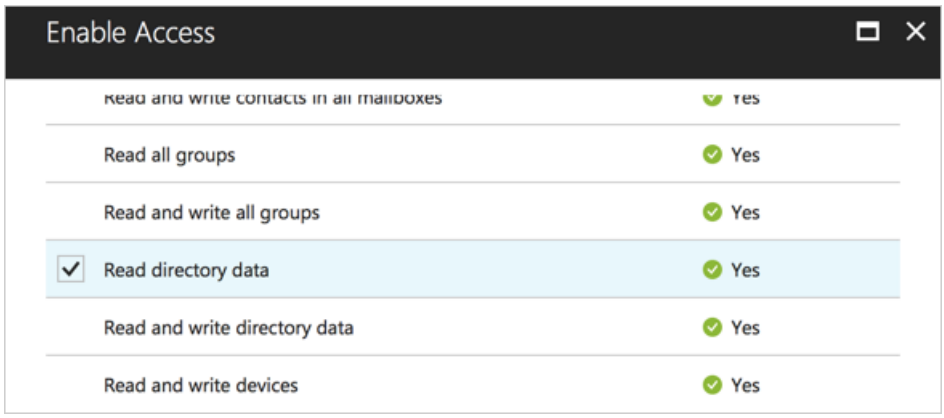
   Add a key description, select an expiration deadline, and select **Save**. Take note of the key value. When deploying an Azure AD enabled AKS cluster, this value is referred to as the `Server application secret`.



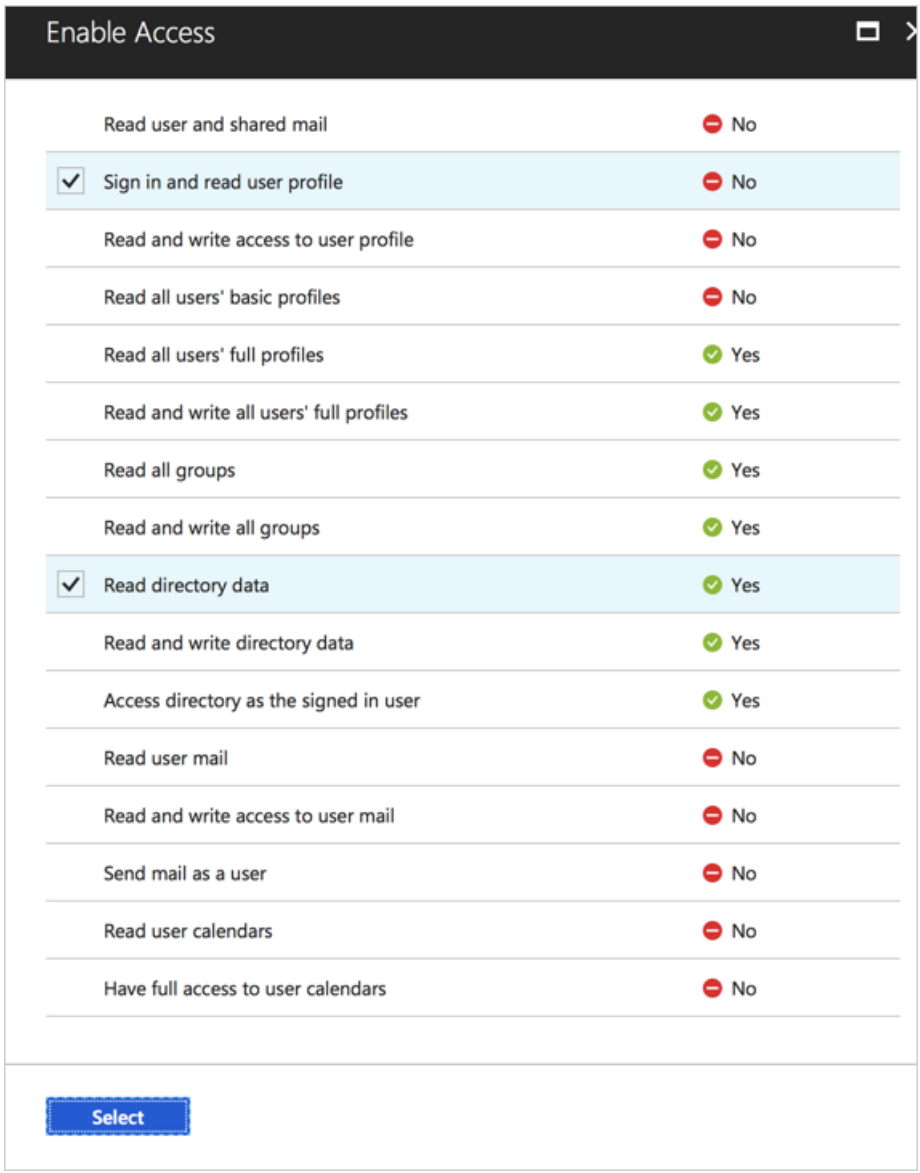
4. Return to the Azure AD application, select **Settings** > **Required permissions** > **Add** > **Select an API** > **Microsoft Graph** > **Select**.



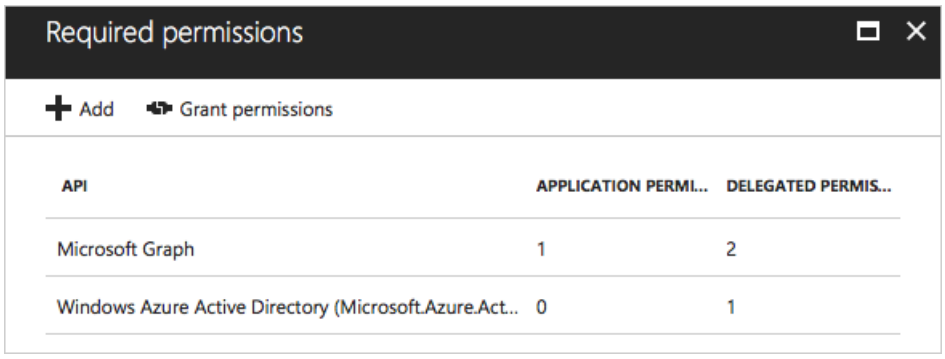
5. Under **APPLICATION PERMISSIONS** place a check next to **Read directory data**.



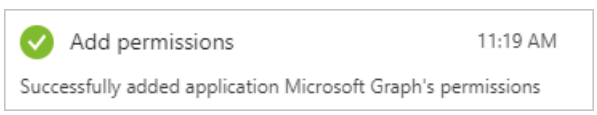
6. Under **DELEGATED PERMISSIONS**, place a check next to **Sign in and read user profile** and **Read directory data**. Save the updates once done.



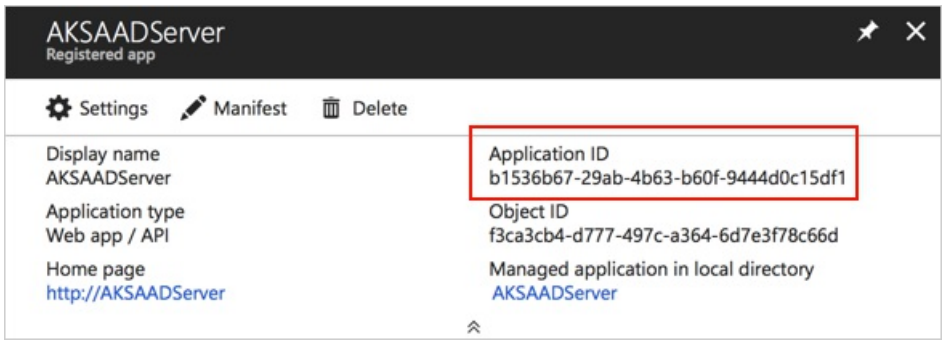
7. Select **Done**, choose *Microsoft Graph* from the list of APIs, then select **Grant Permissions**. This step will fail if the current account is not a tenant admin.



When the permissions have been successfully granted, the following notification is displayed in the portal:



8. Return to the application and take note of the **Application ID**. When deploying an Azure AD-enabled AKS cluster, this value is referred to as the `Server application ID`.

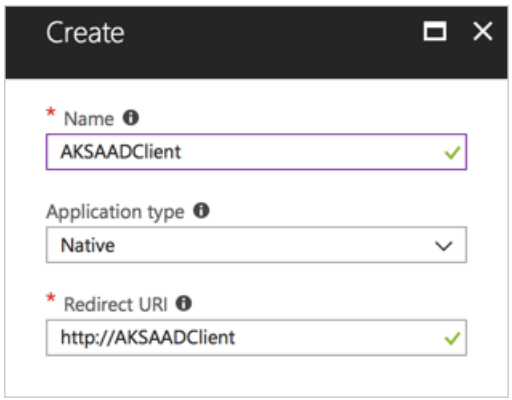


Create client application

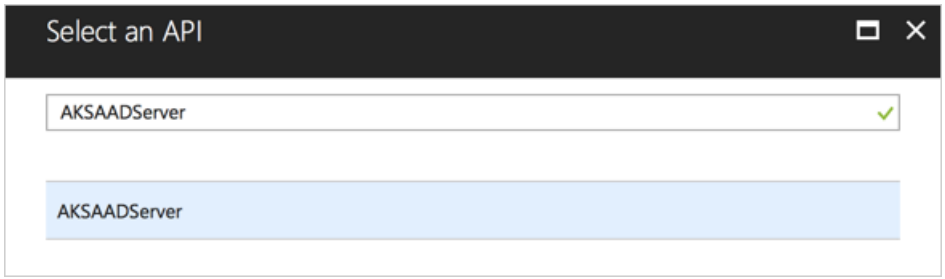
The second Azure AD application is used when logging in with the Kubernetes CLI (kubectl.)

1. Select **Azure Active Directory** > **App registrations** > **New application registration**.

   Give the application a name, select **Native** for the application type, and enter any URI formatted value for **Redirect URI**. Select **Create** when done.

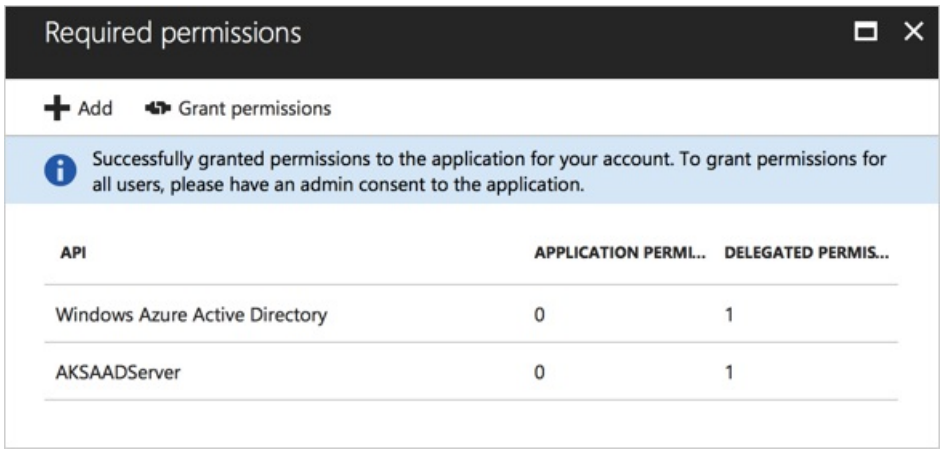


2. From the Azure AD application, select **Settings** > **Required permissions** > **Add** > **Select an API** and search for the name of the server application created in the last step of this document.

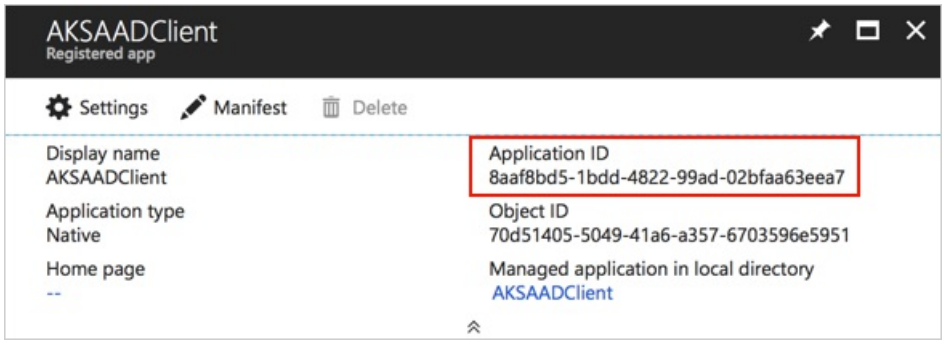


3. Place a check mark next to the application and click **Select**.

4. Select **Done** and **Grant Permissions** to complete this step.



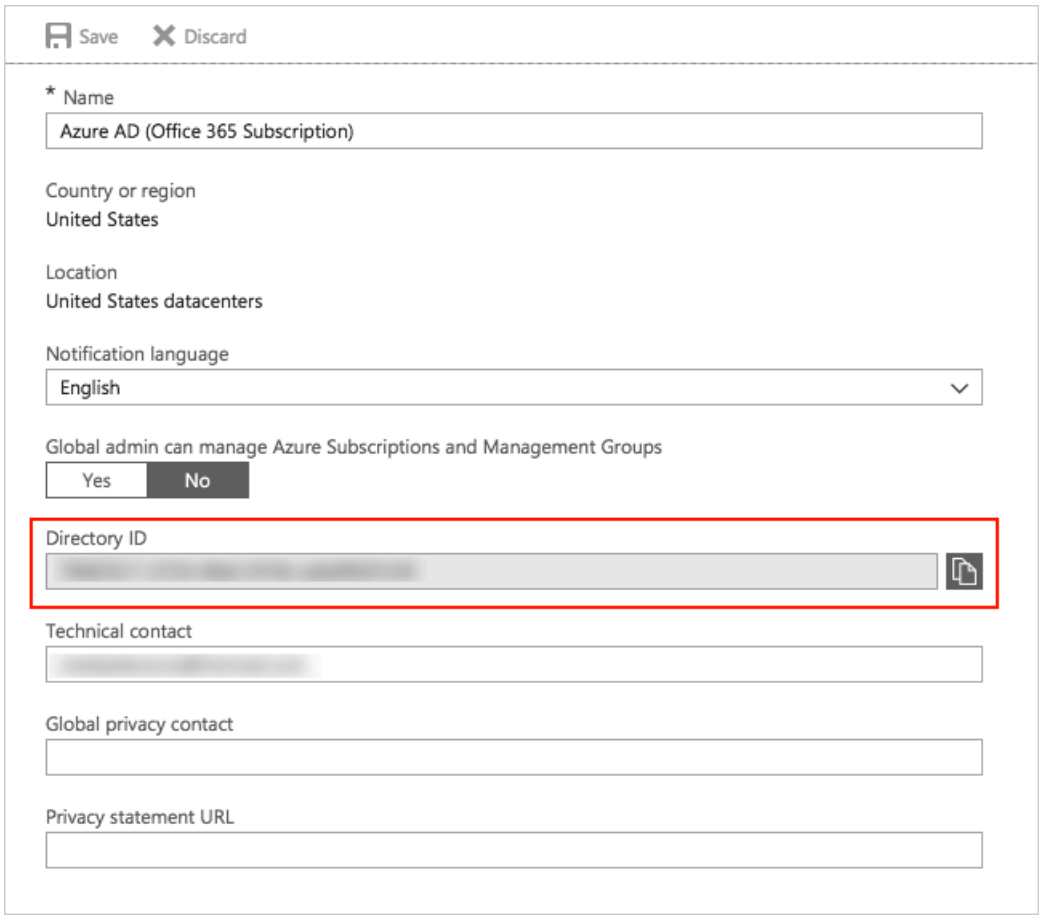
5. Back on the AD application, take note of the **Application ID**. When deploying an Azure AD-enabled AKS cluster, this value is referred to as the `Client application ID`.



Get tenant ID

Finally, get the ID of your Azure tenant. This value is also used when deploying the AKS cluster.

From the Azure portal, select **Azure Active Directory** > **Properties** and take note of the **Directory ID**. When deploying an Azure AD-enabled AKS cluster, this value is referred to as the `Tenant ID`.



Deploy Cluster

Use the az group create command to create a resource group for the AKS cluster.

```
az group create --name myResourceGroup --location eastus
```

Deploy the cluster using the az aks create command. Replace the values in the sample command below with the values collected when creating the Azure AD applications.

```
az aks create \
  --resource-group myResourceGroup \
  --name myAKSCluster \
  --generate-ssh-keys \
  --aad-server-app-id b1536b67-29ab-4b63-b60f-9444d0c15df1 \
  --aad-server-app-secret wHYomLe2i1mHR2B3/d4sFrooHwADZccKwfoQwK2QHg= \
  --aad-client-app-id 8aaf8bd5-1bdd-4822-99ad-02bfaa63eea7 \
  --aad-tenant-id 72f988bf-0000-0000-0000-2d7cd011db47
```

Create RBAC binding

Before an Azure Active Directory account can be used with the AKS cluster, a role binding or cluster role binding needs to be created. *Roles* define the permissions to grant, and *bindings* apply them to desired users. These assignments can be applied to a given namespace, or across the entire cluster. For more information, see Using RBAC authorization.

First, use the az aks get-credentials command with the `--admin` argument to log in to the cluster with admin access.

```
az aks get-credentials --resource-group myResourceGroup --name myAKSCluster --admin
```

Next, use the following manifest to create a ClusterRoleBinding for an Azure AD account. Update the user name with one from your Azure AD tenant. This example gives the account full access to all namespaces of the cluster:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: contoso-cluster-admins
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: User
  name: "user@contoso.com"
```

A role binding can also be created for all members of an Azure AD group. Azure AD groups are specified using the group object ID, as shown in the following example:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
 name: contoso-cluster-admins
roleRef:
 apiGroup: rbac.authorization.k8s.io
 kind: ClusterRole
 name: cluster-admin
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: "894656e1-39f8-4bfe-b16a-510f61af6f41"
```

For more information on securing a Kubernetes cluster with RBAC, see Using RBAC Authorization.

Access cluster with Azure AD

Next, pull the context for the non-admin user using the az aks get-credentials command.

```
az aks get-credentials --resource-group myResourceGroup --name myAKSCluster
```

After running any kubectl command, you will be prompted to authenticate with Azure. Follow the on-screen instructions.

```
$ kubectl get nodes

To sign in, use a web browser to open the page https://microsoft.com/devicelogin and enter the code BUJHWDGNL
to authenticate.

NAME                       STATUS    ROLES     AGE       VERSION
aks-nodepool1-79590246-0   Ready     agent     1h        v1.9.9
aks-nodepool1-79590246-1   Ready     agent     1h        v1.9.9
aks-nodepool1-79590246-2   Ready     agent     1h        v1.9.9
```

Once complete, the authentication token is cached. You are only reprompted to log in when the token has expired or the Kubernetes config file re-created.

If you are seeing an authorization error message after signing in successfully, check that the user you are signing in as is not a Guest in the Azure AD (this is often the case if you are using a federated login from a different directory).

```
error: You must be logged in to the server (Unauthorized)
```

Next Steps

Learn more about securing Kubernetes clusters with RBAC with the Using RBAC Authorization documentation.

Use Virtual Kubelet with Azure Kubernetes Service (AKS)

9/26/2018 • 4 minutes to read • Edit Online

Azure Container Instances (ACI) provide a hosted environment for running containers in Azure. When using ACI, there is no need to manage the underlying compute infrastructure, Azure handles this management for you. When running containers in ACI, you are charged by the second for each running container.

When using the Virtual Kubelet provider for Azure Container Instances, both Linux and Windows containers can be scheduled on a container instance as if it is a standard Kubernetes node. This configuration allows you to take advantage of both the capabilities of Kubernetes and the management value and cost benefit of container instances.

> **NOTE**
> Virtual Kubelet is an experimental open source project and should be used as such. To contribute, file issues, and read more about virtual kubelet, see the Virtual Kubelet GitHub project.

This document details configuring the Virtual Kubelet for container instances on an AKS.

Prerequisite

This document assumes that you have an AKS cluster. If you need an AKS cluster, see theAzure Kubernetes Service (AKS) quickstart.

You also need the Azure CLI version **2.0.33** or later. Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI.

To install the Virtual Kubelet, Helm is also required.

For RBAC-enabled clusters

If your AKS cluster is RBAC-enabled, you must create a service account and role binding for use with Tiller. For more information, see Helm Role-based access control. To create a service account and role binding, create a file named *rbac-virtual-kubelet.yaml* and paste the following definition:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: tiller
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: tiller
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
  - kind: ServiceAccount
    name: tiller
    namespace: kube-system
```

Apply the service account and binding with kubectl apply and specify your *rbac-virtual-kubelet.yaml* file, as shown in the following example:

```
$ kubectl apply -f rbac-virtual-kubelet.yaml

clusterrolebinding.rbac.authorization.k8s.io/tiller created
```

Configure Helm to use the tiller service account:

```
helm init --service-account tiller
```

You can now continue to installing the Virtual Kubelet into your AKS cluster.

Installation

Use the az aks install-connector command to install Virtual Kubelet. The following example deploys both the Linux and Windows connector.

```
az aks install-connector --resource-group myAKSCluster --name myAKSCluster --connector-name virtual-kubelet --
os-type Both
```

These arguments are available for the `aks install-connector` command.

ARGUMENT:	DESCRIPTION	REQUIRED
`--connector-name`	Name of the ACI Connector.	Yes
`--name` `-n`	Name of the managed cluster.	Yes
`--resource-group` `-g`	Name of resource group.	Yes
`--os-type`	Container instances operating system type. Allowed values: Both, Linux, Windows. Default: Linux.	No

ARGUMENT:	DESCRIPTION	REQUIRED
`--aci-resource-group`	The resource group in which to create the ACI container groups.	No
`--location` `-l`	The location to create the ACI container groups.	No
`--service-principal`	Service principal used for authentication to Azure APIs.	No
`--client-secret`	Secret associated with the service principal.	No
`--chart-url`	URL of a Helm chart that installs ACI Connector.	No
`--image-tag`	The image tag of the virtual kubelet container image.	No

Validate Virtual Kubelet

To validate that Virtual Kubelet has been installed, return a list of Kubernetes nodes using the kubectl get nodes command.

```
$ kubectl get nodes

NAME                                   STATUS    ROLES     AGE       VERSION
aks-nodepool1-23443254-0               Ready     agent     16d       v1.9.6
aks-nodepool1-23443254-1               Ready     agent     16d       v1.9.6
aks-nodepool1-23443254-2               Ready     agent     16d       v1.9.6
virtual-kubelet-virtual-kubelet-linux  Ready     agent     4m        v1.8.3
virtual-kubelet-virtual-kubelet-win    Ready     agent     4m        v1.8.3
```

Run Linux container

Create a file named `virtual-kubelet-linux.yaml` and copy in the following YAML. Replace the `kubernetes.io/hostname` value with the name of the Linux Virtual Kubelet node. Take note that a nodeSelector and toleration are being used to schedule the container on the node.

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: aci-helloworld
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: aci-helloworld
    spec:
      containers:
      - name: aci-helloworld
        image: microsoft/aci-helloworld
        ports:
        - containerPort: 80
      nodeSelector:
        kubernetes.io/hostname: virtual-kubelet-virtual-kubelet-linux
      tolerations:
      - key: virtual-kubelet.io/provider
        operator: Equal
        value: azure
        effect: NoSchedule
```

Run the application with the kubectl create command.

```
kubectl create -f virtual-kubelet-linux.yaml
```

Use the kubectl get pods command with the `-o wide` argument to output a list of pods with the scheduled node. Notice that the `aci-helloworld` pod has been scheduled on the `virtual-kubelet-virtual-kubelet-linux` node.

```
$ kubectl get pods -o wide

NAME                                READY     STATUS    RESTARTS   AGE       IP             NODE
aci-helloworld-2559879000-8vmjw     1/1       Running   0          39s       52.179.3.180   virtual-kubelet-
virtual-kubelet-linux
```

Run Windows container

Create a file named `virtual-kubelet-windows.yaml` and copy in the following YAML. Replace the `kubernetes.io/hostname` value with the name of the Windows Virtual Kubelet node. Take note that a nodeSelector and toleration are being used to schedule the container on the node.

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: nanoserver-iis
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: nanoserver-iis
    spec:
      containers:
      - name: nanoserver-iis
        image: microsoft/iis:nanoserver
        ports:
        - containerPort: 80
      nodeSelector:
        kubernetes.io/hostname: virtual-kubelet-virtual-kubelet-win
      tolerations:
      - key: virtual-kubelet.io/provider
        operator: Equal
        value: azure
        effect: NoSchedule
```

Run the application with the kubectl create command.

```
kubectl create -f virtual-kubelet-windows.yaml
```

Use the kubectl get pods command with the `-o wide` argument to output a list of pods with the scheduled node. Notice that the `nanoserver-iis` pod has been scheduled on the `virtual-kubelet-virtual-kubelet-win` node.

```
$ kubectl get pods -o wide

NAME                                READY     STATUS    RESTARTS   AGE       IP             NODE
nanoserver-iis-868bc8d489-tq4st     1/1       Running   8          21m       138.91.121.91   virtual-kubelet-
virtual-kubelet-win
```

Remove Virtual Kubelet

Use the az aks remove-connector command to remove Virtual Kubelet. Replace the argument values with the name of the connector, AKS cluster, and the AKS cluster resource group.

```
az aks remove-connector --resource-group myAKSCluster --name myAKSCluster --connector-name virtual-kubelet
```

NOTE

If you encounter errors removing both OS connectors, or want to remove just the Windows or Linux OS connector, you can manually specify the OS type. Add the `--os-type` parameter to the previous `az aks remove-connector` command, and specify `Windows` or `Linux`.

Next steps

For possible issues with the Virtual Kubelet, see the Known quirks and workarounds. To report problems with the Virtual Kubelet, open a GitHub issue.

Read more about Virtual Kubelet at the Virtual Kubelet Github project.

Cluster Autoscaler on Azure Kubernetes Service (AKS) - Preview

10/8/2018 • 7 minutes to read • Edit Online

Azure Kubernetes Service (AKS) provides a flexible solution to deploy a managed Kubernetes cluster in Azure. As resource demands increase, the cluster autoscaler allows your cluster to grow to meet that demand based on constraints you set. The cluster autoscaler (CA) does this by scaling your agent nodes based on pending pods. It scans the cluster periodically to check for pending pods or empty nodes and increases the size if possible. By default, the CA scans for pending pods every 10 seconds and removes a node if it's unneeded for more than 10 minutes. When used with the horizontal pod autoscaler (HPA), the HPA will update pod replicas and resources as per demand. If there aren't enough nodes or unneeded nodes following this pod scaling, the CA will respond and schedule the pods on the new set of nodes.

This article describes how to deploy the cluster autoscaler on the agent nodes. However, since the cluster autoscaler is deployed in the kube-system namespace, the autoscaler will not scale down the node running that pod.

> **IMPORTANT**
> Azure Kubernetes Service (AKS) cluster autoscaler integration is currently in **preview**. Previews are made available to you on the condition that you agree to the supplemental terms of use. Some aspects of this feature may change prior to general availability (GA).

Prerequisites

This document assumes that you have an RBAC-enabled AKS cluster. If you need an AKS cluster, see theAzure Kubernetes Service (AKS) quickstart.

To use the cluster autoscaler, your cluster must be using Kubernetes v1.10.X or higher and must be RBAC-enabled. To upgrade your cluster, see the article on upgrading an AKS cluster.

Gather information

To generate the permissions for your cluster autoscaler to run in your cluster, run this bash script:

```
#! /bin/bash
ID=`az account show --query id -o json`
SUBSCRIPTION_ID=`echo $ID | tr -d '"' `

TENANT=`az account show --query tenantId -o json`
TENANT_ID=`echo $TENANT | tr -d '"' | base64`

read -p "What's your cluster name? " cluster_name
read -p "Resource group name? " resource_group

CLUSTER_NAME=`echo $cluster_name | base64`
RESOURCE_GROUP=`echo $resource_group | base64`

PERMISSIONS=`az ad sp create-for-rbac --role="Contributor" --scopes="/subscriptions/$SUBSCRIPTION_ID" -o json`
CLIENT_ID=`echo $PERMISSIONS | sed -e 's/^.*"appId"[ ]*:[ ]*"//' -e 's/".*//' | base64`
CLIENT_SECRET=`echo $PERMISSIONS | sed -e 's/^.*"password"[ ]*:[ ]*"//' -e 's/".*//' | base64`

SUBSCRIPTION_ID=`echo $ID | tr -d '"' | base64 `

NODE_RESOURCE_GROUP=`az aks show --name $cluster_name  --resource-group $resource_group -o tsv --query
'nodeResourceGroup' | base64`

echo "---
apiVersion: v1
kind: Secret
metadata:
    name: cluster-autoscaler-azure
    namespace: kube-system
data:
    ClientID: $CLIENT_ID
    ClientSecret: $CLIENT_SECRET
    ResourceGroup: $RESOURCE_GROUP
    SubscriptionID: $SUBSCRIPTION_ID
    TenantID: $TENANT_ID
    VMType: QUtTCg==
    ClusterName: $CLUSTER_NAME
    NodeResourceGroup: $NODE_RESOURCE_GROUP
---"
```

After following the steps in the script, the script will output your details in the form of a secret, like so:

```
---
apiVersion: v1
kind: Secret
metadata:
  name: cluster-autoscaler-azure
  namespace: kube-system
data:
  ClientID: <base64-encoded-client-id>
  ClientSecret: <base64-encoded-client-secret>$
  ResourceGroup: <base64-encoded-resource-group>  SubscriptionID: <base64-encode-subscription-id>
  TenantID: <base64-encoded-tenant-id>
  VMType: QUtTCg==
  ClusterName: <base64-encoded-clustername>
  NodeResourceGroup: <base64-encoded-node-resource-group>
---
```

Next, get the name of your node pool by running the following command.

```
$ kubectl get nodes --show-labels
```

Output:

```
NAME                       STATUS    ROLES     AGE       VERSION   LABELS
aks-nodepool1-37756013-0   Ready     agent     1h        v1.10.3
agentpool=nodepool1,beta.kubernetes.io/arch=amd64,beta.kubernetes.io/instance-
type=Standard_DS1_v2,beta.kubernetes.io/os=linux,failure-domain.beta.kubernetes.io/region=eastus,failure-
domain.beta.kubernetes.io/zone=0,kubernetes.azure.com/cluster=MC_[resource-group]\_[cluster-
name]_[location],kubernetes.io/hostname=aks-nodepool1-37756013-
0,kubernetes.io/role=agent,storageprofile=managed,storagetier=Premium_LRS
```

Then, extract the value of the label **agentpool**. The default name for the node pool of a cluster is "nodepool1".

Now using your secret and node pool, you can create a deployment chart.

Create a deployment chart

Create a file named `aks-cluster-autoscaler.yaml`, and copy into it the following YAML code.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  labels:
    k8s-addon: cluster-autoscaler.addons.k8s.io
    k8s-app: cluster-autoscaler
  name: cluster-autoscaler
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRole
metadata:
  name: cluster-autoscaler
  labels:
    k8s-addon: cluster-autoscaler.addons.k8s.io
    k8s-app: cluster-autoscaler
rules:
- apiGroups: [""]
  resources: ["events","endpoints"]
  verbs: ["create", "patch"]
- apiGroups: [""]
  resources: ["pods/eviction"]
  verbs: ["create"]
- apiGroups: [""]
  resources: ["pods/status"]
  verbs: ["update"]
- apiGroups: [""]
  resources: ["endpoints"]
  resourceNames: ["cluster-autoscaler"]
  verbs: ["get","update"]
- apiGroups: [""]
  resources: ["nodes"]
  verbs: ["watch","list","get","update"]
- apiGroups: [""]
  resources: ["pods","services","replicationcontrollers","persistentvolumeclaims","persistentvolumes"]
  verbs: ["watch","list","get"]
- apiGroups: ["extensions"]
  resources: ["replicasets","daemonsets"]
  verbs: ["watch","list","get"]
- apiGroups: ["policy"]
  resources: ["poddisruptionbudgets"]
  verbs: ["watch","list"]
- apiGroups: ["apps"]
  resources: ["statefulsets"]
  verbs: ["watch","list","get"]
- apiGroups: ["storage.k8s.io"]
  resources: ["storageclasses"]
  verbs: ["get", "list", "watch"]
```

```
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: Role
metadata:
  name: cluster-autoscaler
  namespace: kube-system
  labels:
    k8s-addon: cluster-autoscaler.addons.k8s.io
    k8s-app: cluster-autoscaler
rules:
- apiGroups: [""]
  resources: ["configmaps"]
  verbs: ["create"]
- apiGroups: [""]
  resources: ["configmaps"]
  resourceNames: ["cluster-autoscaler-status"]
  verbs: ["delete","get","update"]

---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: cluster-autoscaler
  labels:
    k8s-addon: cluster-autoscaler.addons.k8s.io
    k8s-app: cluster-autoscaler
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-autoscaler
subjects:
  - kind: ServiceAccount
    name: cluster-autoscaler
    namespace: kube-system

---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: RoleBinding
metadata:
  name: cluster-autoscaler
  namespace: kube-system
  labels:
    k8s-addon: cluster-autoscaler.addons.k8s.io
    k8s-app: cluster-autoscaler
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: cluster-autoscaler
subjects:
  - kind: ServiceAccount
    name: cluster-autoscaler
    namespace: kube-system

---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    app: cluster-autoscaler
  name: cluster-autoscaler
  namespace: kube-system
spec:
  replicas: 1
  selector:
    matchLabels:
      app: cluster-autoscaler
  template:
    metadata:
      labels:
```

```
      app: cluster-autoscaler
  spec:
    serviceAccountName: cluster-autoscaler
    containers:
    - image: gcr.io/google-containers/cluster-autoscaler:v1.2.2
      imagePullPolicy: Always
      name: cluster-autoscaler
      resources:
        limits:
          cpu: 100m
          memory: 300Mi
        requests:
          cpu: 100m
          memory: 300Mi
      command:
      - ./cluster-autoscaler
      - --v=3
      - --logtostderr=true
      - --cloud-provider=azure
      - --skip-nodes-with-local-storage=false
      - --nodes=1:10:nodepool1
      env:
      - name: ARM_SUBSCRIPTION_ID
        valueFrom:
          secretKeyRef:
            key: SubscriptionID
            name: cluster-autoscaler-azure
      - name: ARM_RESOURCE_GROUP
        valueFrom:
          secretKeyRef:
            key: ResourceGroup
            name: cluster-autoscaler-azure
      - name: ARM_TENANT_ID
        valueFrom:
          secretKeyRef:
            key: TenantID
            name: cluster-autoscaler-azure
      - name: ARM_CLIENT_ID
        valueFrom:
          secretKeyRef:
            key: ClientID
            name: cluster-autoscaler-azure
      - name: ARM_CLIENT_SECRET
        valueFrom:
          secretKeyRef:
            key: ClientSecret
            name: cluster-autoscaler-azure
      - name: ARM_VM_TYPE
        valueFrom:
          secretKeyRef:
            key: VMType
            name: cluster-autoscaler-azure
      - name: AZURE_CLUSTER_NAME
        valueFrom:
          secretKeyRef:
            key: ClusterName
            name: cluster-autoscaler-azure
      - name: AZURE_NODE_RESOURCE_GROUP
        valueFrom:
          secretKeyRef:
            key: NodeResourceGroup
            name: cluster-autoscaler-azure
    restartPolicy: Always
```

Copy and paste the secret created in the previous step, and insert it at the start of the file.

Next, to set the range of nodes, fill in the argument for `--nodes` under `command` in the form

MIN:MAX:NODE_POOL_NAME. For example: `--nodes=3:10:nodepool1` sets the minimum number of nodes to 3, the maximum number of nodes to 10, and the node pool name to nodepool1.

Then, fill in the image field under **containers** with the version of the cluster autoscaler you want to use. AKS requires v1.2.2 or newer. The example here uses v1.2.2.

Deployment

Deploy cluster-autoscaler by running

```
kubectl create -f aks-cluster-autoscaler.yaml
```

To check if the cluster autoscaler is running, use the following command and check the list of pods. There should be a pod prefixed with "cluster-autoscaler" running. If you see this, your cluster autoscaler has been deployed.

```
kubectl -n kube-system get pods
```

To view the status of the cluster autoscaler, run

```
kubectl -n kube-system describe configmap cluster-autoscaler-status
```

Interpreting the cluster autoscaler status

```
$ kubectl -n kube-system describe configmap cluster-autoscaler-status
Name:         cluster-autoscaler-status
Namespace:    kube-system
Labels:       <none>
Annotations:  cluster-autoscaler.kubernetes.io/last-updated=2018-07-25 22:59:22.661669494 +0000 UTC

Data
====
status:
----
Cluster-autoscaler status at 2018-07-25 22:59:22.661669494 +0000 UTC:
Cluster-wide:
  Health:      Healthy (ready=1 unready=0 notStarted=0 longNotStarted=0 registered=1 longUnregistered=0)
               LastProbeTime:      2018-07-25 22:59:22.067828801 +0000 UTC
               LastTransitionTime: 2018-07-25 00:38:36.41372897 +0000 UTC
  ScaleUp:     NoActivity (ready=1 registered=1)
               LastProbeTime:      2018-07-25 22:59:22.067828801 +0000 UTC
               LastTransitionTime: 2018-07-25 00:38:36.41372897 +0000 UTC
  ScaleDown:   NoCandidates (candidates=0)
               LastProbeTime:      2018-07-25 22:59:22.067828801 +0000 UTC
               LastTransitionTime: 2018-07-25 00:38:36.41372897 +0000 UTC

NodeGroups:
  Name:        nodepool1
  Health:      Healthy (ready=1 unready=0 notStarted=0 longNotStarted=0 registered=1 longUnregistered=0
cloudProviderTarget=1 (minSize=1, maxSize=5))
               LastProbeTime:      2018-07-25 22:59:22.067828801 +0000 UTC
               LastTransitionTime: 2018-07-25 00:38:36.41372897 +0000 UTC
  ScaleUp:     NoActivity (ready=1 cloudProviderTarget=1)
               LastProbeTime:      2018-07-25 22:59:22.067828801 +0000 UTC
               LastTransitionTime: 2018-07-25 00:38:36.41372897 +0000 UTC
  ScaleDown:   NoCandidates (candidates=0)
               LastProbeTime:      2018-07-25 22:59:22.067828801 +0000 UTC
               LastTransitionTime: 2018-07-25 00:38:36.41372897 +0000 UTC


Events:  <none>
```

The cluster autoscaler status allows you to see the state of the cluster autoscaler on two different levels: cluster-wide and within each node group. Since AKS currently only supports one node pool, these metrics are the same.

- Health indicates the overall health of the nodes. If the cluster autoscaler struggles to create or removes nodes in the cluster, this status will change to "Unhealthy". There's also a breakdown of the status of different nodes:
    - "Ready" means a node is a ready to have pods scheduled on it.
    - "Unready" means a node that broke down after it started.
    - "NotStarted" means a node isn't fully started yet.
    - "LongNotStarted" means a node failed to start within a reasonable limit.
    - "Registered means a node is registered in the group
    - "Unregistered" means a node is present on the cluster provider side but failed to register in Kubernetes.
- ScaleUp allows you to check when the cluster determines a scale up should occur in your cluster.
    - A transition is when the number of nodes in the cluster changes or the status of a node changes.
    - The number of ready nodes is the number of nodes available and ready in the cluster.
    - The cloudProviderTarget is the number of nodes the cluster autoscaler has determined the cluster needs to handle its workload.
- ScaleDown allows you to check if there are candidates for scale down.
    - A candidate for scale down is a node the cluster autoscaler has determined can be removed without

affecting the cluster's ability to handle its workload.
    - The times provided show the last time the cluster was checked for scale down candidates and its last transition time.

Finally, under Events, you can see up any scale or scale down events, failed or successful, and their times, that the cluster autoscaler has carried out.

Next steps

To use the cluster autoscaler with the horizontal pod autoscaler, check out scaling Kubernetes application and infrastructure.

Learn more about deploying and managing AKS with the AKS tutorials.

AKS Tutorial

Create a Kubernetes cluster with Azure Kubernetes Service and Terraform

9/24/2018 • 7 minutes to read • Edit Online

Azure Kubernetes Service (AKS) manages your hosted Kubernetes environment, making it quick and easy to deploy and manage containerized applications without container orchestration expertise. It also eliminates the burden of ongoing operations and maintenance by provisioning, upgrading, and scaling resources on demand, without taking your applications offline.

In this tutorial, you learn how to perform the following tasks in creating a Kubernetes cluster using Terraform and AKS:

- Use HCL (HashiCorp Language) to define a Kubernetes cluster
- Use Terraform and AKS to create a Kubernetes cluster
- Use the kubectl tool to test the availability of a Kubernetes cluster

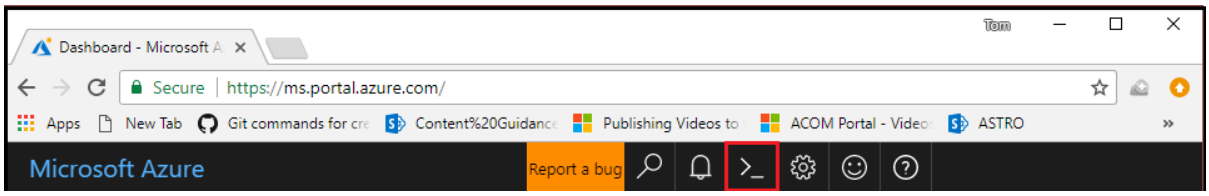
Prerequisites

- **Azure subscription**: If you don't have an Azure subscription, create a free account before you begin.
- **Configure Terraform**: Follow the directions in the article, Terraform and configure access to Azure
- **Azure service principal**: Follow the directions in the section of the **Create the service principal** section in the article, Create an Azure service principal with Azure CLI. Take note of the values for the appId, displayName, password, and tenant.

Create the directory structure

The first step is to create the directory that holds your Terraform configuration files for the exercise.

1. Browse to the Azure portal.
2. Open Azure Cloud Shell. If you didn't select an environment previously, select **Bash** as your environment.



3. Change directories to the `clouddrive` directory.

```
cd clouddrive
```

4. Create a directory named `terraform-aks-k8s`.

```
mkdir terraform-aks-k8s
```

5. Change directories to the new directory:

```
cd terraform-aks-k8s
```

Declare the Azure provider

Create the Terraform configuration file that declares the Azure provider.

1. In Cloud Shell, create a file named `main.tf`.

   ```
   vi main.tf
   ```

2. Enter insert mode by selecting the I key.

3. Paste the following code into the editor:

   ```
   provider "azurerm" {
       version = "~>1.5"
   }

   terraform {
       backend "azurerm" {}
   }
   ```

4. Exit insert mode by selecting the **Esc** key.

5. Save the file and exit the vi editor by entering the following command:

   ```
   :wq
   ```

Define a Kubernetes cluster

Create the Terraform configuration file that declares the resources for the Kubernetes cluster.

1. In Cloud Shell, create a file named `k8s.tf`.

   ```
   vi k8s.tf
   ```

2. Enter insert mode by selecting the I key.

3. Paste the following code into the editor:

```
resource "azurerm_resource_group" "k8s" {
    name     = "${var.resource_group_name}"
    location = "${var.location}"
}

resource "azurerm_kubernetes_cluster" "k8s" {
    name                = "${var.cluster_name}"
    location            = "${azurerm_resource_group.k8s.location}"
    resource_group_name = "${azurerm_resource_group.k8s.name}"
    dns_prefix          = "${var.dns_prefix}"

    linux_profile {
        admin_username = "ubuntu"

        ssh_key {
        key_data = "${file("${var.ssh_public_key}")}"
        }
    }

    agent_pool_profile {
        name            = "default"
        count           = "${var.agent_count}"
        vm_size         = "Standard_DS2_v2"
        os_type         = "Linux"
        os_disk_size_gb = 30
    }

    service_principal {
        client_id     = "${var.client_id}"
        client_secret = "${var.client_secret}"
    }

    tags {
        Environment = "Development"
    }
}
```

The preceding code sets the name of the cluster, location, and the resource_group_name. In addition, the dns_prefix value - that forms part of the fully qualified domain name (FQDN) used to access the cluster - is set.

The **linux_profile** record allows you to configure the settings that enable signing into the worker nodes using SSH.

With AKS, you pay only for the worker nodes. The **agent_pool_profile** record configures the details for these worker nodes. The **agent_pool_profile record** includes the number of worker nodes to create and the type of worker nodes. If you need to scale up or scale down the cluster in the future, you modify the **count** value in this record.

4. Exit insert mode by selecting the **Esc** key.
5. Save the file and exit the vi editor by entering the following command:

```
:wq
```

Declare the variables

1. In Cloud Shell, create a file named `variables.tf`.

```
vi variables.tf
```

2. Enter insert mode by selecting the I key.
3. Paste the following code into the editor:

```
variable "client_id" {}
variable "client_secret" {}

variable "agent_count" {
    default = 3
}

variable "ssh_public_key" {
    default = "~/.ssh/id_rsa.pub"
}

variable "dns_prefix" {
    default = "k8stest"
}

variable cluster_name {
    default = "k8stest"
}

variable resource_group_name {
    default = "azure-k8stest"
}

variable location {
    default = "Central US"
}
```

4. Exit insert mode by selecting the **Esc** key.
5. Save the file and exit the vi editor by entering the following command:

```
:wq
```

Create a Terraform output file

Terraform outputs allow you to define values that will be highlighted to the user when Terraform applies a plan, and can be queried using the `terraform output` command. In this section, you create an output file that allows access to the cluster with kubectl.

1. In Cloud Shell, create a file named `output.tf`.

```
vi output.tf
```

2. Enter insert mode by selecting the I key.
3. Paste the following code into the editor:

```
output "client_key" {
    value = "${azurerm_kubernetes_cluster.k8s.kube_config.0.client_key}"
}

output "client_certificate" {
    value = "${azurerm_kubernetes_cluster.k8s.kube_config.0.client_certificate}"
}

output "cluster_ca_certificate" {
    value = "${azurerm_kubernetes_cluster.k8s.kube_config.0.cluster_ca_certificate}"
}

output "cluster_username" {
    value = "${azurerm_kubernetes_cluster.k8s.kube_config.0.username}"
}

output "cluster_password" {
    value = "${azurerm_kubernetes_cluster.k8s.kube_config.0.password}"
}

output "kube_config" {
    value = "${azurerm_kubernetes_cluster.k8s.kube_config_raw}"
}

output "host" {
    value = "${azurerm_kubernetes_cluster.k8s.kube_config.0.host}"
}
```

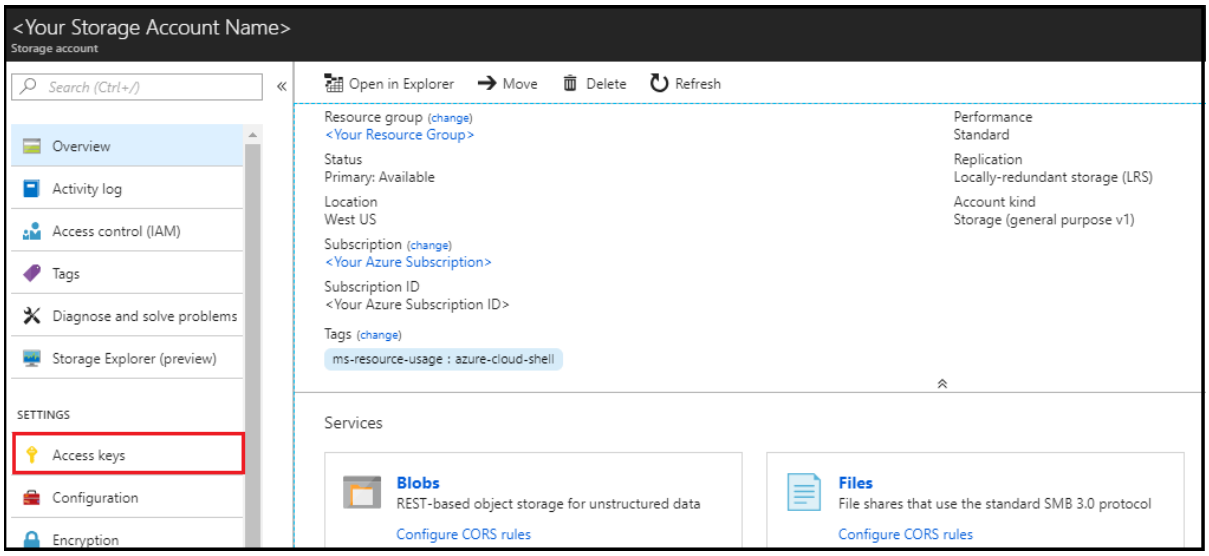
4. Exit insert mode by selecting the **Esc** key.
5. Save the file and exit the vi editor by entering the following command:

```
:wq
```

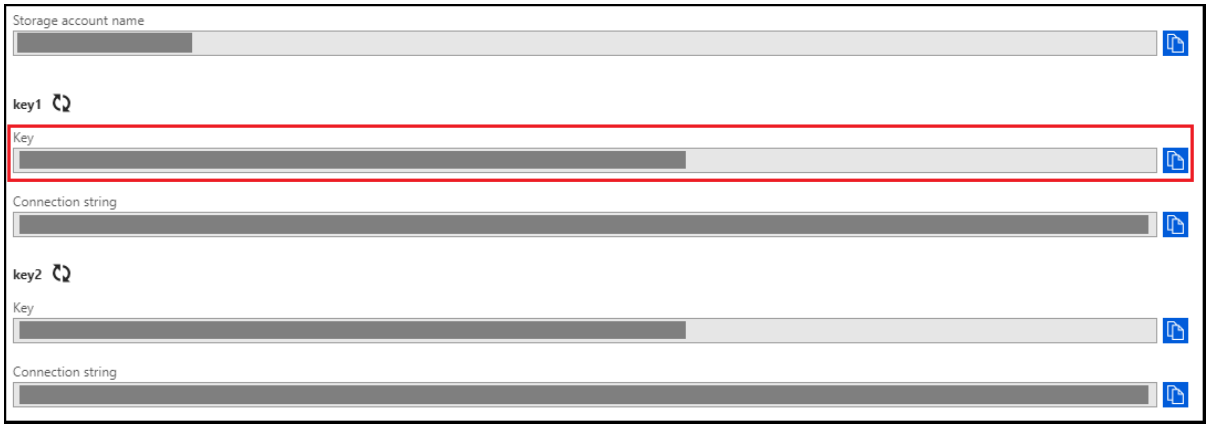
Set up Azure storage to store Terraform state

Terraform tracks state locally via the `terraform.tfstate` file. This pattern works well in a single-person environment. However, in a more practical multi-person environment, you need to track state on the server utilizing Azure storage. In this section, you retrieve the necessary storage account information (account name and account key), and create a storage container into which the Terraform state information will be stored.

1. In the Azure portal, select **All services** in the left menu.
2. Select **Storage accounts**.
3. On the **Storage accounts** tab, select the name of the storage account into which Terraform is to store state. For example, you can use the storage account created when you opened Cloud Shell the first time. The storage account name created by Cloud Shell typically starts with `cs` followed by a random string of numbers and letters. **Remember the name of the storage account you select, as it is needed later.**
4. On the storage account tab, select **Access keys**.



5. Make note of the **key1 key** value. (Selecting the icon to the right of the key copies the value to the clipboard.)



6. In Cloud Shell, create a container in your Azure storage account (replace the <YourAzureStorageAccountName> and <YourAzureStorageAccountAccessKey> placeholders with the appropriate values for your Azure storage account).

```
az storage container create -n tfstate --account-name <YourAzureStorageAccountName> --account-key
<YourAzureStorageAccountKey>
```

Create the Kubernetes cluster

In this section, you see how to use the `terraform init` command to create the resources defined the configuration files you created in the previous sections.

1. In Cloud Shell, initialize Terraform (replace the <YourAzureStorageAccountName> and <YourAzureStorageAccountAccessKey> placeholders with the appropriate values for your Azure storage account).

```
terraform init -backend-config="storage_account_name=<YourAzureStorageAccountName>" -backend-
config="container_name=tfstate" -backend-config="access_key=<YourStorageAccountAccessKey>" -backend-
config="key=codelab.microsoft.tfstate"
```

   The `terraform init` command displays the success of initializing the backend and provider plugin:

```
Initializing the backend...

Initializing provider plugins...

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

2. Export your service principal credentials. Replace the <your-client-id> and <your-client-secret> placeholders with the **appId** and **password** values associated with your service principal, respectively.

```
export TF_VAR_client_id=<your-client-id>
export TF_VAR_client_secret=<your-client-secret>
```

3. Run the `terraform plan` command to create the Terraform plan that defines the infrastructure elements.

```
terraform plan -out out.plan
```

The `terraform plan` command displays the resources that will be created when you run the `terraform apply` command:

```
      id:                                   <computed>
      location:                             "centralus"
      name:                                 "azure-k8stest"
      tags.%:                               <computed>


Plan: 2 to add, 0 to change, 0 to destroy.

------------------------------------------------------------------------

This plan was saved to: out.plan

To perform exactly these actions, run the following command to apply:
    terraform apply "out.plan"
```

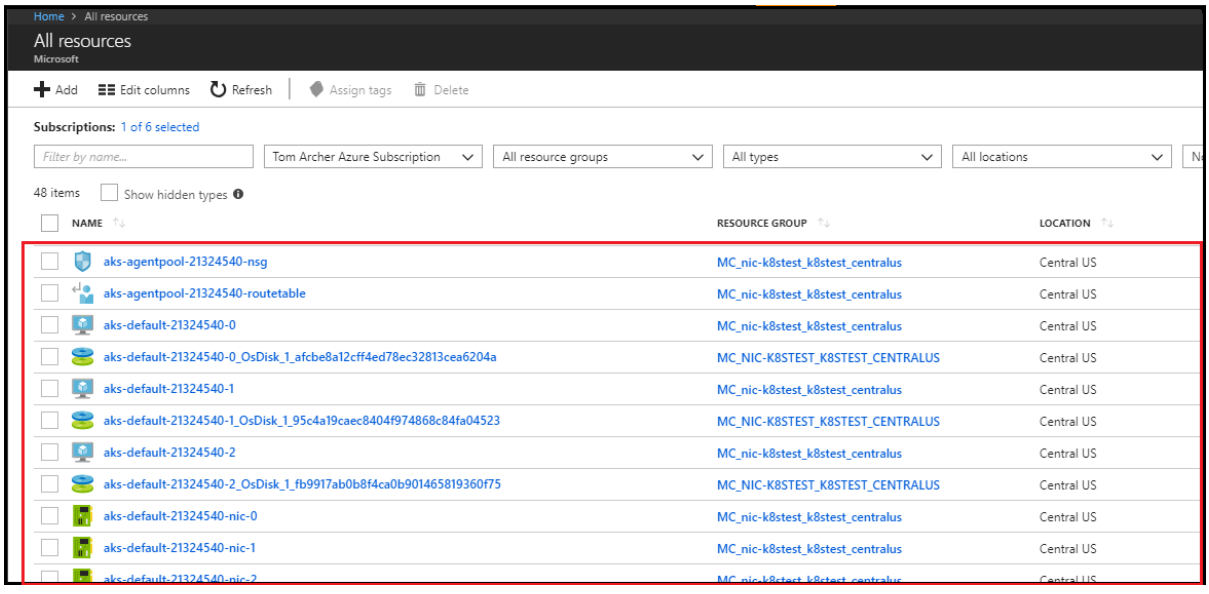
4. Run the `terraform apply` command to apply the plan to create the Kubernetes cluster. The process to create a Kubernetes cluster can take several minutes, resulting in the Cloud Shell session timing out. If the Cloud Shell session times out, you can follow the steps in the section "Recover from a Cloud Shell timeout" to enable you to complete the tutorial.

```
terraform apply out.plan
```

The `terraform apply` command displays the results of creating the resources defined in your configuration files:

```
  tags.%:    "" => "<computed>"
azurerm_resource_group.k8s: Creation complete after 1s (ID: /subscriptions/ad7af7
azurerm_kubernetes_cluster.k8s: Creating...
  agent_pool_profile.#:                       "" => "1"
  agent_pool_profile.0.count:                 "" => "3"
  agent_pool_profile.0.dns_prefix:            "" => "<computed>"
  agent_pool_profile.0.fqdn:                  "" => "<computed>"
  agent_pool_profile.0.name:                  "" => "default"
  agent_pool_profile.0.os_disk_size_gb:       "" => "30"
  agent_pool_profile.0.os_type:               "" => "Linux"
  agent_pool_profile.0.vm_size:               "" => "Standard_D2"
  dns_prefix:                                 "" => "k8stest"
  fqdn:                                       "" => "<computed>"
  kube_config.#:                              "" => "<computed>"
  kube_config_raw:                            "<sensitive>" => "<sensitive>"
  kubernetes_version:                         "" => "<computed>"
  linux_profile.#:                            "" => "1"
  linux_profile.0.admin_username:             "" => "ubuntu"
  linux_profile.0.ssh_key.#:                  "" => "1"
  linux_profile.0.ssh_key.0.key_data:         "" => "ssh-rsa"
  location:                                   "" => "centralus"
  name:                                       "" => "k8stest"
  resource_group_name:                        "" => "azure-k8stest"
  service_principal.#:                        "" => "1"
  service_principal.2782116410.client_id:     "" => ""
  service_principal.2782116410.client_secret: "<sensitive>" => "<sensitive>"
  tags.%:                                     "" => "1"
  tags.Environment:                           "" => "Development"
azurerm_kubernetes_cluster.k8s: Still creating... (10s elapsed)
azurerm_kubernetes_cluster.k8s: Still creating... (20s elapsed)
azurerm_kubernetes_cluster.k8s: Still creating... (30s elapsed)
azurerm_kubernetes_cluster.k8s: Still creating... (40s elapsed)
azurerm_kubernetes_cluster.k8s: Still creating... (50s elapsed)
azurerm_kubernetes_cluster.k8s: Still creating... (1m0s elapsed)
```

5. In the Azure portal, select **All services** in the left menu to see the resources created for your new Kubernetese cluster.



Recover from a Cloud Shell timeout

If the Cloud Shell session times out, you can perform the following steps to recover:

1. Start a Cloud Shell session.

2. Change to the directory containing your Terraform configuration files.

```
cd /clouddrive/terraform-aks-k8s
```

3. Run the following command:

```
export KUBECONFIG=./azurek8s
```

Test the Kubernetes cluster

The Kubernetes tools can be used to verify the newly created cluster.

1. Get the Kubernetes configuration from the Terraform state and store it in a file that kubectl can read.

```
echo "$(terraform output kube_config)" > ./azurek8s
```

2. Set an environment variable so that kubectl picks up the correct config.

```
export KUBECONFIG=./azurek8s
```

3. Verify the health of the cluster.

```
kubectl get nodes
```

You should see the details of your worker nodes, and they should all have a status **Ready**, as shown in the following image:

Next steps

In this article, you learned how to use Terraform and AKS to create a Kubernetes cluster. Here are some additional resources to help you learn more about Terraform on Azure:

Terraform Hub in Microsoft.com
Terraform Azure provider documentation
Terraform Azure provider source
Terraform Azure modules

Create persistent volumes with Azure disks for Azure Kubernetes Service (AKS)

8/24/2018 • 5 minutes to read • Edit Online

A persistent volume represents a piece of storage that has been provisioned for use with Kubernetes pods. A persistent volume can be used by one or many pods and can be dynamically or statically provisioned. For more information on Kubernetes persistent volumes, see Kubernetes persistent volumes. This article shows you how to use persistent volumes with Azure disks in an Azure Kubernetes Service (AKS) cluster.

> **NOTE**
> An Azure disk can only be mounted with *Access mode* type *ReadWriteOnce*, which makes it available to only a single AKS node. If needing to share a persistent volume across multiple nodes, consider using Azure Files.

Built in storage classes

A storage class is used to define how a unit of storage is dynamically created with a persistent volume. For more information on Kubernetes storage classes, see Kubernetes Storage Classes.

Each AKS cluster includes two pre-created storage classes, both configured to work with Azure disks:

- The *default* storage class provisions a standard Azure disk.
  - Standard storage is backed by HDDs, and delivers cost-effective storage while still being performant. Standard disks are ideal for a cost effective dev and test workload.
- The *managed-premium* storage class provisions a premium Azure disk.
  - Premium disks are backed by SSD-based high-performance, low-latency disk. Perfect for VMs running production workload. If the AKS nodes in your cluster use premium storage, select the *managed-premium* class.

Use the kubectl get sc command to see the pre-created storage classes. The following example shows the pre-create storage classes available within an AKS cluster:

```
$ kubectl get sc

NAME                 PROVISIONER                AGE
default (default)    kubernetes.io/azure-disk   1h
managed-premium      kubernetes.io/azure-disk   1h
```

> **NOTE**
> Persistent volume claims are specified in GiB but Azure managed disks are billed by SKU for a specific size. These SKUs range from 32GiB for S4 or P4 disks to 4TiB for S50 or P50 disks. The throughput and IOPS performance of a Premium managed disk depends on the both the SKU and the instance size of the nodes in the AKS cluster. For more information, see Pricing and Performance of Managed Disks.

Create a persistent volume claim

A persistent volume claim (PVC) is used to automatically provision storage based on a storage class. In this case, a

PVC can use one of the pre-created storage classes to create a standard or premium Azure managed disk.

Create a file named `azure-premium.yaml`, and copy in the following manifest. The claim requests a disk named `azure-managed-disk` that is *5GB* in size with *ReadWriteOnce* access. The *managed-premium* storage class is specified as the storage class.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: azure-managed-disk
spec:
  accessModes:
  - ReadWriteOnce
  storageClassName: managed-premium
  resources:
    requests:
      storage: 5Gi
```

TIP

To create a disk that uses standard storage, use `storageClassName: default` rather than *managed-premium*.

Create the persistent volume claim with the kubectl create command and specify your *azure-premium.yaml* file:

```
$ kubectl create -f azure-premium.yaml

persistentvolumeclaim/azure-managed-disk created
```

Use the persistent volume

Once the persistent volume claim has been created and the disk successfully provisioned, a pod can be created with access to the disk. The following manifest creates a basic NGINX pod that uses the persistent volume claim named *azure-managed-disk* to mount the Azure disk at the path `/mnt/azure`.

Create a file named `azure-pvc-disk.yaml`, and copy in the following manifest.

```
kind: Pod
apiVersion: v1
metadata:
  name: mypod
spec:
  containers:
    - name: myfrontend
      image: nginx
      volumeMounts:
      - mountPath: "/mnt/azure"
        name: volume
  volumes:
    - name: volume
      persistentVolumeClaim:
        claimName: azure-managed-disk
```

Create the pod with the kubectl create command, as shown in the following example:

```
$ kubectl create -f azure-pvc-disk.yaml

pod/mypod created
```

You now have a running pod with your Azure disk mounted in the `/mnt/azure` directory. This configuration can be seen when inspecting your pod via `kubectl describe pod mypod`, as shown in the following condensed example:

```
$ kubectl describe pod mypod

[...]
Volumes:
  volume:
    Type:       PersistentVolumeClaim (a reference to a PersistentVolumeClaim in the same namespace)
    ClaimName:  azure-managed-disk
    ReadOnly:   false
  default-token-smm2n:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-smm2n
    Optional:    false

Events:
  Type    Reason                 Age   From                               Message
  ----    ------                 ----  ----                               -------
  Normal  Scheduled              2m    default-scheduler                  Successfully assigned mypod to aks-
nodepool1-79590246-0
  Normal  SuccessfulMountVolume  2m    kubelet, aks-nodepool1-79590246-0  MountVolume.SetUp succeeded for
volume "default-token-smm2n"
  Normal  SuccessfulMountVolume  1m    kubelet, aks-nodepool1-79590246-0  MountVolume.SetUp succeeded for
volume "pvc-faf0f176-8b8d-11e8-923b-deb28c58d242"
[...]
```

Back up a persistent volume

To back up the data in your persistent volume, take a snapshot of the managed disk for the volume. You can then use this snapshot to create a restored disk and attach to pods as a means of restoring the data.

First, get the volume name with the `kubectl get pvc` command, such as for the PVC named *azure-managed-disk*:

```
$ kubectl get pvc azure-managed-disk

NAME                 STATUS    VOLUME                                     CAPACITY   ACCESS MODES
STORAGECLASS      AGE
azure-managed-disk   Bound     pvc-faf0f176-8b8d-11e8-923b-deb28c58d242   5Gi        RWO            managed-
premium   3m
```

This volume name forms the underlying Azure disk name. Query for the disk ID with az disk list and provide your PVC volume name, as shown in the following example:

```
$ az disk list --query '[].id | [?contains(@,`pvc-faf0f176-8b8d-11e8-923b-deb28c58d242`)]' -o tsv

/subscriptions/<guid>/resourceGroups/MC_MYRESOURCEGROUP_MYAKSCLUSTER_EASTUS/providers/MicrosoftCompute/disks/ku
bernetes-dynamic-pvc-faf0f176-8b8d-11e8-923b-deb28c58d242
```

Use the disk ID to create a snapshot disk with az snapshot create. The following example creates a snapshot named *pvcSnapshot* in the same resource group as the AKS cluster (*MC_myResourceGroup_myAKSCluster_eastus*). You may encounter permission issues if you create snapshots and restore disks in resource groups that the AKS cluster does not have access to.

```
$ az snapshot create \
    --resource-group MC_myResourceGroup_myAKSCluster_eastus \
    --name pvcSnapshot \
    --source
/subscriptions/<guid>/resourceGroups/MC_myResourceGroup_myAKSCluster_eastus/providers/MicrosoftCompute/disks/ku
bernetes-dynamic-pvc-faf0f176-8b8d-11e8-923b-deb28c58d242
```

Depending on the amount of data on your disk, it may take a few minutes to create the snapshot.

Restore and use a snapshot

To restore the disk and use it with a Kubernetes pod, use the snapshot as a source when you create a disk with az disk create. This operation preserves the original resource if you then need to access the original data snapshot. The following example creates a disk named *pvcRestored* from the snapshot named *pvcSnapshot*:

```
az disk create --resource-group MC_myResourceGroup_myAKSCluster_eastus --name pvcRestored --source pvcSnapshot
```

To use the restored disk with a pod, specify the ID of the disk in the manifest. Get the disk ID with the az disk show command. The following example gets the disk ID for *pvcRestored* created in the previous step:

```
az disk show --resource-group MC_myResourceGroup_myAKSCluster_eastus --name pvcRestored --query id -o tsv
```

Create a pod manifest named `azure-restored.yaml` and specify the disk URI obtained in the previous step. The following example creates a basic NGINX web server, with the restored disk mounted as a volume at */mnt/azure*:

```
kind: Pod
apiVersion: v1
metadata:
  name: mypodrestored
spec:
  containers:
    - name: myfrontendrestored
      image: nginx
      volumeMounts:
      - mountPath: "/mnt/azure"
        name: volume
  volumes:
    - name: volume
      azureDisk:
        kind: Managed
        diskName: pvcRestored
        diskURI:
/subscriptions/<guid>/resourceGroups/MC_myResourceGroupAKS_myAKSCluster_eastus/providers/Microsoft.Compute/disk
s/pvcRestored
```

Create the pod with the kubectl create command, as shown in the following example:

```
$ kubectl create -f azure-restored.yaml

pod/mypodrestored created
```

You can use `kubectl describe pod mypodrestored` to view details of the pod, such as the following condensed example that shows the volume information:

```
$ kubectl describe pod mypodrestored

[...]
Volumes:
  volume:
    Type:         AzureDisk (an Azure Data Disk mount on the host and bind mount to the pod)
    DiskName:     pvcRestored
    DiskURI:      /subscriptions/19da35d3-9a1a-4f3b-9b9c-
3c56ef409565/resourceGroups/MC_myResourceGroupAKS_myAKSCluster_eastus/providers/Microsoft.Compute/disks/pvcRest
ored
    Kind:         Managed
    FSType:       ext4
    CachingMode:  ReadWrite
    ReadOnly:     false
[...]
```

Next steps

Learn more about Kubernetes persistent volumes using Azure disks.

Kubernetes plugin for Azure disks

Manually create and use Kubernetes volume with Azure disks in Azure Kubernetes Service (AKS)

9/27/2018 • 3 minutes to read • Edit Online

Container-based applications often need to access and persist data in an external data volume. Azure disks can be used as this external data store. In AKS, volumes can be created dynamically using persistent volume claims, or you can manually create and attach an Azure disk directly. This article shows you how to manually create an Azure disk and attach it to a pod in AKS.

For more information on Kubernetes volumes, see Kubernetes volumes.

Before you begin

This article assumes that you have an existing AKS cluster. If you need an AKS cluster, see the AKS quickstart using the Azure CLI or using the Azure portal.

You also need the Azure CLI version 2.0.46 or later installed and configured. Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI.

Create an Azure disk

When you create an Azure disk for use with AKS, you can create the disk resource in the **node** resource group. This approach allows the AKS cluster to access and manage the disk resource. If you instead create the disk in a separate resource group, you must grant the Azure Kubernetes Service (AKS) service principal for your cluster the `Contributor` role to the disk's resource group.

For this article, create the disk in the node resource group. First, get the resource group name with the az aks show command and add the `--query nodeResourceGroup` query parameter. The following example gets the node resource group for the AKS cluster name *myAKSCluster* in the resource group name *myResourceGroup*:

```
$ az aks show --resource-group myResourceGroup --name myAKSCluster --query nodeResourceGroup -o tsv

MC_myResourceGroup_myAKSCluster_eastus
```

Now create a disk using the az disk create command. Specify the node resource group name obtained in the previous command, and then a name for the disk resource, such as *myAKSDisk*. The following example creates a *20*GiB disk, and outputs the ID of the disk once created:

```
az disk create \
  --resource-group MC_myResourceGroup_myAKSCluster_eastus \
  --name myAKSDisk  \
  --size-gb 20 \
  --query id --output tsv
```

NOTE

Azure disks are billed by SKU for a specific size. These SKUs range from 32GiB for S4 or P4 disks to 8TiB for S60 or P60 disks. The throughput and IOPS performance of a Premium managed disk depends on both the SKU and the instance size of the nodes in the AKS cluster. See Pricing and Performance of Managed Disks.

The disk resource ID is displayed once the command has successfully completed, as shown in the following example output. This disk ID is used to mount the disk in the next step.

```
/subscriptions/<subscriptionID>/resourceGroups/MC_myAKSCluster_myAKSCluster_eastus/providers/Microsoft.Compute
/disks/myAKSDisk
```

Mount disk as volume

To mount the Azure disk into your pod, configure the volume in the container spec. Create a new file named `azure-disk-pod.yaml` with the following contents. Update `diskName` with the name of the disk created in the previous step, and `diskURI` with the disk ID shown in output of the disk create command. If desired, update the `mountPath`, which is the path where the Azure disk is mounted in the pod.

```
apiVersion: v1
kind: Pod
metadata:
 name: azure-disk-pod
spec:
 containers:
  - image: microsoft/sample-aks-helloworld
    name: azure
    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
```

Use the `kubectl` command to create the pod.

```
kubectl apply -f azure-disk-pod.yaml
```

You now have a running pod with an Azure disk mounted at `/mnt/azure`. You can use `kubectl describe pod azure-disk-pod` to verify the disk is mounted successfully.

Next steps

For more information about AKS clusters interact with Azure disks, see the Kubernetes plugin for Azure Disks.

Persistent volumes with Azure files

8/30/2018 • 5 minutes to read • Edit Online

A persistent volume is a piece of storage that has been created for use in a Kubernetes cluster. A persistent volume can be used by one or many pods and can be dynamically or statically created. This document details **dynamic creation** of an Azure file share as a persistent volume.

For more information on Kubernetes persistent volumes, including static creation, see Kubernetes persistent volumes.

Create a storage account

When dynamically creating an Azure file share as a Kubernetes volume, any storage account can be used as long as it is in the AKS **node** resource group. This group is the one with the *MC_* prefix that was created by the provisioning of the resources for the AKS cluster. Get the resource group name with the [az aks show][az-aks-show] command.

```
$ az aks show --resource-group myResourceGroup --name myAKSCluster --query nodeResourceGroup -o tsv

MC_myResourceGroup_myAKSCluster_eastus
```

Use the az storage account create command to create the storage account.

Update `--resource-group` with the name of the resource group gathered in the last step, and `--name` to a name of your choice. Provide your own unique storage account name:

```
az storage account create --resource-group MC_myResourceGroup_myAKSCluster_eastus --name mystorageaccount --
sku Standard_LRS
```

NOTE

Azure Files currently only work with Standard storage. If you use Premium storage, the volume fails to provision.

Create a storage class

A storage class is used to define how an Azure file share is created. A storage account can be specified in the class. If a storage account is not specified, a *skuName* and *location* must be specified, and all storage accounts in the associated resource group are evaluated for a match. For more information on Kubernetes storage classes for Azure Files, see Kubernetes Storage Classes.

Create a file named `azure-file-sc.yaml` and copy in the following example manifest. Update the *storageAccount* value with the name of your storage account created in the previous step. For more information on *mountOptions*, see the Mount options section.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: azurefile
provisioner: kubernetes.io/azure-file
mountOptions:
  - dir_mode=0777
  - file_mode=0777
  - uid=1000
  - gid=1000
parameters:
  skuName: Standard_LRS
  storageAccount: mystorageaccount
```

Create the storage class with the kubectl apply command:

```
kubectl apply -f azure-file-sc.yaml
```

Create a cluster role and binding

AKS clusters use Kubernetes role-based access control (RBAC) to limit actions that can be performed. *Roles* define the permissions to grant, and *bindings* apply them to desired users. These assignments can be applied to a given namespace, or across the entire cluster. For more information, see Using RBAC authorization.

To allow the Azure platform to create the required storage resources, create a *ClusterRole* and *ClusterRoleBinding*. Create a file named `azure-pvc-roles.yaml` and copy in the following YAML:

```
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRole
metadata:
  name: system:azure-cloud-provider
rules:
- apiGroups: ['']
  resources: ['secrets']
  verbs:     ['get','create']
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: system:azure-cloud-provider
roleRef:
  kind: ClusterRole
  apiGroup: rbac.authorization.k8s.io
  name: system:azure-cloud-provider
subjects:
- kind: ServiceAccount
  name: persistent-volume-binder
  namespace: kube-system
```

Assign the permissions with the kubectl apply command:

```
kubectl apply -f azure-pvc-roles.yaml
```

Create a persistent volume claim

A persistent volume claim (PVC) uses the storage class object to dynamically provision an Azure file share. The following YAML can be used to create a persistent volume claim *5GB* in size with *ReadWriteMany* access. For

more information on access modes, see the Kubernetes persistent volume documentation.

Now create a file named `azure-file-pvc.yaml` and copy in the following YAML. Make sure that the *storageClassName* matches the storage class created in the last step:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: azurefile
spec:
  accessModes:
    - ReadWriteMany
  storageClassName: azurefile
  resources:
    requests:
      storage: 5Gi
```

Create the persistent volume claim with the kubectl apply command:

```
kubectl apply -f azure-file-pvc.yaml
```

Once completed, the file share will be created. A Kubernetes secret is also created that includes connection information and credentials. You can use the kubectl get command to view the status of the PVC:

```
$ kubectl get pvc azurefile

NAME        STATUS    VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
azurefile   Bound     pvc-8436e62e-a0d9-11e5-8521-5a8664dc0477   5Gi        RWX            azurefile      5m
```

Use the persistent volume

The following YAML creates a pod that uses the persistent volume claim *azurefile* to mount the Azure file share at the */mnt/azure* path.

Create a file named `azure-pvc-files.yaml`, and copy in the following YAML. Make sure that the *claimName* matches the PVC created in the last step.

```
kind: Pod
apiVersion: v1
metadata:
  name: mypod
spec:
  containers:
    - name: myfrontend
      image: nginx
      volumeMounts:
      - mountPath: "/mnt/azure"
        name: volume
  volumes:
    - name: volume
      persistentVolumeClaim:
        claimName: azurefile
```

Create the pod with the kubectl apply command.

```
kubectl apply -f azure-pvc-files.yaml
```

You now have a running pod with your Azure disk mounted in the */mnt/azure* directory. This configuration can be seen when inspecting your pod via `kubectl describe pod mypod`. The following condensed example output shows the volume mounted in the container:

```
Containers:
  myfrontend:
    Container ID:   docker://053bc9c0df72232d755aa040bfba8b533fa696b123876108dec400e364d2523e
    Image:          nginx
    Image ID:       docker-
pullable://nginx@sha256:d85914d547a6c92faa39ce7058bd7529baacab7e0cd4255442b04577c4d1f424
    State:          Running
      Started:      Wed, 15 Aug 2018 22:22:27 +0000
    Ready:          True
    Mounts:
      /mnt/azure from volume (rw)
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-8rv4z (ro)
[...]
Volumes:
  volume:
    Type:       PersistentVolumeClaim (a reference to a PersistentVolumeClaim in the same namespace)
    ClaimName:  azurefile2
    ReadOnly:   false
[...]
```

Mount options

Default *fileMode* and *dirMode* values differ between Kubernetes versions as described in the following table.

VERSION	VALUE
v1.6.x, v1.7.x	0777
v1.8.0-v1.8.5	0700
v1.8.6 or above	0755
v1.9.0	0700
v1.9.1 or above	0755

If using a cluster of version 1.8.5 or greater and dynamically creating the persistent volume with a storage class, mount options can be specified on the storage class object. The following example sets *0777*:

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: azurefile
provisioner: kubernetes.io/azure-file
mountOptions:
  - dir_mode=0777
  - file_mode=0777
  - uid=1000
  - gid=1000
parameters:
  skuName: Standard_LRS
```

If using a cluster of version 1.8.5 or greater and statically creating the persistent volume object, mount options need to be specified on the *PersistentVolume* object. for more information on statically creating a persistent

volume, see Static Persistent Volumes.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: azurefile
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteMany
  azureFile:
    secretName: azure-secret
    shareName: azurefile
    readOnly: false
  mountOptions:
  - dir_mode=0777
  - file_mode=0777
  - uid=1000
  - gid=1000
```

If using a cluster of version 1.8.0 - 1.8.4, a security context can be specified with the *runAsUser* value set to *0*. For more information on Pod security context, see Configure a Security Context.

Next steps

Learn more about Kubernetes persistent volumes using Azure Files.

Kubernetes plugin for Azure Files

Manually create and use an Azure Files share in Azure Kubernetes Service (AKS)

9/27/2018 • 2 minutes to read • Edit Online

Container-based applications often need to access and persist data in an external data volume. If multiple pods need concurrent access to the same storage volume, you can use Azure Files to connect using the Server Message Block (SMB) protocol. This article shows you how to manually create an Azure Files share and attach it to a pod in AKS.

For more information on Kubernetes volumes, see Kubernetes volumes.

Before you begin

This article assumes that you have an existing AKS cluster. If you need an AKS cluster, see the AKS quickstart using the Azure CLI or using the Azure portal.

You also need the Azure CLI version 2.0.46 or later installed and configured. Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI.

Create an Azure file share

Before you can use Azure Files as a Kubernetes volume, you must create an Azure Storage account and the file share. The following script creates a resource group named *myAKSShare*, a storage account, and a Files share named *aksshare*:

```
#!/bin/bash

# Change these four parameters
AKS_PERS_STORAGE_ACCOUNT_NAME=mystorageaccount$RANDOM
AKS_PERS_RESOURCE_GROUP=myAKSShare
AKS_PERS_LOCATION=eastus
AKS_PERS_SHARE_NAME=aksshare

# Create the Resource Group
az group create --name $AKS_PERS_RESOURCE_GROUP --location $AKS_PERS_LOCATION

# Create the storage account
az storage account create -n $AKS_PERS_STORAGE_ACCOUNT_NAME -g $AKS_PERS_RESOURCE_GROUP -l $AKS_PERS_LOCATION -
-sku Standard_LRS

# Export the connection string as an environment variable, this is used when creating the Azure file share
export AZURE_STORAGE_CONNECTION_STRING=`az storage account show-connection-string -n
$AKS_PERS_STORAGE_ACCOUNT_NAME -g $AKS_PERS_RESOURCE_GROUP -o tsv`

# Create the file share
az storage share create -n $AKS_PERS_SHARE_NAME

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

# Echo storage account name and key
echo Storage account name: $AKS_PERS_STORAGE_ACCOUNT_NAME
echo Storage account key: $STORAGE_KEY
```

Make a note of the storage account name and key shown at the end of the script output. These values are needed when you create the Kubernetes volume in one of the following steps.

Create a Kubernetes secret

Kubernetes needs credentials to access the file share created in the previous step. These credentials are stored in a Kubernetes secret, which is referenced when you create a Kubernetes pod.

Use the `kubectl create secret` command to create the secret. The following example creates a shared named *azure-secret*. Replace *STORAGE_ACCOUNT_NAME* with your storage account name shown in the output of the previous step, and *STORAGE_ACCOUNT_KEY* with your storage key:

```
kubectl create secret generic azure-secret --from-literal=azurestorageaccountname=STORAGE_ACCOUNT_NAME --from-literal=azurestorageaccountkey=STORAGE_ACCOUNT_KEY
```

Mount the file share as a volume

To mount the Azure Files share into your pod, configure the volume in the container spec. Create a new file named `azure-files-pod.yaml` with the following contents. If you changed the name of the Files share or secret name, update the *shareName* and *secretName*. If desired, update the `mountPath`, which is the path where the Files share is mounted in the pod.

```
apiVersion: v1
kind: Pod
metadata:
 name: azure-files-pod
spec:
 containers:
  - image: microsoft/sample-aks-helloworld
    name: azure
    volumeMounts:
      - name: azure
        mountPath: /mnt/azure
 volumes:
  - name: azure
    azureFile:
      secretName: azure-secret
      shareName: aksshare
      readOnly: false
```

Use the `kubectl` command to create the pod.

```
kubectl apply -f azure-files-pod.yaml
```

You now have a running pod with an Azure Files share mounted at */mnt/azure*. You can use `kubectl describe pod azure-files-pod` to verify the share is mounted successfully.

Next steps

For more information about AKS clusters interact with Azure Files, see the Kubernetes plugin for Azure Files.

Network configuration in Azure Kubernetes Service (AKS)

10/8/2018 • 9 minutes to read • Edit Online

When you create an Azure Kubernetes Service (AKS) cluster, you can select from two networking options: **Basic** or **Advanced**.

Basic networking

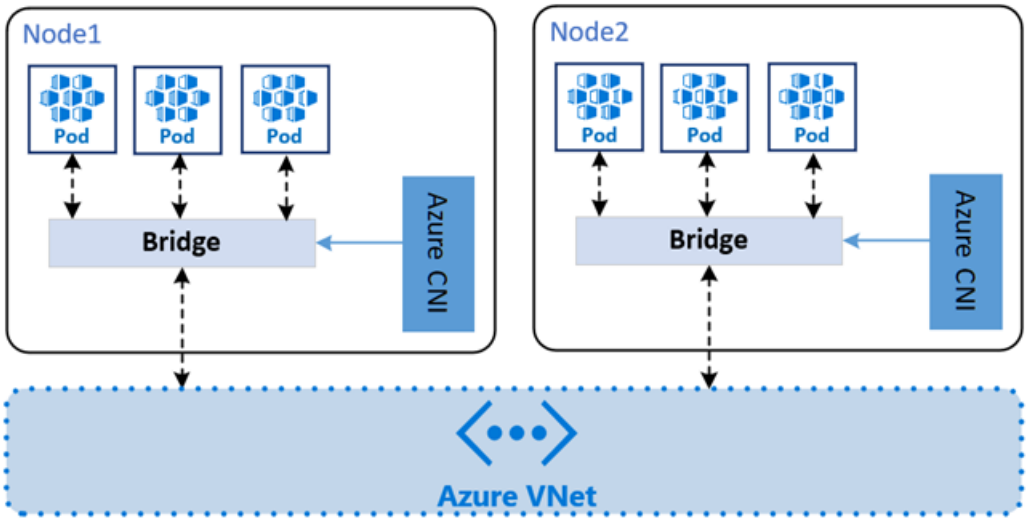
The **Basic** networking option is the default configuration for AKS cluster creation. The network configuration of the cluster and its pods is managed completely by Azure, and is appropriate for deployments that do not require custom VNet configuration. You do not have control over network configuration such as subnets or the IP address ranges assigned to the cluster when you select Basic networking.

Nodes in an AKS cluster configured for Basic networking use the kubenet Kubernetes plugin.

Advanced networking

Advanced networking places your pods in an Azure Virtual Network (VNet) that you configure, providing them automatic connectivity to VNet resources and integration with the rich set of capabilities that VNets offer. Advanced networking is available when deploying AKS clusters with the Azure portal, Azure CLI, or with a Resource Manager template.

Nodes in an AKS cluster configured for Advanced networking use the Azure Container Networking Interface (CNI) Kubernetes plugin.



Advanced networking features

Advanced networking provides the following benefits:

- Deploy your AKS cluster into an existing VNet, or create a new VNet and subnet for your cluster.
- Every pod in the cluster is assigned an IP address in the VNet, and can directly communicate with other pods in the cluster, and other nodes in the VNet.
- A pod can connect to other services in a peered VNet, and to on-premises networks over ExpressRoute and site-to-site (S2S) VPN connections. Pods are also reachable from on-premises.

- Expose a Kubernetes service externally or internally through the Azure Load Balancer. Also a feature of Basic networking.
- Pods in a subnet that have service endpoints enabled can securely connect to Azure services, for example Azure Storage and SQL DB.
- Use user-defined routes (UDR) to route traffic from pods to a Network Virtual Appliance.
- Pods can access resources on the public Internet. Also a feature of Basic networking.

Advanced networking prerequisites

- The virtual network for the AKS cluster must allow outbound internet connectivity.
- Do not create more than one AKS cluster in the same subnet.
- AKS clusters may not use `169.254.0.0/16`, `172.30.0.0/16`, or `172.31.0.0/16` for the Kubernetes service address range.
- The service principal used by the AKS cluster must have at least Network Contributor permissions on the subnet within your virtual network. If you wish to define a custom role instead of using the built-in Network Contributor role, the following permissions are required:
  - `Microsoft.Network/virtualNetworks/subnets/join/action`
  - `Microsoft.Network/virtualNetworks/subnets/read`

Plan IP addressing for your cluster

Clusters configured with Advanced networking require additional planning. The size of your virtual network and its subnet must accommodate both the number of pods you plan to run as well as the number of nodes for the cluster.

IP addresses for the pods and the cluster's nodes are assigned from the specified subnet within the virtual network. Each node is configured with a primary IP, which is the IP of the node and 30 additional IP addresses pre-configured by Azure CNI that are assigned to pods scheduled to the node. When you scale out your cluster, each node is similarly configured with IP addresses from the subnet.

The IP address plan for an AKS cluster consists of a virtual network, at least one subnet for nodes and pods, and a Kubernetes service address range.

ADDRESS RANGE / AZURE RESOURCE	LIMITS AND SIZING
Virtual network	The Azure virtual network can be as large as /8, but is limited to 65,536 configured IP addresses.
Subnet	Must be large enough to accommodate the nodes, pods, and all Kubernetes and Azure resources that might be provisioned in your cluster. For example, if you deploy an internal Azure Load Balancer, its front-end IPs are allocated from the cluster subnet, not public IPs. To calculate *minimum* subnet size: `(number of nodes) + (number of nodes * pods per node)` Example for a 50 node cluster: `(50) + (50 * 30) = 1,550` (/21 or larger)
Kubernetes service address range	This range should not be used by any network element on or connected to this virtual network. Service address CIDR must be smaller than /12.

ADDRESS RANGE / AZURE RESOURCE	LIMITS AND SIZING
Kubernetes DNS service IP address	IP address within the Kubernetes service address range that will be used by cluster service discovery (kube-dns).
Docker bridge address	IP address (in CIDR notation) used as the Docker bridge IP address on nodes. Default of 172.17.0.1/16.

Maximum pods per node

The default maximum number of pods per node in an AKS cluster varies between Basic and Advanced networking, and the method of cluster deployment.

Default maximum

These are the *default* maximums when you deploy an AKS cluster without specifying the maximum number of pods at deployment time:

DEPLOYMENT METHOD	BASIC	ADVANCED	CONFIGURABLE AT DEPLOYMENT
Azure CLI	110	30	Yes
Resource Manager template	110	30	Yes
Portal	110	30	No

Configure maximum - new clusters

To specify a different maximum number of pods per node when you deploy an AKS cluster:

- **Azure CLI**: Specify the `--max-pods` argument when you deploy a cluster with the az aks create command.
- **Resource Manager template**: Specify the `maxPods` property in the ManagedClusterAgentPoolProfile object when you deploy a cluster with a Resource Manager template.
- **Azure portal**: You cannot modify the maximum number of pods per node when you deploy a cluster with the Azure portal. Advanced networking clusters are limited to 30 pods per node when deployed in the Azure portal.

Configure maximum - existing clusters

You can't change the maximum pods per node on an existing AKS cluster. You can adjust the number only when you initially deploy the cluster.

Deployment parameters

When you create an AKS cluster, the following parameters are configurable for advanced networking:

Virtual network: The virtual network into which you want to deploy the Kubernetes cluster. If you want to create a new virtual network for your cluster, select *Create new* and follow the steps in the *Create virtual network* section. For information about the limits and quotas for an Azure virtual network, see Azure subscription and service limits, quotas, and constraints.

Subnet: The subnet within the virtual network where you want to deploy the cluster. If you want to create a new subnet in the virtual network for your cluster, select *Create new* and follow the steps in the *Create subnet* section.

Kubernetes service address range: This is the set of virtual IPs that Kubernetes assigns to services in your cluster. You can use any private address range that satisfies the following requirements:

- Must not be within the virtual network IP address range of your cluster

- Must not overlap with any other virtual networks with which the cluster virtual network peers
- Must not overlap with any on-premises IPs
- Must not be within the ranges `169.254.0.0/16`, `172.30.0.0/16`, or `172.31.0.0/16`

Although it's technically possible to specify a service address range within the same virtual network as your cluster, doing so is not recommended. Unpredictable behavior can result if overlapping IP ranges are used. For more information, see the FAQ section of this article. For more information on Kubernetes services, see Services in the Kubernetes documentation.

Kubernetes DNS service IP address: The IP address for the cluster's DNS service. This address must be within the *Kubernetes service address range*.

Docker Bridge address: The IP address and netmask to assign to the Docker bridge. This IP address must not be within the virtual network IP address range of your cluster.

Configure networking - CLI

When you create an AKS cluster with the Azure CLI, you can also configure advanced networking. Use the following commands to create a new AKS cluster with advanced networking features enabled.

First, get the subnet resource ID for the existing subnet into which the AKS cluster will be joined:

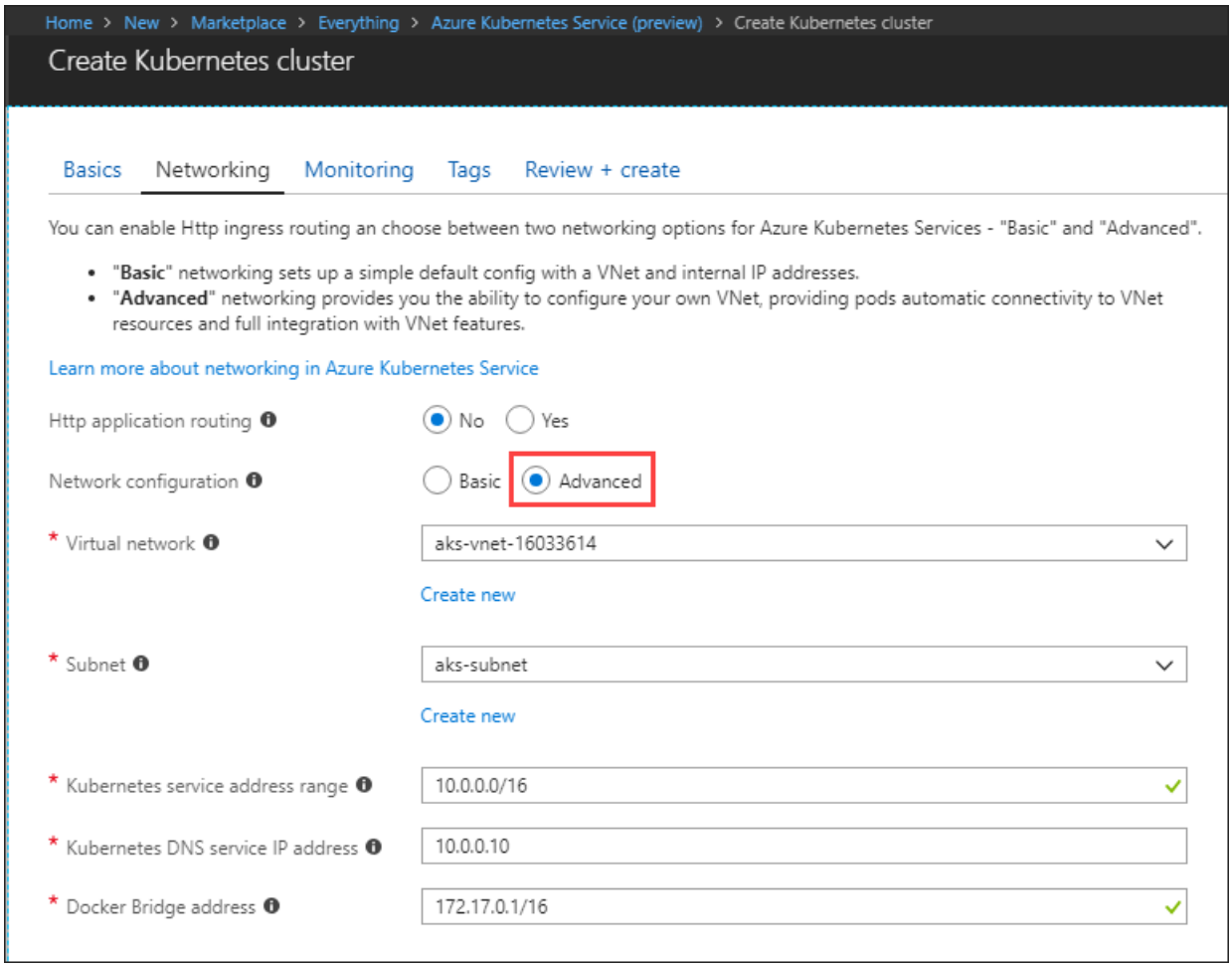
```
$ az network vnet subnet list --resource-group myVnet --vnet-name myVnet --query [].id --output tsv

/subscriptions/d5b9d4b7-6fc1-46c5-bafe-
38effaed19b2/resourceGroups/myVnet/providers/Microsoft.Network/virtualNetworks/myVnet/subnets/default
```

Use the az aks create command with the `--network-plugin azure` argument to create a cluster with advanced networking. Update the `--vnet-subnet-id` value with the subnet ID collected in the previous step:

```
az aks create --resource-group myAKSCluster --name myAKSCluster --network-plugin azure --vnet-subnet-id
<subnet-id> --docker-bridge-address 172.17.0.1/16 --dns-service-ip 10.2.0.10 --service-cidr 10.2.0.0/24
```

Configure networking - portal

The following screenshot from the Azure portal shows an example of configuring these settings during AKS cluster creation:



Frequently asked questions

The following questions and answers apply to the **Advanced** networking configuration.

- *Can I deploy VMs in my cluster subnet?*

  No. Deploying VMs in the subnet used by your Kubernetes cluster is not supported. VMs may be deployed in the same virtual network, but in a different subnet.

- *Can I configure per-pod network policies?*

  No. Per-pod network policies are currently unsupported.

- *Is the maximum number of pods deployable to a node configurable?*

  Yes, when you deploy a cluster with the Azure CLI or a Resource Manager template. See Maximum pods per node.

  You can't change the maximum number of pods per node on an existing cluster.

- *How do I configure additional properties for the subnet that I created during AKS cluster creation? For example, service endpoints.*

  The complete list of properties for the virtual network and subnets that you create during AKS cluster creation can be configured in the standard virtual network configuration page in the Azure portal.

- *Can I use a different subnet within my cluster virtual network for the* **Kubernetes service address range**?

  It's not recommended, but this configuration is possible. The service address range is a set of virtual IPs (VIPs) that Kubernetes assigns to the services in your cluster. Azure Networking has no visibility into the service IP range of the Kubernetes cluster. Because of the lack of visibility into the cluster's service address range, it's possible to later create a new subnet in the cluster virtual network that overlaps with the service

address range. If such an overlap occurs, Kubernetes could assign a service an IP that's already in use by another resource in the subnet, causing unpredictable behavior or failures. By ensuring you use an address range outside the cluster's virtual network, you can avoid this overlap risk.

Next steps

Networking in AKS

Learn more about networking in AKS in the following articles:

- Use a static IP address with the Azure Kubernetes Service (AKS) load balancer
- Use an internal load balancer with Azure Container Service (AKS)
- Create a basic ingress controller with external network connectivity
- Enable the HTTP application routing add-on
- Create an ingress controller that uses an internal, private network and IP address
- Create an ingress controller with a dynamic public IP and configure Let's Encrypt to automatically generate TLS certificates
- Create an ingress controller with a static public IP and configure Let's Encrypt to automatically generate TLS certificates

ACS Engine

Azure Container Service Engine (ACS Engine) is an open-source project that generates Azure Resource Manager templates you can use for deploying Docker-enabled clusters on Azure. Kubernetes, DC/OS, Swarm Mode, and Swarm orchestrators can be deployed with ACS Engine.

Kubernetes clusters created with ACS Engine support both the kubenet and Azure CNI plugins. As such, both basic and advanced networking scenarios are supported by ACS Engine.

Use a static public IP address with the Azure Kubernetes Service (AKS) load balancer

9/27/2018 • 3 minutes to read • Edit Online

By default, the public IP address assigned to a load balancer resource created by an AKS cluster is only valid for the lifespan of that resource. If you delete the Kubernetes service, the associated load balancer and IP address are also deleted. If you want to assign a specific IP address or retain an IP address for redeployed Kubernetes services, you can create and use a static public IP address.

This article shows you how to create a static public IP address and assign it to your Kubernetes service.

Before you begin

This article assumes that you have an existing AKS cluster. If you need an AKS cluster, see the AKS quickstart using the Azure CLI or using the Azure portal.

You also need the Azure CLI version 2.0.46 or later installed and configured. Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI.

Create a static IP address

When you create a static public IP address for use with AKS, the IP address resource must be created in the **node** resource group. Get the resource group name with the az aks show command and add the `--query nodeResourceGroup` query parameter. The following example gets the node resource group for the AKS cluster name *myAKSCluster* in the resource group name *myResourceGroup*:

```
$ az aks show --resource-group myResourceGroup --name myAKSCluster --query nodeResourceGroup -o tsv

MC_myResourceGroup_myAKSCluster_eastus
```

Now create a static public IP address with the az network public ip create command. Specify the node resource group name obtained in the previous command, and then a name for the IP address resource, such as *myAKSPublicIP*:

```
az network public-ip create \
    --resource-group MC_myResourceGroup_myAKSCluster_eastus \
    --name myAKSPublicIP \
    --allocation-method static
```

The IP address is shown, as shown in the following condensed example output:

```
{
  "publicIp": {
    "dnsSettings": null,
    "etag": "W/\"6b6fb15c-5281-4f64-b332-8f68f46e1358\"",
    "id":
"/subscriptions/<SubscriptionID>/resourceGroups/MC_myResourceGroup_myAKSCluster_eastus/providers/Microsoft.Net
work/publicIPAddresses/myAKSPublicIP",
    "idleTimeoutInMinutes": 4,
    "ipAddress": "40.121.183.52",
    [..]
  }
```

You can later get the public IP address using the az network public-ip list command. Specify the name of the node resource group, and then query for the *ipAddress* as shown in the following example:

```
$ az network public-ip list --resource-group MC_myResourceGroup_myAKSCluster_eastus --query [0].ipAddress --
output tsv

40.121.183.52
```

Create a service using the static IP address

To create a service with the static public IP address, add the `loadBalancerIP` property and the value of the static public IP address to the YAML manifest. Create a file named `load-balancer-service.yaml` and copy in the following YAML. Provide your own public IP address created in the previous step.

```
apiVersion: v1
kind: Service
metadata:
  name: azure-load-balancer
spec:
  loadBalancerIP: 40.121.183.52
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    app: azure-load-balancer
```

Create the service and deployment with the `kubectl apply` command.

```
kubectl apply -f load-balancer-service.yaml
```

Troubleshoot

If the static IP address defined in the *loadBalancerIP* property of the Kubernetes service manifest does not exist, or has not been created in the node resource group, the load balancer service creation fails. To troubleshoot, review the service creation events with the kubectl describe command. Provide the name of the service as specified in the YAML manifest, as shown in the following example:

```
kubectl describe service azure-load-balancer
```

Information about the Kubernetes service resource is displayed. The *Events* at the end of the following example output indicate that the *user supplied IP Address was not found*. In these scenarios, verify that you have created the static public IP address in the node resource group and that the IP address specified in the Kubernetes service

manifest is correct.

```
Name:                     azure-load-balancer
Namespace:                default
Labels:                   <none>
Annotations:              <none>
Selector:                 app=azure-load-balancer
Type:                     LoadBalancer
IP:                       10.0.18.125
IP:                       40.121.183.52
Port:                     <unset>  80/TCP
TargetPort:               80/TCP
NodePort:                 <unset>  32582/TCP
Endpoints:                <none>
Session Affinity:         None
External Traffic Policy:  Cluster
Events:
  Type     Reason                      Age               From                Message
  ----     ------                      ----              ----                -------
  Normal   CreatingLoadBalancer        7s (x2 over 22s)  service-controller  Creating load balancer
  Warning  CreatingLoadBalancerFailed  6s (x2 over 12s)  service-controller  Error creating load balancer
(will retry): Failed to create load balancer for service default/azure-load-balancer: user supplied IP Address
40.121.183.52 was not found
```

Next steps

For additional control over the network traffic to your applications, you may want to instead create an ingress controller. You can also create an ingress controller with a static public IP address.

Use an internal load balancer with Azure Kubernetes Service (AKS)

7/12/2018 • 2 minutes to read • Edit Online

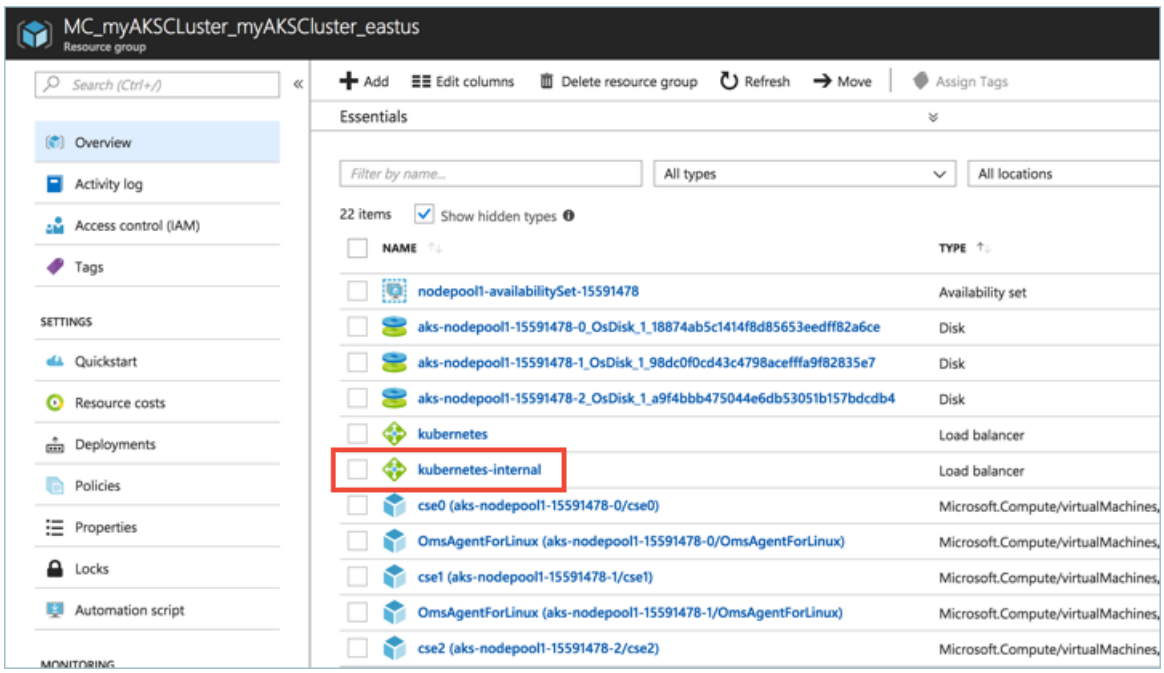
Internal load balancing makes a Kubernetes service accessible to applications running in the same virtual network as the Kubernetes cluster. This article shows you how to create and use an internal load balancer with Azure Kubernetes Service (AKS). Azure Load Balancer is available in two SKUs: Basic and Standard. AKS uses the Basic SKU.

Create an internal load balancer

To create an internal load balancer, create a service manifest with the service type *LoadBalancer* and the *azure-load-balancer-internal* annotation as shown in the following example:

```
apiVersion: v1
kind: Service
metadata:
  name: azure-vote-front
  annotations:
    service.beta.kubernetes.io/azure-load-balancer-internal: "true"
spec:
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    app: azure-vote-front
```

Once deployed with `kubetctl apply`, an Azure load balancer is created and made available on the same virtual network as the AKS cluster.



When you view the service details, the IP address in the *EXTERNAL-IP* column is the IP address of the internal load balancer, as shown in the following example:

```
$ kubectl get service azure-vote-front

NAME               TYPE           CLUSTER-IP    EXTERNAL-IP    PORT(S)        AGE
azure-vote-front   LoadBalancer   10.0.248.59   10.240.0.7     80:30555/TCP   10s
```

Specify an IP address

If you would like to use a specific IP address with the internal load balancer, add the *loadBalancerIP* property to the load balancer spec. The specified IP address must reside in the same subnet as the AKS cluster and must not already be assigned to a resource.

```
apiVersion: v1
kind: Service
metadata:
  name: azure-vote-front
  annotations:
    service.beta.kubernetes.io/azure-load-balancer-internal: "true"
spec:
  type: LoadBalancer
  loadBalancerIP: 10.240.0.25
  ports:
  - port: 80
  selector:
    app: azure-vote-front
```

When you view the service details, the IP address on the *EXTERNAL-IP* reflects the specified IP address:

```
$ kubectl get service azure-vote-front

NAME               TYPE           CLUSTER-IP     EXTERNAL-IP    PORT(S)        AGE
azure-vote-front   LoadBalancer   10.0.184.168   10.240.0.25    80:30225/TCP   4m
```

Use private networks

When you create your AKS cluster, you can specify advanced networking settings. This approach lets you deploy the cluster into an existing Azure virtual network and subnets. One scenario is to deploy your AKS cluster into a private network connected to your on-premises environment and run services only accessible internally. For more information, see advanced network configuration in AKS.

No changes to the previous steps are needed to deploy an internal load balancer in an AKS cluster that uses a private network. The load balancer is created in the same resource group as your AKS cluster but connected to your private virtual network and subnet, as shown in the following example:

```
$ kubectl get service azure-vote-front

NAME               TYPE           CLUSTER-IP    EXTERNAL-IP    PORT(S)        AGE
azure-vote-front   LoadBalancer   10.1.15.188   10.0.0.35      80:31669/TCP   1m
```

NOTE

You may need to grant the service principal for your AKS cluster the *Network Contributor* role to the resource group where your Azure virtual network resources are deployed. View the service principal with az aks show, such as `az aks show --resource-group myResourceGroup --name myAKSCluster --query "servicePrincipalProfile.clientId"`. To create a role assignment, use the az role assignment create command.

Specify a different subnet

To specify a subnet for your load balancer, add the *azure-load-balancer-internal-subnet* annotation to your service. The subnet specified must be in the same virtual network as your AKS cluster. When deployed, the load balancer *EXTERNAL-IP* address is part of the specified subnet.

```
apiVersion: v1
kind: Service
metadata:
  name: azure-vote-front
  annotations:
    service.beta.kubernetes.io/azure-load-balancer-internal: "true"
    service.beta.kubernetes.io/azure-load-balancer-internal-subnet: "apps-subnet"
spec:
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    app: azure-vote-front
```

Delete the load balancer

When all services that use the internal load balancer are deleted, the load balancer itself is also deleted.

Next steps

Learn more about Kubernetes services at the Kubernetes services documentation.

Create an ingress controller in Azure Kubernetes Service (AKS)

9/10/2018 • 4 minutes to read • Edit Online

An ingress controller is a piece of software that provides reverse proxy, configurable traffic routing, and TLS termination for Kubernetes services. Kubernetes ingress resources are used to configure the ingress rules and routes for individual Kubernetes services. Using an ingress controller and ingress rules, a single IP address can be used to route traffic to multiple services in a Kubernetes cluster.

This article shows you how to deploy the NGINX ingress controller in an Azure Kubernetes Service (AKS) cluster. Two applications are then run in the AKS cluster, each of which is accessible over the single IP address.

You can also:

- Enable the HTTP application routing add-on
- Create an ingress controller that uses an internal, private network and IP address
- Create an ingress controller with a dynamic public IP and configure Let's Encrypt to automatically generate TLS certificates
- Create an ingress controller with a static public IP address and configure Let's Encrypt to automatically generate TLS certificates

Before you begin

This article uses Helm to install the NGINX ingress controller, cert-manager, and a sample web app. You need to have Helm initialized within your AKS cluster and using a service account for Tiller. For more information on configuring and using Helm, see Install applications with Helm in Azure Kubernetes Service (AKS).

This article also requires that you are running the Azure CLI version 2.0.41 or later. Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI.

Create an ingress controller

To create the ingress controller, use `Helm` to install *nginx-ingress*.

> **TIP**
> The following example installs the ingress controller in the `kube-system` namespace. You can specify a different namespace for your own environment if desired. If your AKS cluster is not RBAC enabled, add `--set rbac.create=false` to the commands.

```
helm install stable/nginx-ingress --namespace kube-system
```

When the Kubernetes load balancer service is created for the NGINX ingress controller, a dynamic public IP address is assigned, as shown in the following example output:

```
$ kubectl get service -l app=nginx-ingress --namespace kube-system

NAME                                        TYPE           CLUSTER-IP    EXTERNAL-IP   PORT(S)
AGE
masked-otter-nginx-ingress-controller       LoadBalancer   10.0.92.99    40.117.74.8
80:31077/TCP,443:32592/TCP   7m
masked-otter-nginx-ingress-default-backend   ClusterIP      10.0.46.106   <none>        80/TCP
7m
```

No ingress rules have been created yet, so the NGINX ingress controller's default 404 page is displayed if you browse to the internal IP address. Ingress rules are configured in the following steps.

Run demo applications

To see the ingress controller in action, let's run two demo applications in your AKS cluster. In this example, Helm is used to deploy two instances of a simple 'Hello world' application.

Before you can install the sample Helm charts, add the Azure samples repository to your Helm environment as follows:

```
helm repo add azure-samples https://azure-samples.github.io/helm-charts/
```

Create the first demo application from a Helm chart with the following command:

```
helm install azure-samples/aks-helloworld
```

Now install a second instance of the demo application. For the second instance, you specify a new title so that the two applications are visually distinct. You also specify a unique service name:

```
helm install azure-samples/aks-helloworld --set title="AKS Ingress Demo" --set serviceName="ingress-demo"
```

Create an ingress route

Both applications are now running on your Kubernetes cluster. To route traffic to each application, create a Kubernetes ingress resource. The ingress resource configures the rules that route traffic to one of the two applications.

In the following example, traffic to the address `http://40.117.74.8/` is routed to the service named `aks-helloworld`. Traffic to the address `http://40.117.74.8/hello-world-two` is routed to the `ingress-demo` service.

Create a file named `hello-world-ingress.yaml` and copy in the following example YAML.

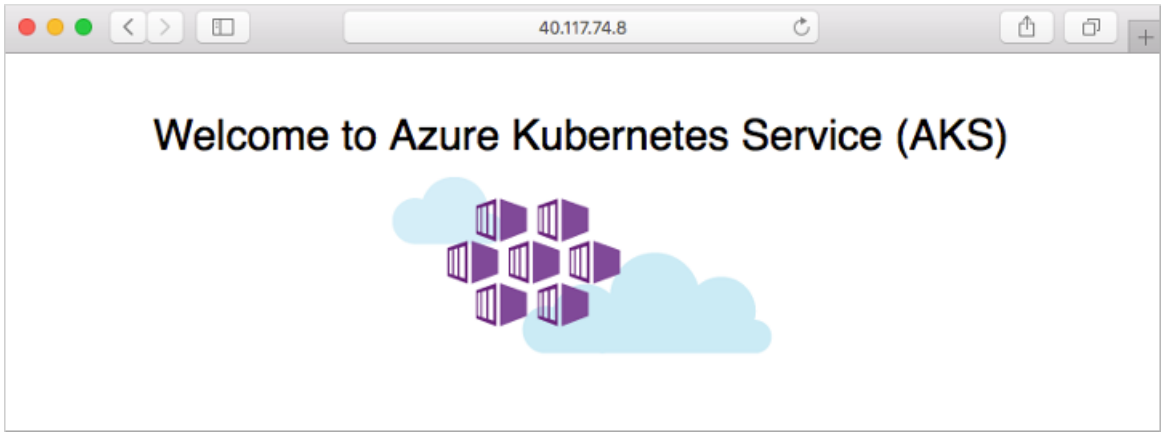
```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: hello-world-ingress
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - http:
      paths:
      - path: /
        backend:
          serviceName: aks-helloworld
          servicePort: 80
      - path: /hello-world-two
        backend:
          serviceName: ingress-demo
          servicePort: 80
```

Create the ingress resource using the `kubectl apply -f hello-world-ingress.yaml` command.

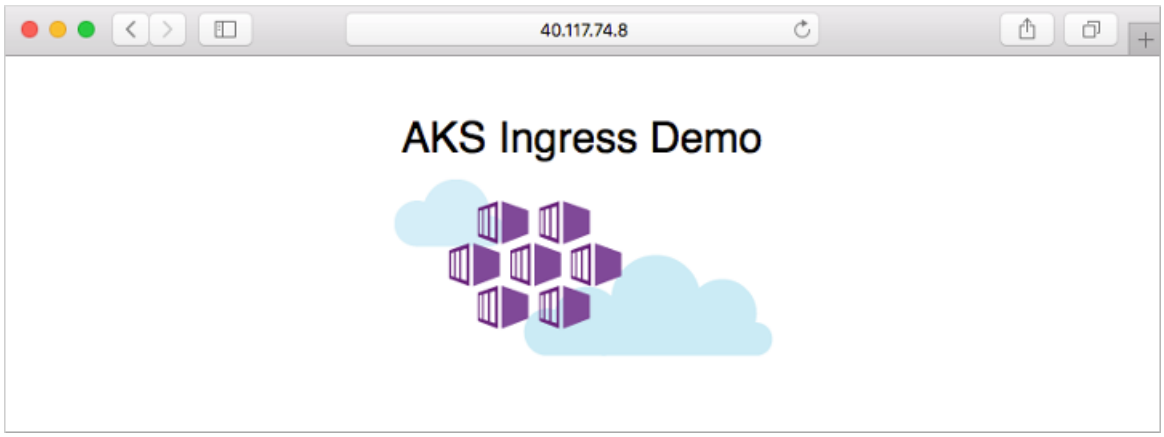
```
$ kubectl apply -f hello-world-ingress.yaml

ingress.extensions/hello-world-ingress created
```

Test the ingress controller

To test the routes for the ingress controller, browse to the two applications. Open a web browser to the IP address of your NGINX ingress controller, such as *http://40.117.74.8*. The first demo application is displayed in the web browser, as shown in the follow example:



Now add the */hello-world-two* path to the IP address, such as *http://40.117.74.8/hello-world-two*. The second demo application with the custom title is displayed:



Next steps

This article included some external components to AKS. To learn more about these components, see the following project pages:

- Helm CLI
- NGINX ingress controller

You can also:

- Enable the HTTP application routing add-on
- Create an ingress controller that uses an internal, private network and IP address
- Create an ingress controller with a dynamic public IP and configure Let's Encrypt to automatically generate TLS certificates
- Create an ingress controller with a static public IP address and configure Let's Encrypt to automatically generate TLS certificates

HTTP application routing

10/3/2018 • 5 minutes to read • Edit Online

The HTTP application routing solution makes it easy to access applications that are deployed to your Azure Kubernetes Service (AKS) cluster. When the solution's enabled, it configures an Ingress controller in your AKS cluster. As applications are deployed, the solution also creates publicly accessible DNS names for application endpoints.

When the add-on is enabled, it creates a DNS Zone in your subscription. For more information about DNS cost, see DNS pricing.

HTTP routing solution overview

The add-on deploys two components: a Kubernetes Ingress controller and an External-DNS controller.

- **Ingress controller**: The Ingress controller is exposed to the internet by using a Kubernetes service of type LoadBalancer. The Ingress controller watches and implements Kubernetes Ingress resources, which creates routes to application endpoints.
- **External-DNS controller**: Watches for Kubernetes Ingress resources and creates DNS A records in the cluster-specific DNS zone.

Deploy HTTP routing: CLI

The HTTP application routing add-on can be enabled with the Azure CLI when deploying an AKS cluster. To do so, use the az aks create command with the `--enable-addons` argument.

```
az aks create --resource-group myResourceGroup --name myAKSCluster --enable-addons http_application_routing
```

You can also enable HTTP routing on an existing AKS cluster using the az aks enable-addons command. To enable HTTP routing on an existing cluster, add the `--addons` parameter and specify *http_application_routing* as shown in the following example:

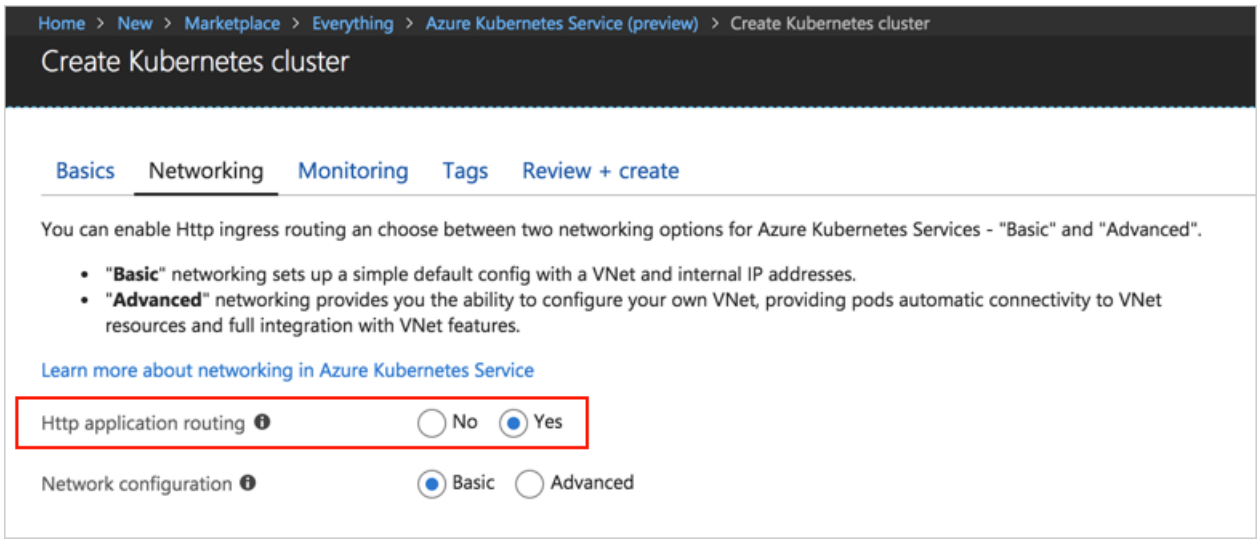
```
az aks enable-addons --resource-group myResourceGroup --name myAKSCluster --addons http_application_routing
```

After the cluster is deployed or updated, use the az aks show command to retrieve the DNS zone name. This name is needed to deploy applications to the AKS cluster.

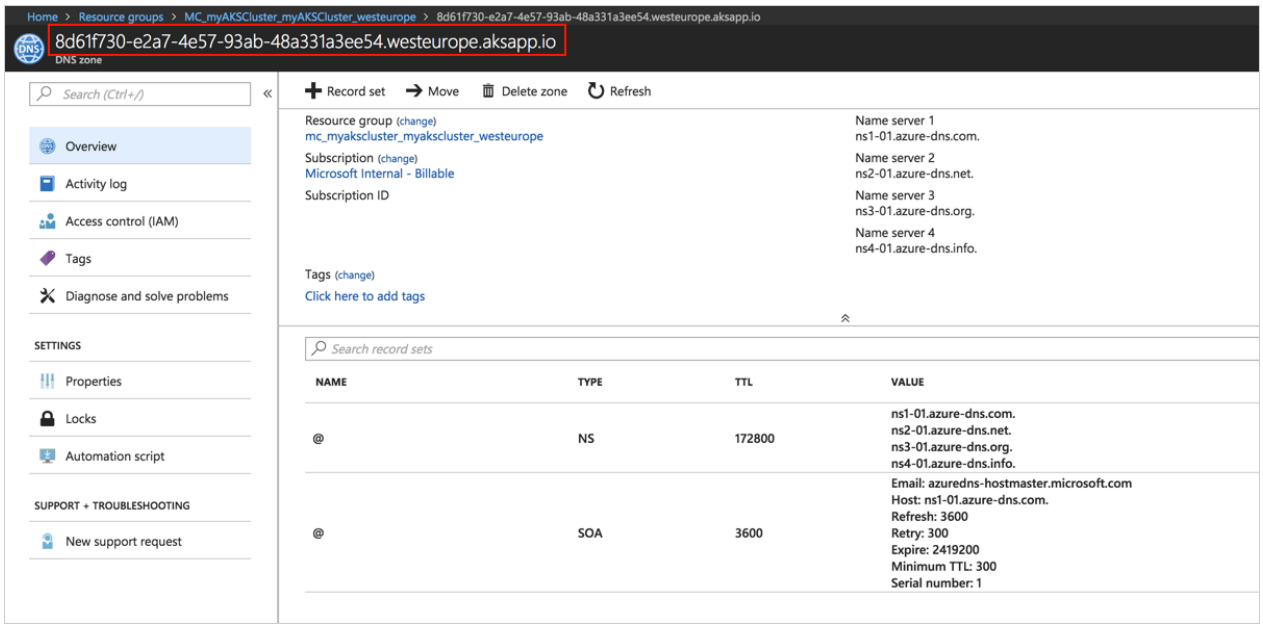
```
$ az aks show --resource-group myResourceGroup --name myAKSCluster --query
addonProfiles.httpApplicationRouting.config.HTTPApplicationRoutingZoneName -o table

Result
----------------------------------------------------
9f9c1fe7-21a1-416d-99cd-3543bb92e4c3.eastus.aksapp.io
```

Deploy HTTP routing: Portal

The HTTP application routing add-on can be enabled through the Azure portal when deploying an AKS cluster.



After the cluster is deployed, browse to the auto-created AKS resource group and select the DNS zone. Take note of the DNS zone name. This name is needed to deploy applications to the AKS cluster.



Use HTTP routing

The HTTP application routing solution may only be triggered on Ingress resources that are annotated as follows:

```
annotations:
  kubernetes.io/ingress.class: addon-http-application-routing
```

Create a file named **samples-http-application-routing.yaml** and copy in the following YAML. On line 43, update `<CLUSTER_SPECIFIC_DNS_ZONE>` with the DNS zone name collected in the previous step of this article.

```
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
        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.<CLUSTER_SPECIFIC_DNS_ZONE>
    http:
      paths:
      - backend:
          serviceName: party-clippy
          servicePort: 80
        path: /
```

Use the kubectl apply command to create the resources.

```
$ kubectl apply -f samples-http-application-routing.yaml

deployment "party-clippy" created
service "party-clippy" created
ingress "party-clippy" created
```

Use cURL or a browser to navigate to the hostname specified in the host section of the samples-http-application-routing.yaml file. The application can take up to one minute before it's available via the internet.

```
$ curl party-clippy.471756a6-e744-4aa0-aa01-89c4d162a7a7.canadaeast.aksapp.io


 _____________________________________
/ It looks like you're building a \
\ microservice.                     /
 -------------------------------------
 \
  \
     __
    /  \
    |  |
    @  @
    |  |
    || |/
    || ||
    |\_/|
    \___/
```

Remove HTTP routing

The HTTP routing solution can be removed using the Azure CLI. To do so run the following command, substituting your AKS cluster and resource group name.

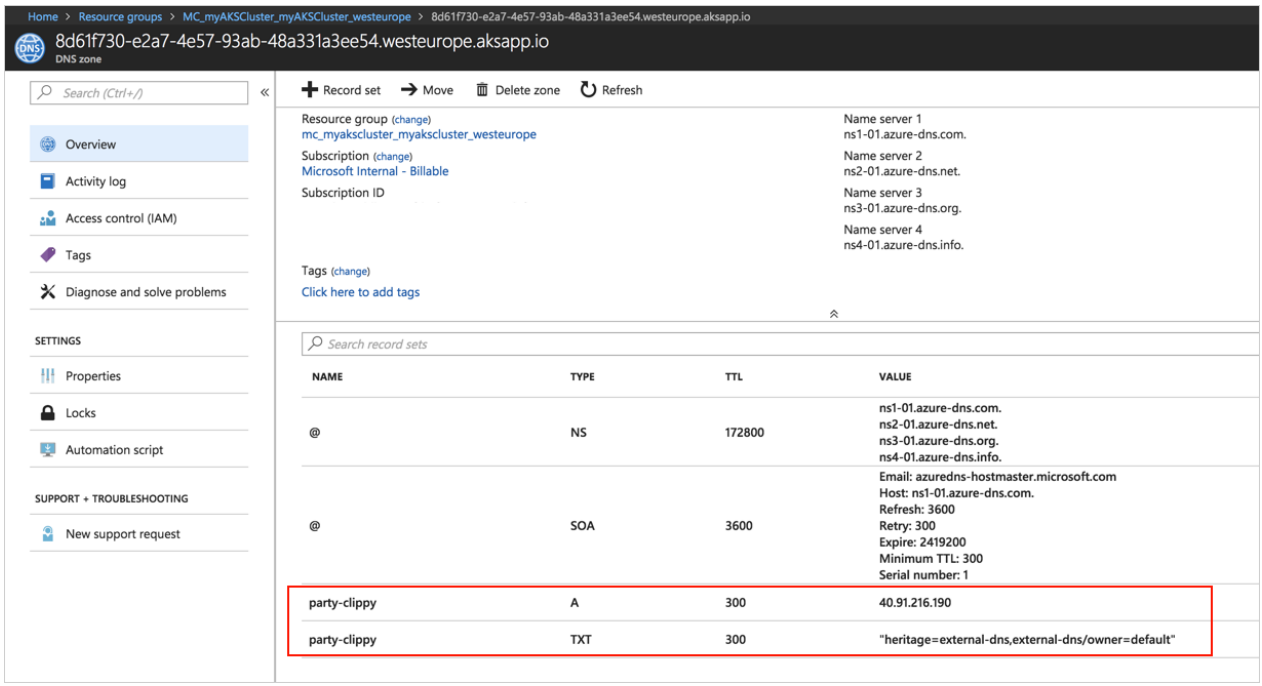
```
az aks disable-addons --addons http_application_routing --name myAKSCluster --resource-group myResourceGroup
--no-wait
```

Troubleshoot

Use the kubectl logs command to view the application logs for the External-DNS application. The logs should confirm that an A and TXT DNS record were created successfully.

```
$ kubectl logs -f deploy/addon-http-application-routing-external-dns -n kube-system

time="2018-04-26T20:36:19Z" level=info msg="Updating A record named 'party-clippy' to '52.242.28.189' for
Azure DNS zone '471756a6-e744-4aa0-aa01-89c4d162a7a7.canadaeast.aksapp.io'."
time="2018-04-26T20:36:21Z" level=info msg="Updating TXT record named 'party-clippy' to '"heritage=external-
dns,external-dns/owner=default"' for Azure DNS zone '471756a6-e744-4aa0-aa01-
89c4d162a7a7.canadaeast.aksapp.io'."
```

These records can also be seen on the DNS zone resource in the Azure portal.



Use the kubectl logs command to view the application logs for the Nginx Ingress controller. The logs should confirm the `CREATE` of an Ingress resource and the reload of the controller. All HTTP activity is logged.

```
$ kubectl logs -f deploy/addon-http-application-routing-nginx-ingress-controller -n kube-system

-------------------------------------------------------------------------------
NGINX Ingress controller
  Release:    0.13.0
  Build:      git-4bc943a
  Repository: https://github.com/kubernetes/ingress-nginx
-------------------------------------------------------------------------------

I0426 20:30:12.212936       9 flags.go:162] Watching for ingress class: addon-http-application-routing
W0426 20:30:12.213041       9 flags.go:165] only Ingress with class "addon-http-application-routing" will be
processed by this ingress controller
W0426 20:30:12.213505       9 client_config.go:533] Neither --kubeconfig nor --master was specified.  Using
the inClusterConfig.  This might not work.
I0426 20:30:12.213752       9 main.go:181] Creating API client for https://10.0.0.1:443
I0426 20:30:12.287928       9 main.go:225] Running in Kubernetes Cluster version v1.8 (v1.8.11) - git (clean)
commit 1df6a8381669a6c753f79cb31ca2e3d57ee7c8a3 - platform linux/amd64
I0426 20:30:12.290988       9 main.go:84] validated kube-system/addon-http-application-routing-default-http-
backend as the default backend
I0426 20:30:12.294314       9 main.go:105] service kube-system/addon-http-application-routing-nginx-ingress
validated as source of Ingress status
I0426 20:30:12.426443       9 stat_collector.go:77] starting new nginx stats collector for Ingress controller
running in namespace  (class addon-http-application-routing)
I0426 20:30:12.426509       9 stat_collector.go:78] collector extracting information from port 18080
I0426 20:30:12.448779       9 nginx.go:281] starting Ingress controller
I0426 20:30:12.463585       9 event.go:218] Event(v1.ObjectReference{Kind:"ConfigMap", Namespace:"kube-
system", Name:"addon-http-application-routing-nginx-configuration", UID:"2588536c-4990-11e8-a5e1-
0a58ac1f0ef2", APIVersion:"v1", ResourceVersion:"559", FieldPath:""}): type: 'Normal' reason: 'CREATE'
ConfigMap kube-system/addon-http-application-routing-nginx-configuration
I0426 20:30:12.466945       9 event.go:218] Event(v1.ObjectReference{Kind:"ConfigMap", Namespace:"kube-
system", Name:"addon-http-application-routing-tcp-services", UID:"258ca065-4990-11e8-a5e1-0a58ac1f0ef2",
APIVersion:"v1", ResourceVersion:"561", FieldPath:""}): type: 'Normal' reason: 'CREATE' ConfigMap kube-
system/addon-http-application-routing-tcp-services
I0426 20:30:12.467053       9 event.go:218] Event(v1.ObjectReference{Kind:"ConfigMap", Namespace:"kube-
system", Name:"addon-http-application-routing-udp-services", UID:"259023bc-4990-11e8-a5e1-0a58ac1f0ef2",
APIVersion:"v1", ResourceVersion:"562", FieldPath:""}): type: 'Normal' reason: 'CREATE' ConfigMap kube-
system/addon-http-application-routing-udp-services
I0426 20:30:13.649195       9 nginx.go:302] starting NGINX process...
I0426 20:30:13.649347       9 leaderelection.go:175] attempting to acquire leader lease  kube-system/ingress-
controller-leader-addon-http-application-routing...
I0426 20:30:13.649776       9 controller.go:170] backend reload required
I0426 20:30:13.649800       9 stat_collector.go:34] changing prometheus collector from  to default
I0426 20:30:13.662191       9 leaderelection.go:184] successfully acquired lease kube-system/ingress-
controller-leader-addon-http-application-routing
I0426 20:30:13.662292       9 status.go:196] new leader elected: addon-http-application-routing-nginx-
ingress-controller-5cxntd6
I0426 20:30:13.763362       9 controller.go:179] ingress backend successfully reloaded...
I0426 21:51:55.249327       9 event.go:218] Event(v1.ObjectReference{Kind:"Ingress", Namespace:"default",
Name:"party-clippy", UID:"092c9599-499c-11e8-a5e1-0a58ac1f0ef2", APIVersion:"extensions",
ResourceVersion:"7346", FieldPath:""}): type: 'Normal' reason: 'CREATE' Ingress default/party-clippy
W0426 21:51:57.908771       9 controller.go:775] service default/party-clippy does not have any active
endpoints
I0426 21:51:57.908951       9 controller.go:170] backend reload required
I0426 21:51:58.042932       9 controller.go:179] ingress backend successfully reloaded...
167.220.24.46 - [167.220.24.46] - - [26/Apr/2018:21:53:20 +0000] "GET / HTTP/1.1" 200 234 "" "Mozilla/5.0
(compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0)" 197 0.001 [default-party-clippy-80] 10.244.0.13:8080 234
0.004 200
```

Clean up

Remove the associated Kubernetes objects created in this article.

```
$ kubectl delete -f samples-http-application-routing.yaml

deployment "party-clippy" deleted
service "party-clippy" deleted
ingress "party-clippy" deleted
```

Next steps

For information on how to install an HTTPS-secured Ingress controller in AKS, see HTTPS Ingress on Azure Kubernetes Service (AKS).

Create an ingress controller to an internal virtual network in Azure Kubernetes Service (AKS)

9/10/2018 • 5 minutes to read • Edit Online

An ingress controller is a piece of software that provides reverse proxy, configurable traffic routing, and TLS termination for Kubernetes services. Kubernetes ingress resources are used to configure the ingress rules and routes for individual Kubernetes services. Using an ingress controller and ingress rules, a single IP address can be used to route traffic to multiple services in a Kubernetes cluster.

This article shows you how to deploy the NGINX ingress controller in an Azure Kubernetes Service (AKS) cluster. The ingress controller is configured on an internal, private virtual network and IP address. No external access is allowed. Two applications are then run in the AKS cluster, each of which is accessible over the single IP address.

You can also:

- Create a basic ingress controller with external network connectivity
- Enable the HTTP application routing add-on
- Create an ingress controller with a dynamic public IP and configure Let's Encrypt to automatically generate TLS certificates
- Create an ingress controller with a static public IP address and configure Let's Encrypt to automatically generate TLS certificates

Before you begin

This article uses Helm to install the NGINX ingress controller, cert-manager, and a sample web app. You need to have Helm initialized within your AKS cluster and using a service account for Tiller. For more information on configuring and using Helm, see Install applications with Helm in Azure Kubernetes Service (AKS).

This article also requires that you are running the Azure CLI version 2.0.41 or later. Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI.

Create an ingress controller

By default, an NGINX ingress controller is created with a dynamic public IP address assignment. A common configuration requirement is to use an internal, private network and IP address. This approach allows you to restrict access to your services to internal users, with no external access.

Create a file named *internal-ingress.yaml* using the following example manifest file. This example assigns *10.240.0.42* to the *loadBalancerIP* resource. Provide your own internal IP address for use with the ingress controller. Make sure that this IP address is not already in use within your virtual network.

```
controller:
  service:
    loadBalancerIP: 10.240.0.42
    annotations:
      service.beta.kubernetes.io/azure-load-balancer-internal: "true"
```

Now deploy the *nginx-ingress* chart with Helm. To use the manifest file created in the previous step, add the `-f internal-ingress.yaml` parameter:

TIP

The following example installs the ingress controller in the `kube-system` namespace. You can specify a different namespace for your own environment if desired. If your AKS cluster is not RBAC enabled, add `--set rbac.create=false` to the commands.

```
helm install stable/nginx-ingress --namespace kube-system -f internal-ingress.yaml
```

When the Kubernetes load balancer service is created for the NGINX ingress controller, your internal IP address is assigned, as shown in the following example output:

```
$ kubectl get service -l app=nginx-ingress --namespace kube-system

NAME                                            TYPE           CLUSTER-IP    EXTERNAL-IP   PORT(S)
AGE
alternating-coral-nginx-ingress-controller      LoadBalancer   10.0.97.109   10.240.0.42
80:31507/TCP,443:30707/TCP   1m
alternating-coral-nginx-ingress-default-backend   ClusterIP      10.0.134.66   <none>        80/TCP
1m
```

No ingress rules have been created yet, so the NGINX ingress controller's default 404 page is displayed if you browse to the internal IP address. Ingress rules are configured in the following steps.

Run demo applications

To see the ingress controller in action, let's run two demo applications in your AKS cluster. In this example, Helm is used to deploy two instances of a simple 'Hello world' application.

Before you can install the sample Helm charts, add the Azure samples repository to your Helm environment as follows:

```
helm repo add azure-samples https://azure-samples.github.io/helm-charts/
```

Create the first demo application from a Helm chart with the following command:

```
helm install azure-samples/aks-helloworld
```

Now install a second instance of the demo application. For the second instance, you specify a new title so that the two applications are visually distinct. You also specify a unique service name:

```
helm install azure-samples/aks-helloworld --set title="AKS Ingress Demo" --set serviceName="ingress-demo"
```

Create an ingress route

Both applications are now running on your Kubernetes cluster. To route traffic to each application, create a Kubernetes ingress resource. The ingress resource configures the rules that route traffic to one of the two applications.

In the following example, traffic to the address `http://10.240.0.42/` is routed to the service named `aks-helloworld`. Traffic to the address `http://10.240.0.42/hello-world-two` is routed to the `ingress-demo` service.

Create a file named `hello-world-ingress.yaml` and copy in the following example YAML.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: hello-world-ingress
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - http:
      paths:
      - path: /
        backend:
          serviceName: aks-helloworld
          servicePort: 80
      - path: /hello-world-two
        backend:
          serviceName: ingress-demo
          servicePort: 80
```

Create the ingress resource using the `kubectl apply -f hello-world-ingress.yaml` command.

```
$ kubectl apply -f hello-world-ingress.yaml

ingress.extensions/hello-world-ingress created
```

Test the ingress controller

To test the routes for the ingress controller, browse to the two applications with a web client. If needed, you can quickly test this internal-only functionality from a pod on the AKS cluster. Create a test pod and attach a terminal session to it:

```
kubectl run -it --rm aks-ingress-test --image=debian
```

Install `curl` in the pod using `apt-get`:

```
apt-get update && apt-get install -y curl
```

Now access the address of your Kubernetes ingress controller using `curl`, such as *http://10.240.0.42*. Provide your own internal IP address specified when you deployed the ingress controller in the first step of this article.

```
curl -L http://10.240.0.42
```

No additional path was provided with the address, so the ingress controller defaults to the */* route. The first demo application is returned, as shown in the following condensed example output:

```
$ curl -L 10.240.0.42

<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <link rel="stylesheet" type="text/css" href="/static/default.css">
    <title>Welcome to Azure Kubernetes Service (AKS)</title>
[...]
```

Now add */hello-world-two* path to the address, such as *http://10.240.0.42/hello-world-two*. The second demo application with the custom title is returned, as shown in the following condensed example output:

```
$ curl -L -k http://10.240.0.42/hello-world-two

<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <link rel="stylesheet" type="text/css" href="/static/default.css">
    <title>AKS Ingress Demo</title>
[...]
```

Next steps

This article included some external components to AKS. To learn more about these components, see the following project pages:

- Helm CLI
- NGINX ingress controller

You can also:

- Create a basic ingress controller with external network connectivity
- Enable the HTTP application routing add-on
- Create an ingress controller with a dynamic public IP and configure Let's Encrypt to automatically generate TLS certificates
- Create an ingress controller with a static public IP address and configure Let's Encrypt to automatically generate TLS certificates

Create an HTTPS ingress controller on Azure Kubernetes Service (AKS)

9/10/2018 • 7 minutes to read • Edit Online

An ingress controller is a piece of software that provides reverse proxy, configurable traffic routing, and TLS termination for Kubernetes services. Kubernetes ingress resources are used to configure the ingress rules and routes for individual Kubernetes services. Using an ingress controller and ingress rules, a single IP address can be used to route traffic to multiple services in a Kubernetes cluster.

This article shows you how to deploy the NGINX ingress controller in an Azure Kubernetes Service (AKS) cluster. The cert-manager project is used to automatically generate and configure Let's Encrypt certificates. Finally, two applications are run in the AKS cluster, each of which is accessible over a single IP address.

You can also:

- Create a basic ingress controller with external network connectivity
- Enable the HTTP application routing add-on
- Create an ingress controller that uses an internal, private network and IP address
- Create an ingress controller with a static public IP address and configure Let's Encrypt to automatically generate TLS certificates

Before you begin

This article uses Helm to install the NGINX ingress controller, cert-manager, and a sample web app. You need to have Helm initialized within your AKS cluster and using a service account for Tiller. Make sure that you are using the latest release of Helm. For upgrade instructions, see the Helm install docs. For more information on configuring and using Helm, see Install applications with Helm in Azure Kubernetes Service (AKS).

This article also requires that you are running the Azure CLI version 2.0.41 or later. Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI.

Create an ingress controller

To create the ingress controller, use `Helm` to install *nginx-ingress*.

> **TIP**
> The following example installs the ingress controller in the `kube-system` namespace. You can specify a different namespace for your own environment if desired. If your AKS cluster is not RBAC enabled, add `--set rbac.create=false` to the commands.

```
helm install stable/nginx-ingress --namespace kube-system
```

During the installation, an Azure public IP address is created for the ingress controller. This public IP address is static for the life-span of the ingress controller. If you delete the ingress controller, the public IP address assignment is lost. If you then create an additional ingress controller, a new public IP address is assigned. If you wish to retain the use of the public IP address, you can instead create an ingress controller with a static public IP address.

To get the public IP address, use the `kubectl get service` command. It takes a few minutes for the IP address to be assigned to the service.

```
$ kubectl get service -l app=nginx-ingress --namespace kube-system

NAME                                     TYPE           CLUSTER-IP     EXTERNAL-IP     PORT(S)
AGE
eager-crab-nginx-ingress-controller      LoadBalancer   10.0.182.160   51.145.155.210
80:30920/TCP,443:30426/TCP   20m
eager-crab-nginx-ingress-default-backend   ClusterIP      10.0.255.77    <none>          80/TCP
20m
```

No ingress rules have been created yet. If you browse to the public IP address, the NGINX ingress controller's default 404 page is displayed.

Configure a DNS name

For the HTTPS certificates to work correctly, configure an FQDN for the ingress controller IP address. Update the following script with the IP address of your ingress controller and a unique name that you would like to use for the FQDN:

```
#!/bin/bash

# Public IP address of your ingress controller
IP="51.145.155.210"

# Name to associate with public IP address
DNSNAME="demo-aks-ingress"

# Get the resource-id of the public ip
PUBLICIPID=$(az network public-ip list --query "[?ipAddress!=null]|[?contains(ipAddress, '$IP')].[id]" --
output tsv)

# Update public ip address with DNS name
az network public-ip update --ids $PUBLICIPID --dns-name $DNSNAME
```

The ingress controller is now accessible through the FQDN.

Install cert-manager

The NGINX ingress controller supports TLS termination. There are several ways to retrieve and configure certificates for HTTPS. This article demonstrates using cert-manager, which provides automatic Lets Encrypt certificate generation and management functionality.

> **NOTE**
> This article uses the `staging` environment for Let's Encrypt. In production deployments, use `letsencrypt-prod` and `https://acme-v02.api.letsencrypt.org/directory` in the resource definitions and when installing the Helm chart.

To install the cert-manager controller in an RBAC-enabled cluster, use the following `helm install` command:

```
helm install stable/cert-manager --set ingressShim.defaultIssuerName=letsencrypt-staging --set
ingressShim.defaultIssuerKind=ClusterIssuer
```

If your cluster is not RBAC enabled, instead use the following command:

```
helm install stable/cert-manager \
  --set ingressShim.defaultIssuerName=letsencrypt-staging \
  --set ingressShim.defaultIssuerKind=ClusterIssuer \
  --set rbac.create=false \
  --set serviceAccount.create=false
```

For more information on cert-manager configuration, see the cert-manager project.

Create a CA cluster issuer

Before certificates can be issued, cert-manager requires an Issuer or ClusterIssuer resource. These Kubernetes resources are identical in functionality, however `Issuer` works in a single namespace, and `ClusterIssuer` works across all namespaces. For more information, see the cert-manager issuer documentation.

Create a cluster issuer, such as `cluster-issuer.yaml`, using the following example manifest. Update the email address with a valid address from your organization:

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: ClusterIssuer
metadata:
  name: letsencrypt-staging
spec:
  acme:
    server: https://acme-staging-v02.api.letsencrypt.org/directory
    email: user@contoso.com
    privateKeySecretRef:
      name: letsencrypt-staging
    http01: {}
```

To create the issuer, use the `kubectl apply -f cluster-issuer.yaml` command.

```
$ kubectl apply -f cluster-issuer.yaml

clusterissuer.certmanager.k8s.io/letsencrypt-staging created
```

Create a certificate object

Next, a certificate resource must be created. The certificate resource defines the desired X.509 certificate. For more information, see cert-manager certificates.

Create the certificate resource, such as `certificates.yaml`, with the following example manifest. Update the *dnsNames* and *domains* to the DNS name you created in a previous step. If you use an internal-only ingress controller, specify the internal DNS name for your service.

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Certificate
metadata:
  name: tls-secret
spec:
  secretName: tls-secret
  dnsNames:
  - demo-aks-ingress.eastus.cloudapp.azure.com
  acme:
    config:
    - http01:
        ingressClass: nginx
      domains:
      - demo-aks-ingress.eastus.cloudapp.azure.com
  issuerRef:
    name: letsencrypt-staging
    kind: ClusterIssuer
```

To create the certificate resource, use the `kubectl apply -f certificates.yaml` command.

```
$ kubectl apply -f certificates.yaml

certificate.certmanager.k8s.io/tls-secret created
```

Run demo applications

An ingress controller and a certificate management solution have been configured. Now let's run two demo applications in your AKS cluster. In this example, Helm is used to deploy two instances of a simple 'Hello world' application.

Before you can install the sample Helm charts, add the Azure samples repository to your Helm environment as follows:

```
helm repo add azure-samples https://azure-samples.github.io/helm-charts/
```

Create the first demo application from a Helm chart with the following command:

```
helm install azure-samples/aks-helloworld
```

Now install a second instance of the demo application. For the second instance, you specify a new title so that the two applications are visually distinct. You also specify a unique service name:

```
helm install azure-samples/aks-helloworld --set title="AKS Ingress Demo" --set serviceName="ingress-demo"
```

Create an ingress route

Both applications are now running on your Kubernetes cluster, however they're configured with a service of type `ClusterIP`. As such, the applications aren't accessible from the internet. To make them publicly available, create a Kubernetes ingress resource. The ingress resource configures the rules that route traffic to one of the two applications.

In the following example, traffic to the address `https://demo-aks-ingress.eastus.cloudapp.azure.com/` is routed to the service named `aks-helloworld`. Traffic to the address `https://demo-aks-ingress.eastus.cloudapp.azure.com/hello-world-two` is routed to the `ingress-demo` service.

Update the *hosts* and *host* to the DNS name you created in a previous step.

Create a file named `hello-world-ingress.yaml` and copy in the following example YAML.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: hello-world-ingress
  annotations:
    kubernetes.io/ingress.class: nginx
    certmanager.k8s.io/cluster-issuer: letsencrypt-staging
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  tls:
  - hosts:
    - demo-aks-ingress.eastus.cloudapp.azure.com
    secretName: tls-secret
  rules:
  - host: demo-aks-ingress.eastus.cloudapp.azure.com
    http:
      paths:
      - path: /
        backend:
          serviceName: aks-helloworld
          servicePort: 80
      - path: /hello-world-two
        backend:
          serviceName: ingress-demo
          servicePort: 80
```

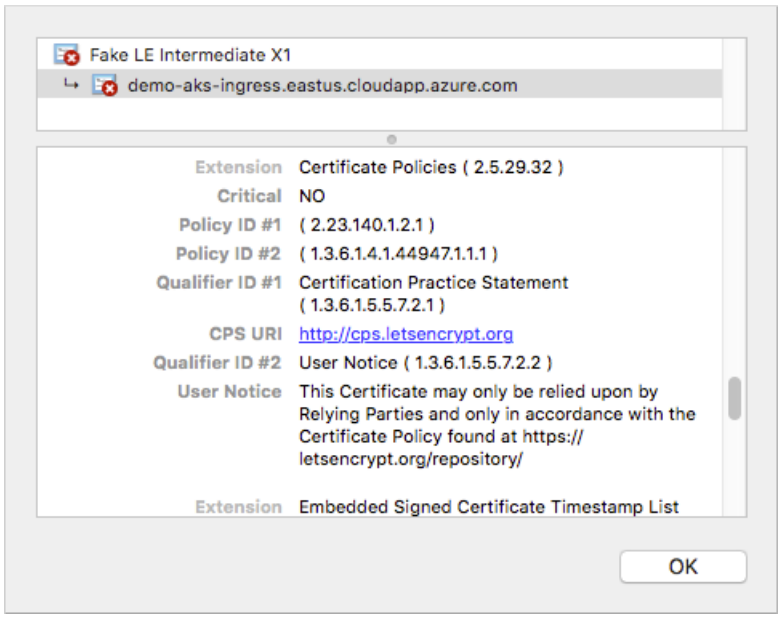
Create the ingress resource using the `kubectl apply -f hello-world-ingress.yaml` command.

```
$ kubectl apply -f hello-world-ingress.yaml

ingress.extensions/hello-world-ingress created
```

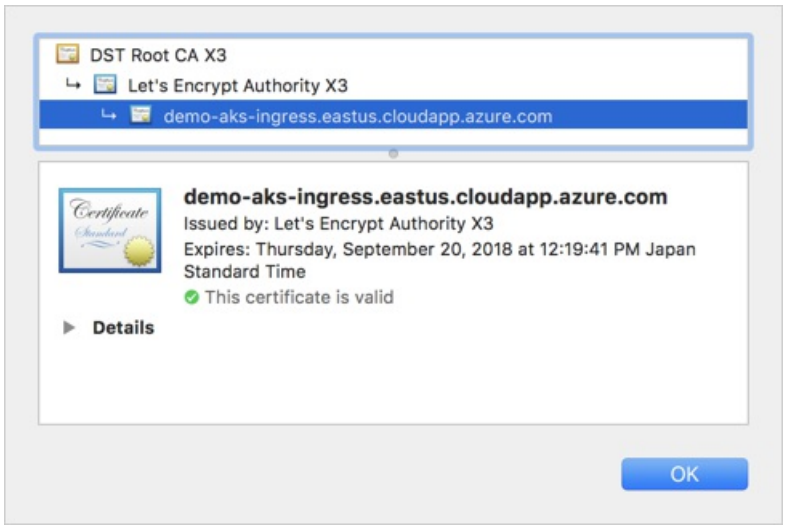
Test the ingress configuration

Open a web browser to the FQDN of your Kubernetes ingress controller, such as *https://demo-aks-ingress.eastus.cloudapp.azure.com*.

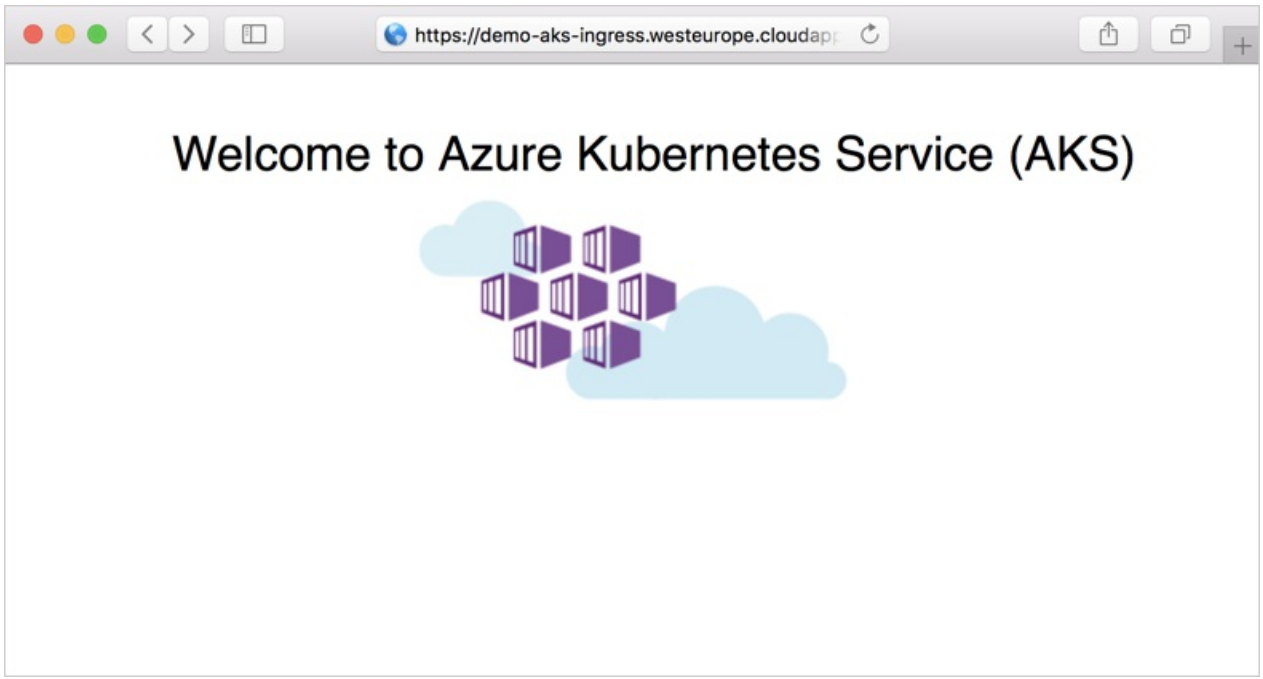
As these examples use `letsencrypt-staging`, the issued SSL certificate is not trusted by the browser. Accept the warning prompt to continue to your application. The certificate information shows this *Fake LE Intermediate X1* certificate is issued by Let's Encrypt. This fake certificate indicates `cert-manager` processed the request correctly and received a certificate from the provider:



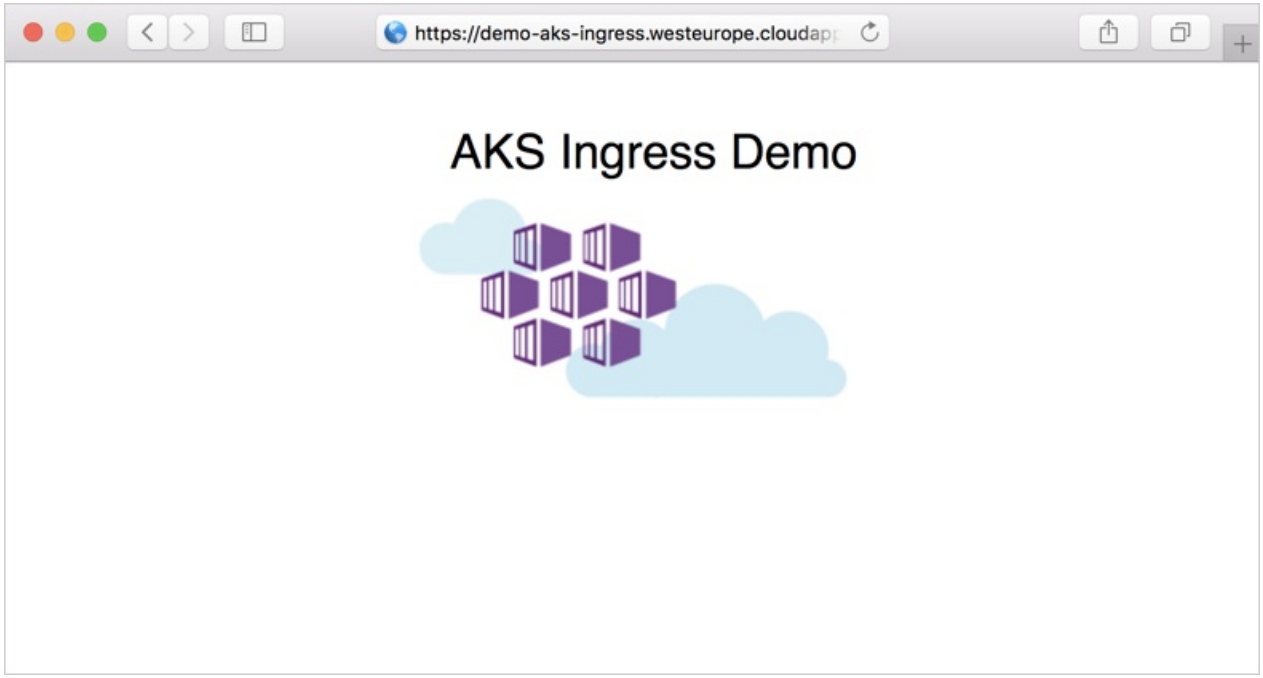
When you change Let's Encrypt to use `prod` rather than `staging`, a trusted certificate issued by Let's Encrypt is used, as shown in the following example:



The demo application is shown in the web browser:



Now add the */hello-world-two* path to the FQDN, such as *https://demo-aks-ingress.eastus.cloudapp.azure.com/hello-world-two*. The second demo application with the custom title is shown:



Next steps

This article included some external components to AKS. To learn more about these components, see the following project pages:

- Helm CLI
- NGINX ingress controller
- cert-manager

You can also:

- Create a basic ingress controller with external network connectivity
- Enable the HTTP application routing add-on
- Create an ingress controller that uses an internal, private network and IP address
- Create an ingress controller with a static public IP address and configure Let's Encrypt to automatically generate TLS certificates

Create an ingress controller with a static public IP address in Azure Kubernetes Service (AKS)

9/24/2018 • 7 minutes to read • Edit Online

An ingress controller is a piece of software that provides reverse proxy, configurable traffic routing, and TLS termination for Kubernetes services. Kubernetes ingress resources are used to configure the ingress rules and routes for individual Kubernetes services. Using an ingress controller and ingress rules, a single IP address can be used to route traffic to multiple services in a Kubernetes cluster.

This article shows you how to deploy the NGINX ingress controller in an Azure Kubernetes Service (AKS) cluster. The ingress controller is configured with a static public IP address. The cert-manager project is used to automatically generate and configure Let's Encrypt certificates. Finally, two applications are run in the AKS cluster, each of which is accessible over a single IP address.

You can also:

- Create a basic ingress controller with external network connectivity
- Enable the HTTP application routing add-on
- Create an ingress controller that uses an internal, private network and IP address
- Create an ingress controller with a dynamic public IP and configure Let's Encrypt to automatically generate TLS certificates

Before you begin

This article uses Helm to install the NGINX ingress controller, cert-manager, and a sample web app. You need to have Helm initialized within your AKS cluster and using a service account for Tiller. Make sure that you are using the latest release of Helm. Make sure that you are using the latest release of Helm. For upgrade instructions, see the Helm install docs. For more information on configuring and using Helm, see Install applications with Helm in Azure Kubernetes Service (AKS).

This article also requires that you are running the Azure CLI version 2.0.41 or later. Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI.

Create an ingress controller

By default, an NGINX ingress controller is created with a new public IP address assignment. This public IP address is only static for the life-span of the ingress controller, and is lost if the controller is deleted and re-created. A common configuration requirement is to provide the NGINX ingress controller an existing static public IP address. The static public IP address remains if the ingress controller is deleted. This approach allows you to use existing DNS records and network configurations in a consistent manner throughout the lifecycle of your applications.

If you need to create a static public IP address, first get the resource group name of the AKS cluster with the az aks show command:

```
az aks show --resource-group myResourceGroup --name myAKSCluster --query nodeResourceGroup -o tsv
```

Next, create a public IP address with the *static* allocation method using the az network public-ip create command. The following example creates a public IP address named *myAKSPublicIP* in the AKS cluster

resource group obtained in the previous step:

```
az network public-ip create --resource-group MC_myResourceGroup_myAKSCluster_eastus --name myAKSPublicIP --
allocation-method static
```

Now deploy the *nginx-ingress* chart with Helm. Add the `--set controller.service.loadBalancerIP` parameter, and specify your own public IP address created in the previous step.

TIP

The following examples install the ingress controller and certificates in the `kube-system` namespace. You can specify a different namespace for your own environment if desired. Also, if your AKS cluster is not RBAC enabled, add `--set rbac.create=false` to the commands.

```
helm install stable/nginx-ingress --namespace kube-system --set
controller.service.loadBalancerIP="40.121.63.72"
```

When the Kubernetes load balancer service is created for the NGINX ingress controller, your static IP address is assigned, as shown in the following example output:

```
$ kubectl get service -l app=nginx-ingress --namespace kube-system

NAME                                     TYPE           CLUSTER-IP    EXTERNAL-IP    PORT(S)
AGE
dinky-panda-nginx-ingress-controller     LoadBalancer   10.0.232.56   40.121.63.72
80:31978/TCP,443:32037/TCP   3m
dinky-panda-nginx-ingress-default-backend   ClusterIP      10.0.95.248   <none>         80/TCP
3m
```

No ingress rules have been created yet, so the NGINX ingress controller's default 404 page is displayed if you browse to the public IP address. Ingress rules are configured in the following steps.

Configure a DNS name

For the HTTPS certificates to work correctly, configure an FQDN for the ingress controller IP address. Update the following script with the IP address of your ingress controller and a unique name that you would like to use for the FQDN:

```
#!/bin/bash

# Public IP address of your ingress controller
IP="51.145.155.210"

# Name to associate with public IP address
DNSNAME="demo-aks-ingress"

# Get the resource-id of the public ip
PUBLICIPID=$(az network public-ip list --query "[?ipAddress!=null]|[?contains(ipAddress, '$IP')].[id]" --
output tsv)

# Update public ip address with DNS name
az network public-ip update --ids $PUBLICIPID --dns-name $DNSNAME
```

The ingress controller is now accessible through the FQDN.

Install cert-manager

The NGINX ingress controller supports TLS termination. There are several ways to retrieve and configure certificates for HTTPS. This article demonstrates using cert-manager, which provides automatic Lets Encrypt certificate generation and management functionality.

> **NOTE**
> This article uses the `staging` environment for Let's Encrypt. In production deployments, use `letsencrypt-prod` and `https://acme-v02.api.letsencrypt.org/directory` in the resource definitions and when installing the Helm chart.

To install the cert-manager controller in an RBAC-enabled cluster, use the following `helm install` command. Again, if desired, change `--namespace` to something other than *kube-system*:

```
helm install stable/cert-manager \
  --namespace kube-system \
  --set ingressShim.defaultIssuerName=letsencrypt-staging \
  --set ingressShim.defaultIssuerKind=ClusterIssuer
```

If your cluster is not RBAC enabled, instead use the following command:

```
helm install stable/cert-manager \
  --namespace kube-system \
  --set ingressShim.defaultIssuerName=letsencrypt-staging \
  --set ingressShim.defaultIssuerKind=ClusterIssuer \
  --set rbac.create=false \
  --set serviceAccount.create=false
```

For more information on cert-manager configuration, see the cert-manager project.

Create a CA cluster issuer

Before certificates can be issued, cert-manager requires an Issuer or ClusterIssuer resource. These Kubernetes resources are identical in functionality, however `Issuer` works in a single namespace, and `ClusterIssuer` works across all namespaces. For more information, see the cert-manager issuer documentation.

Create a cluster issuer, such as `cluster-issuer.yaml`, using the following example manifest. Update the email address with a valid address from your organization:

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: ClusterIssuer
metadata:
  name: letsencrypt-staging
spec:
  acme:
    server: https://acme-staging-v02.api.letsencrypt.org/directory
    email: user@contoso.com
    privateKeySecretRef:
      name: letsencrypt-staging
    http01: {}
```

To create the issuer, use the `kubectl apply -f cluster-issuer.yaml` command.

```
$ kubectl apply -f cluster-issuer.yaml

clusterissuer.certmanager.k8s.io/letsencrypt-staging created
```

Create a certificate object

Next, a certificate resource must be created. The certificate resource defines the desired X.509 certificate. For more information, see cert-manager certificates.

Create the certificate resource, such as `certificates.yaml`, with the following example manifest. Update the *dnsNames* and *domains* to the DNS name you created in a previous step. If you use an internal-only ingress controller, specify the internal DNS name for your service.

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Certificate
metadata:
  name: tls-secret
spec:
  secretName: tls-secret
  dnsNames:
  - demo-aks-ingress.eastus.cloudapp.azure.com
  acme:
    config:
    - http01:
        ingressClass: nginx
      domains:
      - demo-aks-ingress.eastus.cloudapp.azure.com
  issuerRef:
    name: letsencrypt-staging
    kind: ClusterIssuer
```

To create the certificate resource, use the `kubectl apply -f certificates.yaml` command.

```
$ kubectl apply -f certificates.yaml

certificate.certmanager.k8s.io/tls-secret created
```

Run demo applications

An ingress controller and a certificate management solution have been configured. Now let's run two demo applications in your AKS cluster. In this example, Helm is used to deploy two instances of a simple 'Hello world' application.

Before you can install the sample Helm charts, add the Azure samples repository to your Helm environment as follows:

```
helm repo add azure-samples https://azure-samples.github.io/helm-charts/
```

Create the first demo application from a Helm chart with the following command:

```
helm install azure-samples/aks-helloworld
```

Now install a second instance of the demo application. For the second instance, you specify a new title so that the two applications are visually distinct. You also specify a unique service name:

```
helm install azure-samples/aks-helloworld --set title="AKS Ingress Demo" --set serviceName="ingress-demo"
```

Create an ingress route

Both applications are now running on your Kubernetes cluster, however they're configured with a service of type `ClusterIP`. As such, the applications aren't accessible from the internet. To make them publicly available, create a Kubernetes ingress resource. The ingress resource configures the rules that route traffic to one of the two applications.

In the following example, traffic to the address `https://demo-aks-ingress.eastus.cloudapp.azure.com/` is routed to the service named `aks-helloworld`. Traffic to the address `https://demo-aks-ingress.eastus.cloudapp.azure.com/hello-world-two` is routed to the `ingress-demo` service. Update the *hosts* and *host* to the DNS name you created in a previous step.

Create a file named `hello-world-ingress.yaml` and copy in the following example YAML.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: hello-world-ingress
  annotations:
    kubernetes.io/ingress.class: nginx
    certmanager.k8s.io/cluster-issuer: letsencrypt-staging
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  tls:
  - hosts:
    - demo-aks-ingress.eastus.cloudapp.azure.com
    secretName: tls-secret
  rules:
  - host: demo-aks-ingress.eastus.cloudapp.azure.com
    http:
      paths:
      - path: /
        backend:
          serviceName: aks-helloworld
          servicePort: 80
      - path: /hello-world-two
        backend:
          serviceName: ingress-demo
          servicePort: 80
```

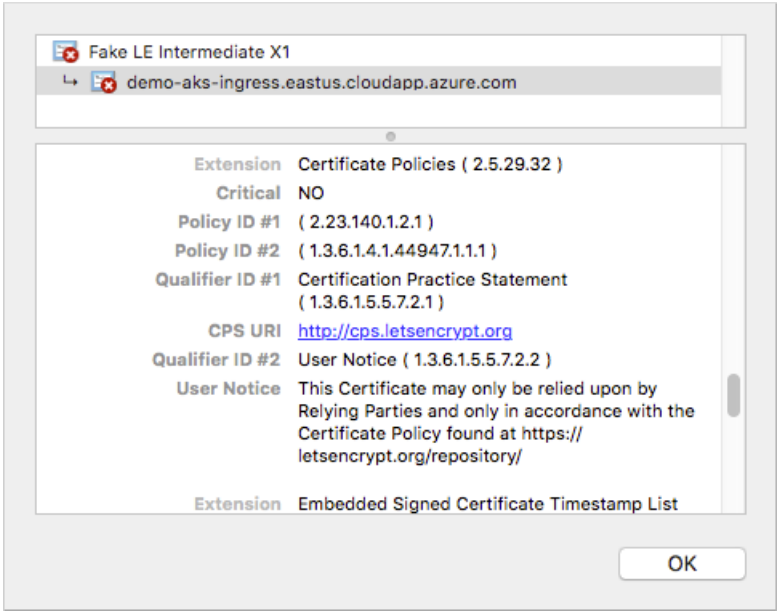
Create the ingress resource using the `kubectl apply -f hello-world-ingress.yaml` command.

```
$ kubectl apply -f hello-world-ingress.yaml

ingress.extensions/hello-world-ingress created
```

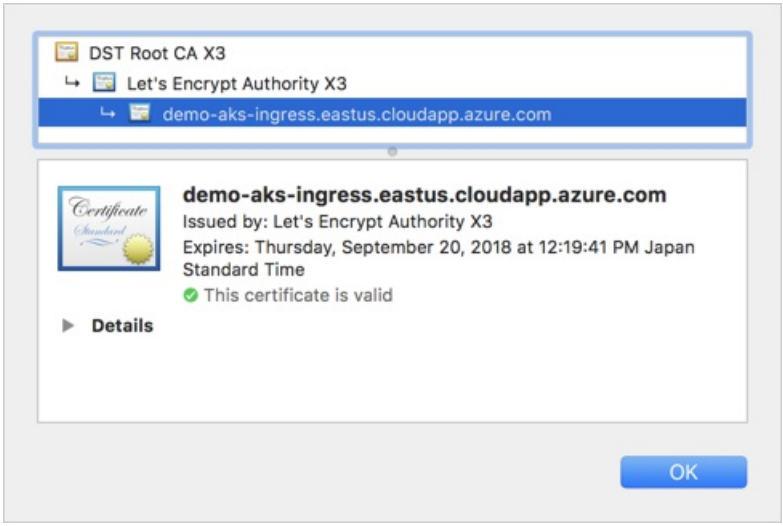
Test the ingress configuration

Open a web browser to the FQDN of your Kubernetes ingress controller, such as *https://demo-aks-ingress.eastus.cloudapp.azure.com*.

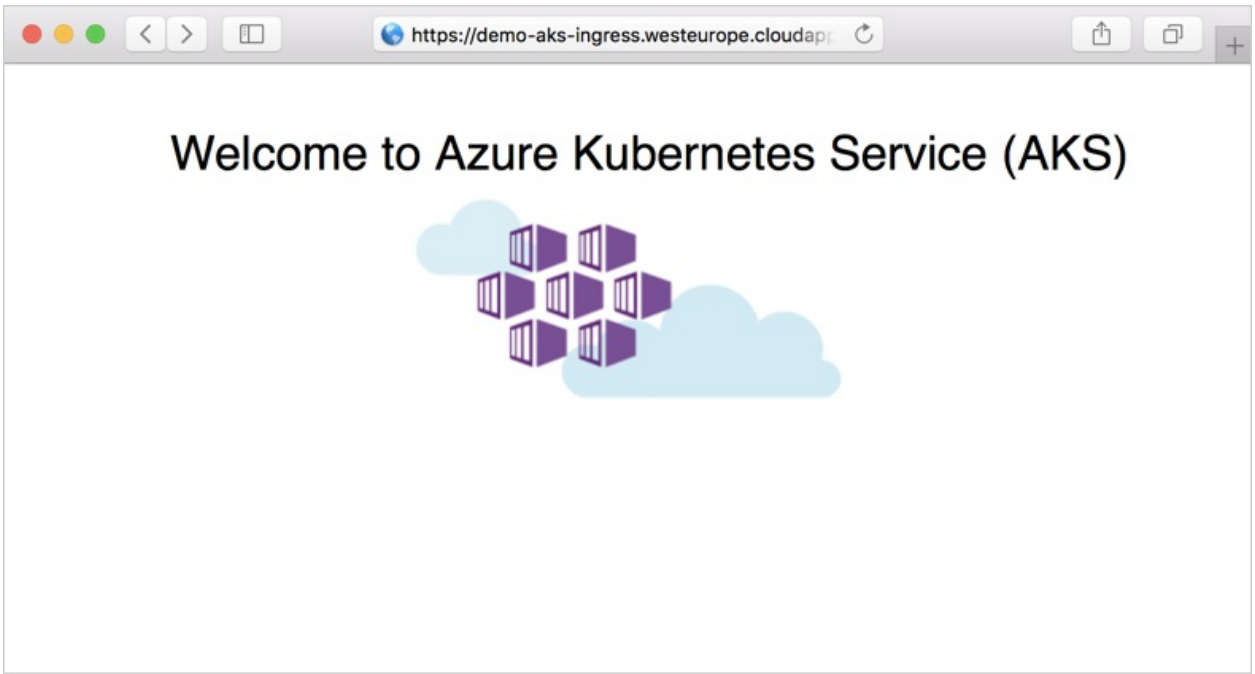
As these examples use `letsencrypt-staging`, the issued SSL certificate is not trusted by the browser. Accept the warning prompt to continue to your application. The certificate information shows this *Fake LE Intermediate X1* certificate is issued by Let's Encrypt. This fake certificate indicates `cert-manager` processed the request correctly and received a certificate from the provider:



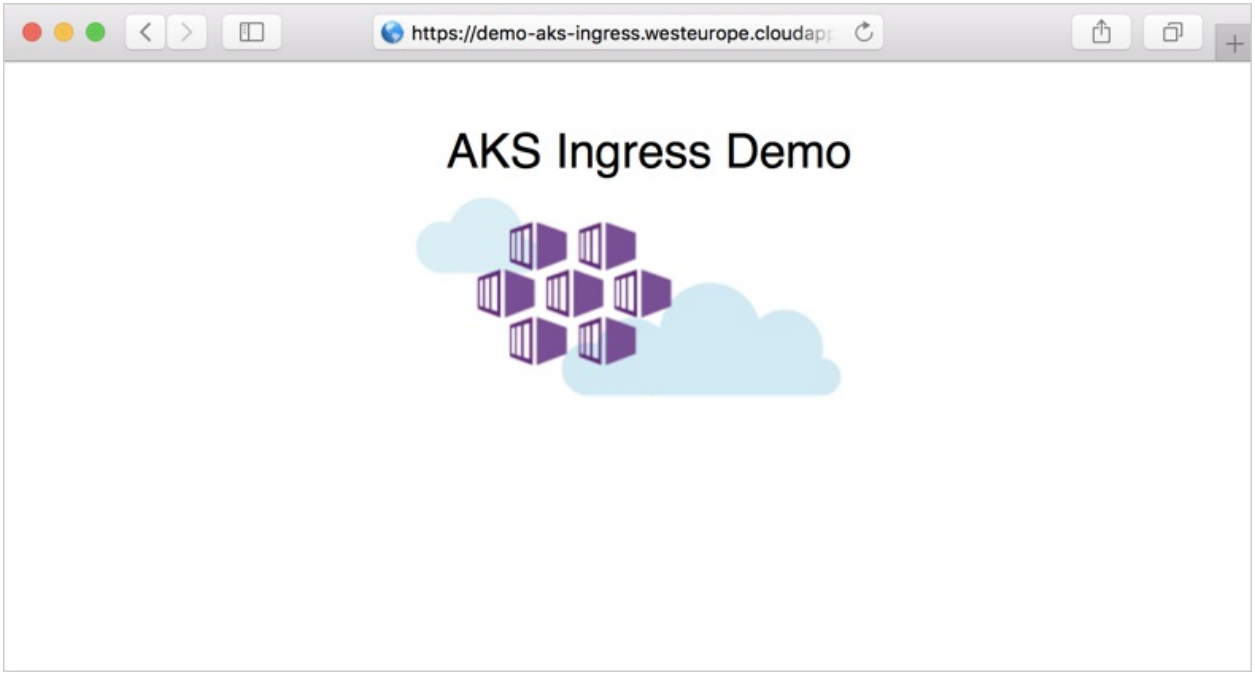
When you change Let's Encrypt to use `prod` rather than `staging`, a trusted certificate issued by Let's Encrypt is used, as shown in the following example:



The demo application is shown in the web browser:



Now add the */hello-world-two* path to the FQDN, such as *https://demo-aks-ingress.eastus.cloudapp.azure.com/hello-world-two*. The second demo application with the custom title is shown:



Next steps

This article included some external components to AKS. To learn more about these components, see the following project pages:

- Helm CLI
- NGINX ingress controller
- cert-manager

You can also:

- Create a basic ingress controller with external network connectivity
- Enable the HTTP application routing add-on
- Create an ingress controller that uses an internal, private network and IP address
- Create an ingress controller with a dynamic public IP and configure Let's Encrypt to automatically generate TLS certificates

Use a static public IP address for egress traffic in Azure Kubernetes Service (AKS)

9/27/2018 • 4 minutes to read • Edit Online

By default, the egress IP address from an Azure Kubernetes Service (AKS) cluster is randomly assigned. This configuration is not ideal when you need to identify an IP address for access to external services, for example. Instead, you may need to assign a static IP address that can be whitelisted for service access.

This article shows you how to create and use a static public IP address for use with egress traffic in an AKS cluster.

Before you begin

This article assumes that you have an existing AKS cluster. If you need an AKS cluster, see the AKS quickstart using the Azure CLI or using the Azure portal.

You also need the Azure CLI version 2.0.46 or later installed and configured. Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI.

Egress traffic overview

Outbound traffic from an AKS cluster follows Azure Load Balancer conventions. Before the first Kubernetes service of type `LoadBalancer` is created, the agent nodes in an AKS cluster are not part of any Azure Load Balancer pool. In this configuration, the nodes have no instance level Public IP address. Azure translates the outbound flow to a public source IP address that is not configurable or deterministic.

Once a Kubernetes service of type `LoadBalancer` is created, agent nodes are added to an Azure Load Balancer pool. For outbound flow, Azure translates it to the first public IP address configured on the load balancer. This public IP address is only valid for the lifespan of that resource. If you delete the Kubernetes LoadBalancer service, the associated load balancer and IP address are also deleted. If you want to assign a specific IP address or retain an IP address for redeployed Kubernetes services, you can create and use a static public IP address.

Create a static public IP

When you create a static public IP address for use with AKS, the IP address resource must be created in the **node** resource group. Get the resource group name with the az aks show command and add the `--query nodeResourceGroup` query parameter. The following example gets the node resource group for the AKS cluster name *myAKSCluster* in the resource group name *myResourceGroup*:

```
$ az aks show --resource-group myResourceGroup --name myAKSCluster --query nodeResourceGroup -o tsv

MC_myResourceGroup_myAKSCluster_eastus
```

Now create a static public IP address with the az network public ip create command. Specify the node resource group name obtained in the previous command, and then a name for the IP address resource, such as *myAKSPublicIP*:

```
az network public-ip create \
    --resource-group MC_myResourceGroup_myAKSCluster_eastus \
    --name myAKSPublicIP \
    --allocation-method static
```

The IP address is shown, as shown in the following condensed example output:

```
{
  "publicIp": {
    "dnsSettings": null,
    "etag": "W/\"6b6fb15c-5281-4f64-b332-8f68f46e1358\"",
    "id":
"/subscriptions/<SubscriptionID>/resourceGroups/MC_myResourceGroup_myAKSCluster_eastus/providers/Microsoft.Netw
ork/publicIPAddresses/myAKSPublicIP",
    "idleTimeoutInMinutes": 4,
    "ipAddress": "40.121.183.52",
    [..]
  }
```

You can later get the public IP address using the az network public-ip list command. Specify the name of the node resource group, and then query for the *ipAddress* as shown in the following example:

```
$ az network public-ip list --resource-group MC_myResourceGroup_myAKSCluster_eastus --query [0].ipAddress --
output tsv

40.121.183.52
```

Create a service with the static IP

To create a service with the static public IP address, add the `loadBalancerIP` property and the value of the static public IP address to the YAML manifest. Create a file named `egress-service.yaml` and copy in the following YAML. Provide your own public IP address created in the previous step.

```
apiVersion: v1
kind: Service
metadata:
  name: azure-egress
spec:
  loadBalancerIP: 40.121.183.52
  type: LoadBalancer
  ports:
  - port: 80
```

Create the service and deployment with the `kubectl apply` command.

```
kubectl apply -f egress-service.yaml
```

This service configures a new frontend IP on the Azure Load Balancer. If you do not have any other IPs configured, then **all** egress traffic should now use this address. When multiple addresses are configured on the Azure Load Balancer, egress uses the first IP on that load balancer.

Verify egress address

To verify that the static public IP address is being used, you can use DNS look-up service such as `checkip.dyndns.org`.

Start and attach to a basic *Debian* pod:

```
kubectl run -it --rm aks-ip --image=debian
```

To access a web site from within the container, use `apt-get` to install `curl` into the container.

```
apt-get update && apt-get install curl -y
```

Now use curl to access the *checkip.dyndns.org* site. The egress IP address is shown, as displayed in the following example output. This IP address matches the static public IP address created and defined for the loadBalancer service:

```
$ curl -s checkip.dyndns.org

<html><head><title>Current IP Check</title></head><body>Current IP Address: 23.101.128.81</body></html>
```

Next steps

To avoid maintaining multiple public IP addresses on the Azure Load Balancer, you can instead use an ingress controller. Ingress controllers provide additional benefits such as SSL/TLS termination, support for URI rewrites, and upstream SSL/TLS encryption. For more information, see Create a basic ingress controller in AKS.

Service principals with Azure Kubernetes Service (AKS)

9/26/2018 • 3 minutes to read • Edit Online

To interact with Azure APIs, an AKS cluster requires an Azure Active Directory (AD) service principal. The service principal is needed to dynamically create and manage other Azure resources such as an Azure load balancer or container registry (ACR).

This article shows how to create and use a service principal for your AKS clusters.

Before you begin

To create an Azure AD service principal, you must have permissions to register an application with your Azure AD tenant, and to assign the application to a role in your subscription. If you don't have the necessary permissions, you might need to ask your Azure AD or subscription administrator to assign the necessary permissions, or pre-create a service principal for you to use with the AKS cluster.

You also need the Azure CLI version 2.0.46 or later installed and configured. Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI.

Automatically create and use a service principal

When you create an AKS cluster in the Azure portal or using the az aks create command, Azure can automatically generate a service principal.

In the following Azure CLI example, a service principal is not specified. In this scenario, the Azure CLI creates a service principal for the AKS cluster. To successfully complete the operation, your Azure account must have the proper rights to create a service principal.

```
az aks create --name myAKSCluster --resource-group myResourceGroup --generate-ssh-keys
```

Manually create a service principal

To manually create a service principal with the Azure CLI, use the az ad sp create-for-rbac command. In the following example, the `--skip-assignment` parameter prevents any additional default assignments being assigned:

```
az ad sp create-for-rbac --skip-assignment
```

The output is similar to the following example. Make a note of your own `appId` and `password`. These values are used when you create an AKS cluster in the next section.

```
{
  "appId": "559513bd-0c19-4c1a-87cd-851a26afd5fc",
  "displayName": "azure-cli-2018-09-25-21-10-19",
  "name": "http://azure-cli-2018-09-25-21-10-19",
  "password": "e763725a-5eee-40e8-a466-dc88d980f415",
  "tenant": "72f988bf-86f1-41af-91ab-2d7cd011db48"
}
```

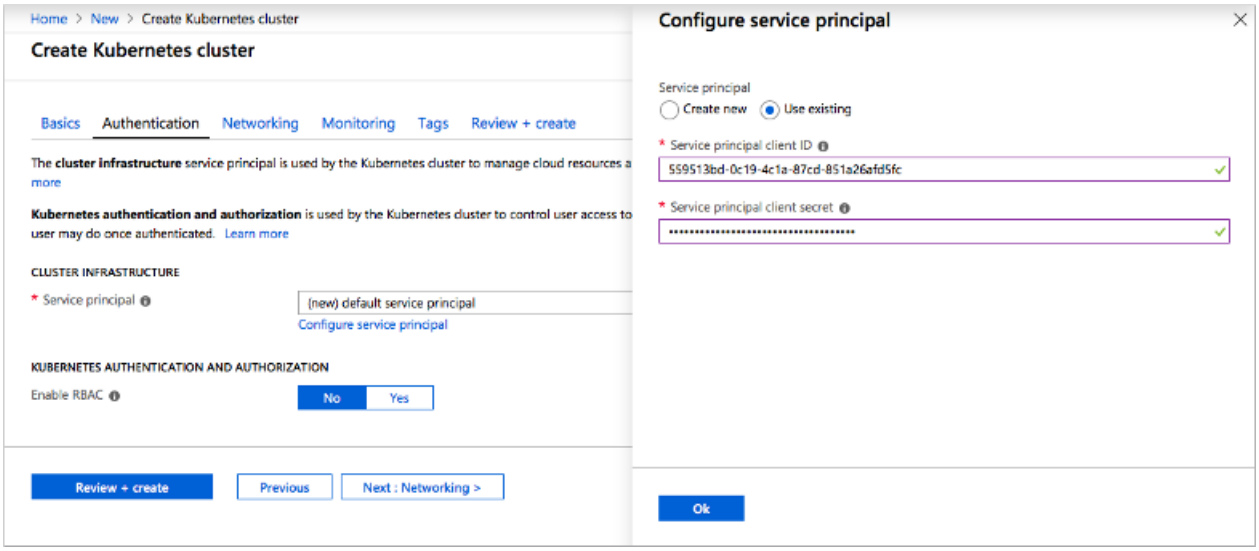
Specify a service principal for an AKS cluster

To use an existing service principal when you create an AKS cluster using the az aks create command, use the `--service-principal` and `--client-secret` parameters to specify the `appId` and `password` from the output of the az ad sp create-for-rbac command:

```
az aks create \
    --resource-group myResourceGroup \
    --name myAKSCluster \
    --service-principal <appId> \
    --client-secret <password>
```

If you deploy an AKS cluster using the Azure portal, on the *Authentication* page of the **Create Kubernetes cluster** dialog, choose to **Configure service principal**. Select **Use existing**, and specify the following values:

- **Service principal client ID** is your *appId*
- **Service principal client secret** is the *password* value



Additional considerations

When using AKS and Azure AD service principals, keep the following considerations in mind.

- The service principal for Kubernetes is a part of the cluster configuration. However, don't use the identity to deploy the cluster.
- Every service principal is associated with an Azure AD application. The service principal for a Kubernetes cluster can be associated with any valid Azure AD application name (for example: *https://www.contoso.org/example*). The URL for the application doesn't have to be a real endpoint.
- When you specify the service principal **Client ID**, use the value of the `appId`.
- On the master and node VMs in the Kubernetes cluster, the service principal credentials are stored in the file `/etc/kubernetes/azure.json`
- When you use the az aks create command to generate the service principal automatically, the service principal credentials are written to the file `~/.azure/aksServicePrincipal.json` on the machine used to run the command.
- When you delete an AKS cluster that was created by az aks create, the service principal that was created automatically is not deleted.
    - To delete the service principal, first get the ID for the service principal with az ad app list. The following example queries for the cluster named *myAKSCluster* and then deletes the app ID with az ad app delete. Replace these names with your own values:

```
az ad app list --query "[?displayName=='myAKSCluster'].{Name:displayName,Id:appId}" --output
table
az ad app delete --id <appId>
```

Next steps

For more information about Azure Active Directory service principals, see Application and service principal objects

Authenticate with Azure Container Registry from Azure Kubernetes Service

10/8/2018 • 2 minutes to read • Edit Online

When you're using Azure Container Registry (ACR) with Azure Kubernetes Service (AKS), an authentication mechanism needs to be established. This article details the recommended configurations for authentication between these two Azure services.

Grant AKS access to ACR

When you create an AKS cluster, Azure also creates a service principal to support cluster operability with other Azure resources. You can use this auto-generated service principal for authentication with an ACR registry. To do so, you need to create an Azure AD role assignment that grants the cluster's service principal access to the container registry.

Use the following script to grant the AKS-generated service principal access to an Azure container registry. Modify the `AKS_*` and `ACR_*` variables for your environment before running the script.

```
#!/bin/bash

AKS_RESOURCE_GROUP=myAKSResourceGroup
AKS_CLUSTER_NAME=myAKSCluster
ACR_RESOURCE_GROUP=myACRResourceGroup
ACR_NAME=myACRRegistry

# Get the id of the service principal configured for AKS
CLIENT_ID=$(az aks show --resource-group $AKS_RESOURCE_GROUP --name $AKS_CLUSTER_NAME --query
"servicePrincipalProfile.clientId" --output tsv)

# Get the ACR registry resource id
ACR_ID=$(az acr show --name $ACR_NAME --resource-group $ACR_RESOURCE_GROUP --query "id" --output tsv)

# Create role assignment
az role assignment create --assignee $CLIENT_ID --role Reader --scope $ACR_ID
```

Access with Kubernetes secret

In some instances, you might not be able to assign the required role to the auto-generated AKS service principal granting it access to ACR. For example, due to your organization's security model, you might not have sufficient permission in your Azure AD directory to assign a role to the AKS-generated service principal. In such a case, you can create a new service principal, then grant it access to the container registry using a Kubernetes image pull secret.

Use the following script to create a new service principal (you'll use its credentials for the Kubernetes image pull secret). Modify the `ACR_NAME` variable for your environment before running the script.

```
#!/bin/bash

ACR_NAME=myacrinstance
SERVICE_PRINCIPAL_NAME=acr-service-principal

# Populate the ACR login server and resource id.
ACR_LOGIN_SERVER=$(az acr show --name $ACR_NAME --query loginServer --output tsv)
ACR_REGISTRY_ID=$(az acr show --name $ACR_NAME --query id --output tsv)

# Create a 'Reader' role assignment with a scope of the ACR resource.
SP_PASSWD=$(az ad sp create-for-rbac --name $SERVICE_PRINCIPAL_NAME --role Reader --scopes $ACR_REGISTRY_ID --
query password --output tsv)

# Get the service principal client id.
CLIENT_ID=$(az ad sp show --id http://$SERVICE_PRINCIPAL_NAME --query appId --output tsv)

# Output used when creating Kubernetes secret.
echo "Service principal ID: $CLIENT_ID"
echo "Service principal password: $SP_PASSWD"
```

You can now store the service principal's credentials in a Kubernetes image pull secret, which your AKS cluster will reference when running containers.

Use the following **kubectl** command to create the Kubernetes secret. Replace `<acr-login-server>` with the fully qualified name of your Azure container registry (it's in the format "acrname.azurecr.io"). Replace `<service-principal-ID>` and `<service-principal-password>` with the values you obtained by running the previous script. Replace `<email-address>` with any well-formed email address.

```
kubectl create secret docker-registry acr-auth --docker-server <acr-login-server> --docker-username <service-
principal-ID> --docker-password <service-principal-password> --docker-email <email-address>
```

You can now use the Kubernetes secret in pod deployments by specifying its name (in this case, "acr-auth") in the `imagePullSecrets` parameter:

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: acr-auth-example
spec:
  template:
    metadata:
      labels:
        app: acr-auth-example
    spec:
      containers:
      - name: acr-auth-example
        image: myacrregistry.azurecr.io/acr-auth-example
      imagePullSecrets:
      - name: acr-auth
```

2 minutes to read

Enable and review Kubernetes master node logs in Azure Kubernetes Service (AKS)

9/13/2018 • 4 minutes to read • Edit Online

With Azure Kubernetes Service (AKS), the master components such as the *kube-apiserver* and *kube-controller-manager* are provided as a managed service. You create and manage the nodes that run the *kubelet* and container runtime, and deploy your applications through the managed Kubernetes API server. To help troubleshoot your application and services, you may need to view the logs generated by these master components. This article shows you how to use Azure Log Analytics to enable and query the logs from the Kubernetes master components.

Before you begin

This article requires an existing AKS cluster running in your Azure account. If you do not already have an AKS cluster, create one using the Azure CLI or Azure portal. Log Analytics works with both RBAC and non-RBAC enabled AKS clusters.

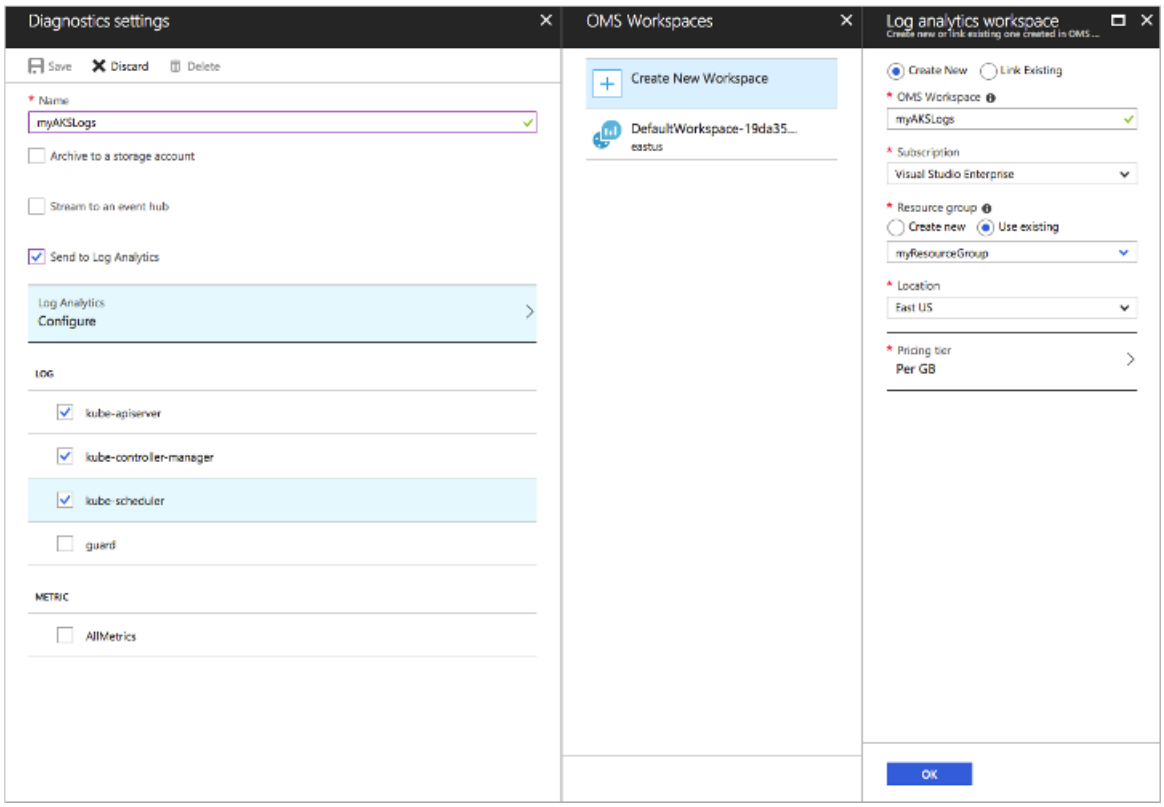
Enable diagnostics logs

To help collect and review data from multiple sources, Log Analytics provides a query language and analytics engine that provides insights to your environment. A workspace is used to collate and analyze the data, and can integrate with other Azure services such as Application Insights and Security Center. To use a different platform to analyze the logs, you can instead choose to send diagnostic logs to an Azure storage account or event hub. For more information, see What is Azure Log Analytics?.

Log Analytics is enabled and managed in the Azure portal. To enable log collection for the Kubernetes master components in your AKS cluster, open the Azure portal in a web browser and complete the following steps:

1. Select the resource group for your AKS cluster, such as *myResourceGroup*. Don't select the resource group that contains your individual AKS cluster resources, such as *MC_myResourceGroup_myAKSCluster_eastus*.
2. On the left-hand side, choose **Diagnostic settings**.
3. Select your AKS cluster, such as *myAKSCluster*, then choose to **Turn on diagnostics**.
4. Enter a name, such as *myAKSLogs*, then select the option to **Send to Log Analytics**.
   - Choose to *Configure* Log Analytics, then select an existing workspace or **Create New Workspace**.
   - If you need to create a workspace, provide a name, a resource group, and a location.
5. In the list of available logs, select the logs you wish to enable, such as *kube-apiserver*, *kube-controller-manager*, and *kube-scheduler*. You can return and change the collected logs once Log Analytics are enabled.
6. When ready, select **Save** to enable collection of the selected logs.

The following example portal screenshot shows the *Diagnostics settings* window and then option to create an OMS workspace:



Schedule a test pod on the AKS cluster

To generate some logs, create a new pod in your AKS cluster. The following example YAML manifest can be used to create a basic NGINX instance. Create a file named `nginx.yaml` in an editor of your choice and paste the following content:

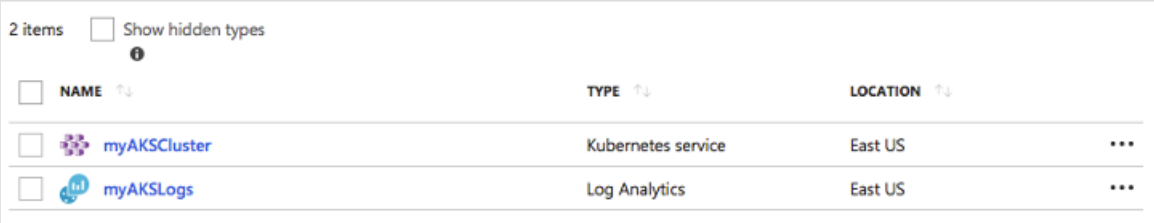
```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: myfrontend
    image: nginx
    ports:
    - containerPort: 80
```

Create the pod with the kubectl create command and specify your YAML file, as shown in the following example:

```
$ kubectl create -f nginx.yaml

pod/nginx created
```

View collected logs

It may take a few minutes for the diagnostics logs to be enabled and appear in the OMS workspace. In the Azure portal, select the resource group for your Log Analytics workspace, such as *myResourceGroup*, then choose your Log Analytics resource, such as *myAKSLogs*.



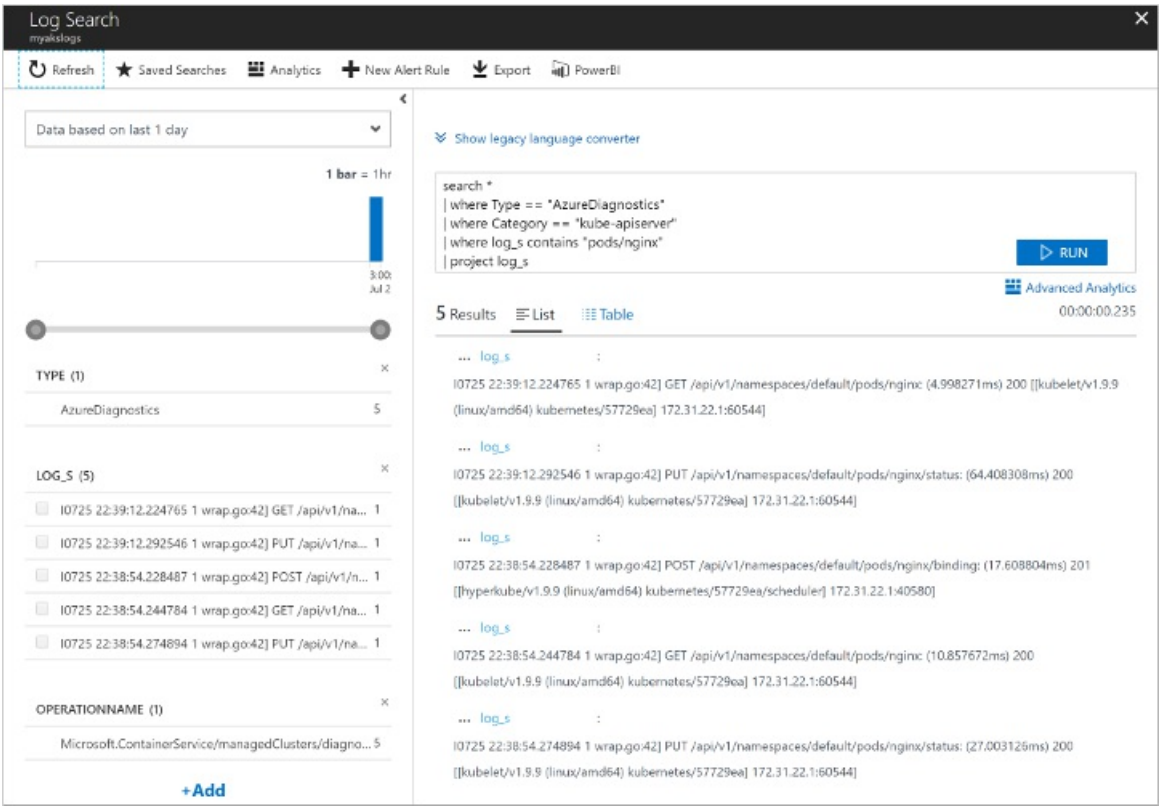
On the left-hand side, choose **Log Search**. To view the *kube-apiserver*, enter the following query in the text box:

```
AzureDiagnostics
| where Category == "kube-apiserver"
| project log_s
```

Many logs are likely returned for the API server. To scope down the query to view the logs about the NGINX pod created in the previous step, add an additional *where* statement to search for *pods/nginx* as shown in the following example query:

```
AzureDiagnostics
| where Category == "kube-apiserver"
| where log_s contains "pods/nginx"
| project log_s
```

The specific logs for your NGINX pod are displayed, as shown in the following example screenshot:



To view additional logs, you can update the query for the *Category* name to *kube-controller-manager* or *kube-scheduler*, depending on what additional logs you enable. Additional *where* statements can then be used to refine the events you are looking for.

For more information on how to query and filter your log data, see View or analyze data collected with Log Analytics log search.

Log event schema

To help analyze the log data, the following table details the schema used for each event:

FIELD NAME	DESCRIPTION
resourceId	Azure resource that produced the log
time	Timestamp of when the log was uploaded
category	Name of container/component generating the log
operationName	Always *Microsoft.ContainerService/managedClusters/diagnositicLogs/Read*
properties.log	Full text of the log from the component
properties.stream	*stderr* or *stdout*
properties.pod	Pod name that the log came from
properties.containerID	Id of the docker container this log came from

Next steps

In this article, you learned how to enable and review the logs for the Kubernetes master components in your AKS cluster. To monitor and troubleshoot further, you can also view the Kubelet logs and enable SSH node access.

Get kubelet logs from Azure Kubernetes Service (AKS) cluster nodes

8/22/2018 • 2 minutes to read • Edit Online

Occasionally, you may need to get *kubelet* logs from an Azure Kubernetes Service (AKS) node for troubleshooting purposes. This article shows you how you can use `journalctl` to view the *kubelet* logs.

Create an SSH connection

First, create an SSH connection with the node on which you need to view *kubelet* logs. This operation is detailed in the SSH into Azure Kubernetes Service (AKS) cluster nodes document.

Get kubelet logs

Once you have connected to the node, run the following command to pull the *kubelet* logs:

```
sudo journalctl -u kubelet -o cat
```

The following sample output shows the *kubelet* log data:

```
I0508 12:26:17.905042    8672 kubelet_node_status.go:497] Using Node Hostname from cloudprovider: "aks-
agentpool-11482510-0"
I0508 12:26:27.943494    8672 kubelet_node_status.go:497] Using Node Hostname from cloudprovider: "aks-
agentpool-11482510-0"
I0508 12:26:28.920125    8672 server.go:796] GET /stats/summary: (10.370874ms) 200 [[Ruby] 10.244.0.2:52292]
I0508 12:26:37.964650    8672 kubelet_node_status.go:497] Using Node Hostname from cloudprovider: "aks-
agentpool-11482510-0"
I0508 12:26:47.996449    8672 kubelet_node_status.go:497] Using Node Hostname from cloudprovider: "aks-
agentpool-11482510-0"
I0508 12:26:58.019746    8672 kubelet_node_status.go:497] Using Node Hostname from cloudprovider: "aks-
agentpool-11482510-0"
I0508 12:27:05.107680    8672 server.go:796] GET /stats/summary/: (24.853838ms) 200 [[Go-http-client/1.1]
10.244.0.3:44660]
I0508 12:27:08.041736    8672 kubelet_node_status.go:497] Using Node Hostname from cloudprovider: "aks-
agentpool-11482510-0"
I0508 12:27:18.068505    8672 kubelet_node_status.go:497] Using Node Hostname from cloudprovider: "aks-
agentpool-11482510-0"
I0508 12:27:28.094889    8672 kubelet_node_status.go:497] Using Node Hostname from cloudprovider: "aks-
agentpool-11482510-0"
I0508 12:27:38.121346    8672 kubelet_node_status.go:497] Using Node Hostname from cloudprovider: "aks-
agentpool-11482510-0"
I0508 12:27:44.015205    8672 server.go:796] GET /stats/summary: (30.236824ms) 200 [[Ruby] 10.244.0.2:52588]
I0508 12:27:48.145640    8672 kubelet_node_status.go:497] Using Node Hostname from cloudprovider: "aks-
agentpool-11482510-0"
I0508 12:27:58.178534    8672 kubelet_node_status.go:497] Using Node Hostname from cloudprovider: "aks-
agentpool-11482510-0"
I0508 12:28:05.040375    8672 server.go:796] GET /stats/summary/: (27.78503ms) 200 [[Go-http-client/1.1]
10.244.0.3:44660]
I0508 12:28:08.214158    8672 kubelet_node_status.go:497] Using Node Hostname from cloudprovider: "aks-
agentpool-11482510-0"
I0508 12:28:18.242160    8672 kubelet_node_status.go:497] Using Node Hostname from cloudprovider: "aks-
agentpool-11482510-0"
I0508 12:28:28.274408    8672 kubelet_node_status.go:497] Using Node Hostname from cloudprovider: "aks-
agentpool-11482510-0"
I0508 12:28:38.296074    8672 kubelet_node_status.go:497] Using Node Hostname from cloudprovider: "aks-
agentpool-11482510-0"
I0508 12:28:48.321952    8672 kubelet_node_status.go:497] Using Node Hostname from cloudprovider: "aks-
agentpool-11482510-0"
I0508 12:28:58.344656    8672 kubelet_node_status.go:497] Using Node Hostname from cloudprovider: "aks-
agentpool-11482510-0"
```

Next steps

If you need additional troubleshooting information from the Kubernetes master, see view Kubernetes master node logs in AKS.

Install applications with Helm in Azure Kubernetes Service (AKS)

7/13/2018 • 5 minutes to read • Edit Online

Helm is an open-source packaging tool that helps you install and manage the lifecycle of Kubernetes applications. Similar to Linux package managers such as *APT* and *Yum*, Helm is used to manage Kubernetes charts, which are packages of preconfigured Kubernetes resources.

This article shows you how to configure and use Helm in a Kubernetes cluster on AKS.

Before you begin

The steps detailed in this document assume that you have created an AKS cluster and have established a `kubectl` connection with the cluster. If you need these items see, the AKS quickstart.

Install Helm CLI

The Helm CLI is a client that runs on your development system and allows you to start, stop, and manage applications with Helm.

If you use the Azure Cloud Shell, the Helm CLI is already installed. To install the Helm CLI on a Mac, use `brew`. For additional installation options see, Installing Helm.

```
brew install kubernetes-helm
```

Output:

```
==> Downloading https://homebrew.bintray.com/bottles/kubernetes-helm-2.9.1.high_sierra.bottle.tar.gz
######################################################################## 100.0%
==> Pouring kubernetes-helm-2.9.1.high_sierra.bottle.tar.gz
==> Caveats
Bash completion has been installed to:
  /usr/local/etc/bash_completion.d
==> Summary
🍺  /usr/local/Cellar/kubernetes-helm/2.9.1: 50 files, 66.2MB
```

Create a service account

Before you can deploy Helm in an RBAC-enabled cluster, you need a service account and role binding for the Tiller service. For more information on securing Helm / Tiller in an RBAC enabled cluster, see Tiller, Namespaces, and RBAC. If your cluster is not RBAC enabled, skip this step.

Create a file named `helm-rbac.yaml` and copy in the following YAML:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: tiller
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: tiller
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
  - kind: ServiceAccount
    name: tiller
    namespace: kube-system
```

Create the service account and role binding with the `kubectl create` command:

```
kubectl create -f helm-rbac.yaml
```

Secure Tiller and Helm

The Helm client and Tiller service authenticate and communicate with each other using TLS/SSL. This authentication method helps to secure the Kubernetes cluster and what services can be deployed. To improve security, you can generate your own signed certificates. Each Helm user would receive their own client certificate, and Tiller would be initialized in the Kubernetes cluster with certificates applied. For more information, see Using TLS/SSL between Helm and Tiller.

With an RBAC-enabled Kubernetes cluster, you can control the level of access Tiller has to the cluster. You can define the Kubernetes namespace that Tiller is deployed in, and restrict what namespaces Tiller can then deploy resources in. This approach lets you create Tiller instances in different namespaces and limit deployment boundaries, and scope the users of Helm client to certain namespaces. For more information, see Helm role-based access controls.

Configure Helm

To deploy a basic Tiller into an AKS cluster, use the helm init command. If your cluster is not RBAC enabled, remove the `--service-account` argument and value. If you configured TLS/SSL for Tiller and Helm, skip this basic initialization step and instead provide the required `--tiller-tls-` as shown in the next example.

```
helm init --service-account tiller
```

If you configured TLS/SSL between Helm and Tiller provide the `--tiller-tls-` parameters and names of your own certificates, as shown in the following example:

```
helm init \
    --tiller-tls \
    --tiller-tls-cert tiller.cert.pem \
    --tiller-tls-key tiller.key.pem \
    --tiller-tls-verify \
    --tls-ca-cert ca.cert.pem \
    --service-account tiller
```

Find Helm charts

Helm charts are used to deploy applications into a Kubernetes cluster. To search for pre-created Helm charts, use the helm search command:

```
helm search
```

The following condensed example output shows some of the Helm charts available for use:

```
$ helm search

NAME                            CHART VERSION    APP VERSION  DESCRIPTION
stable/acs-engine-autoscaler    2.2.0            2.1.1        Scales worker nodes within agent pools
stable/aerospike                0.1.7            v3.14.1.2    A Helm chart for Aerospike in Kubernetes
stable/anchore-engine           0.1.7            0.1.10       Anchore container analysis and policy
evaluatio...
stable/apm-server               0.1.0            6.2.4        The server receives data from the Elastic APM
a...
stable/ark                      1.0.1            0.8.2        A Helm chart for ark
stable/artifactory              7.2.1            6.0.0        Universal Repository Manager supporting all
maj...
stable/artifactory-ha           0.2.1            6.0.0        Universal Repository Manager supporting all
maj...
stable/auditbeat                0.1.0            6.2.4        A lightweight shipper to audit the activities
o...
stable/aws-cluster-autoscaler   0.3.3                         Scales worker nodes within autoscaling groups.
stable/bitcoind                 0.1.3            0.15.1       Bitcoin is an innovative payment network and a
...
stable/buildkite                0.2.3            3            Agent for Buildkite
stable/burrow                   0.4.4            0.17.1       Burrow is a permissionable smart contract machine
stable/centrifugo               2.0.1            1.7.3        Centrifugo is a real-time messaging server.
stable/cerebro                  0.1.0            0.7.3        A Helm chart for Cerebro - a web admin tool
tha...
stable/cert-manager             v0.3.3           v0.3.1       A Helm chart for cert-manager
stable/chaoskube                0.7.0            0.8.0        Chaoskube periodically kills random pods in
you...
stable/chartmuseum              1.5.0            0.7.0        Helm Chart Repository with support for Amazon
S...
stable/chronograf               0.4.5            1.3          Open-source web application written in Go and
R...
stable/cluster-autoscaler       0.6.4            1.2.2        Scales worker nodes within autoscaling groups.
stable/cockroachdb              1.1.1            2.0.0        CockroachDB is a scalable, survivable,
strongly...
stable/concourse                1.10.1           3.14.1       Concourse is a simple and scalable CI system.
stable/consul                   3.2.0            1.0.0        Highly available and distributed service
discov...
stable/coredns                  0.9.0            1.0.6        CoreDNS is a DNS server that chains plugins
and...
stable/coscale                  0.2.1            3.9.1        CoScale Agent
stable/dask                     1.0.4            0.17.4       Distributed computation in Python with task
sch...
stable/dask-distributed         2.0.2                         DEPRECATED: Distributed computation in Python
stable/datadog                  0.18.0           6.3.0        DataDog Agent
...
```

To update the list of charts, use the helm repo update command. The following example shows a successful repo update:

```
$ helm repo update

Hang tight while we grab the latest from your chart repositories...
...Skip local chart repository
...Successfully got an update from the "stable" chart repository
Update Complete. ⎈ Happy Helming!⎈
```

Run Helm charts

To install charts with Helm, use the helm install command and specify the name of the chart to install. To see this in action, let's install a basic Wordpress deployment using a Helm chart. If you configured TLS/SSL, add the `--tls` parameter to use your Helm client certificate.

```
helm install stable/wordpress
```

The following condensed example output shows the deployment status of the Kubernetes resources created by the Helm chart:

```
$ helm install stable/wordpress

NAME:   wishful-mastiff
LAST DEPLOYED: Thu Jul 12 15:53:56 2018
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1beta1/Deployment
NAME                       DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
wishful-mastiff-wordpress  1        1        1           0          1s

==> v1beta1/StatefulSet
NAME                     DESIRED  CURRENT  AGE
wishful-mastiff-mariadb  1        1        1s

==> v1/Pod(related)
NAME                                     READY  STATUS   RESTARTS  AGE
wishful-mastiff-wordpress-6f96f8fdf9-q84sz  0/1    Pending  0         1s
wishful-mastiff-mariadb-0                   0/1    Pending  0         1s

==> v1/Secret
NAME                       TYPE    DATA  AGE
wishful-mastiff-mariadb    Opaque  2     2s
wishful-mastiff-wordpress  Opaque  2     2s

==> v1/ConfigMap
NAME                           DATA  AGE
wishful-mastiff-mariadb        1     2s
wishful-mastiff-mariadb-tests  1     2s

==> v1/PersistentVolumeClaim
NAME                       STATUS   VOLUME   CAPACITY  ACCESS MODES  STORAGECLASS  AGE
wishful-mastiff-wordpress  Pending  default  2s

==> v1/Service
NAME                       TYPE          CLUSTER-IP   EXTERNAL-IP  PORT(S)                      AGE
wishful-mastiff-mariadb    ClusterIP     10.1.116.54  <none>       3306/TCP                     2s
wishful-mastiff-wordpress  LoadBalancer  10.1.217.64  <pending>    80:31751/TCP,443:31264/TCP  2s
...
```

It takes a minute or two for the *EXTERNAL-IP* address of the Wordpress service to be populated and allow you to

access it with a web browser.

List Helm releases

To see a list of releases installed on your cluster, use the helm list command. The following example shows the Wordpress release deployed in the previous step. If you configured TLS/SSL, add the `--tls` parameter to use your Helm client certificate.

```
$ helm list

NAME             REVISION    UPDATED                     STATUS      CHART              NAMESPACE
wishful-mastiff  1           Thu Jul 12 15:53:56 2018    DEPLOYED    wordpress-2.1.3  default
```

Next steps

For more information about managing Kubernetes application deployments with Helm, see the Helm documentation.

Helm documentation

2 minutes to read

Use Draft with Azure Kubernetes Service (AKS)

9/10/2018 • 6 minutes to read • Edit Online

Draft is an open-source tool that helps package and deploy application containers in a Kubernetes cluster, leaving you free to concentrate on the dev cycle - the "inner loop" of concentrated development. Draft works as the code is being developed, but before committing to version control. With Draft, you can quickly redeploy an application to Kubernetes as code changes occur. For more information on Draft, see the Draft documentation on Github.

This article shows you how to use Draft with a Kubernetes cluster on AKS.

Prerequisites

The steps detailed in this article assume that you have created an AKS cluster and have established a `kubectl` connection with the cluster. If you need these items, see the AKS quickstart.

You need a private Docker registry in Azure Container Registry (ACR). For steps on how to create an ACR instance, see the Azure Container Registry quickstart.

Helm must also be installed in your AKS cluster. For more information on how to install and configure Helm, see Use Helm with Azure Kubernetes Service (AKS).

Finally, you must install Docker.

Install Draft

The Draft CLI is a client that runs on your development system and allows you to deploy code into a Kubernetes cluster. To install the Draft CLI on a Mac, use `brew`. For additional installation options, see the Draft Install guide.

> **NOTE**
> If you installed Draft prior to version 0.12, first delete Draft from your cluster using `helm delete --purge draft` and then remove your local configuration by running `rm -rf ~/.draft`. If you are on MacOS, then run `brew upgrade draft`.

```
brew tap azure/draft
brew install draft
```

Now initialize Draft with the `draft init` command:

```
draft init
```

Configure Draft

Draft builds the container images locally, and then either deploys them from the local registry (such as with Minikube), or uses an image registry that you specify. This article uses Azure Container Registry (ACR), so you must establish a trust relationship between your AKS cluster and the ACR registry, then configure Draft to push your container images to ACR.

Create trust between AKS cluster and ACR

To establish trust between an AKS cluster and an ACR registry, grant permissions for the Azure Active Directory

service principal used by the AKS cluster to access the ACR registry. In the following commands, provide your own `<resourceGroupName>`, replace `<aksName>` with name of your AKS cluster, and replace `<acrName>` with the name of your ACR registry:

```
# Get the service principal ID of your AKS cluster
AKS_SP_ID=$(az aks show --resource-group <resourceGroupName> --name <aksName> --query
"servicePrincipalProfile.clientId" -o tsv)

# Get the resource ID of your ACR instance
ACR_RESOURCE_ID=$(az acr show --resource-group <resourceGroupName> --name <acrName> --query "id" -o tsv)

# Create a role assignment for your AKS cluster to access the ACR instance
az role assignment create --assignee $AKS_SP_ID --scope $ACR_RESOURCE_ID --role contributor
```

For more information on these steps to access ACR, see authenticating with ACR.

Configure Draft to push to and deploy from ACR

Now that there is a trust relationship between AKS and ACR, enable the use of ACR from your AKS cluster.

1. Set the Draft configuration *registry* value. In the following commands, replace `<acrName>` with the name of your ACR registry:

   ```
   draft config set registry <acrName>.azurecr.io
   ```

2. Log on to the ACR registry with az acr login:

   ```
   az acr login --name <acrName>
   ```

As a trust was created between AKS and ACR, no passwords or secrets are required to push to or pull from the ACR registry. Authentication happens at the Azure Resource Manager level, using Azure Active Directory.

Run an application

To see Draft in action, let's deploy a sample application from the Draft repository. First, clone the repo:

```
git clone https://github.com/Azure/draft
```

Change to the Java examples directory:

```
cd draft/examples/example-java/
```

Use the `draft create` command to start the process. This command creates the artifacts that are used to run the application in a Kubernetes cluster. These items include a Dockerfile, a Helm chart, and a *draft.toml* file, which is the Draft configuration file.

```
$ draft create

--> Draft detected Java (92.205567%)
--> Ready to sail
```

To run the sample application in your AKS cluster, use the `draft up` command. This command builds the Dockerfile to create a container image, pushes the image to ACR, and finally installs the Helm chart to start the application in AKS.

The first time this command is run, pushing and pulling the container image may take some time. Once the base layers are cached, the time taken to deploy the application is dramatically reduced.

```
$ draft up

Draft Up Started: 'example-java': 01CMZAR1F4T1TJZ8SWJQ70HCNH
example-java: Building Docker Image: SUCCESS ⍰  (73.0720s)
example-java: Pushing Docker Image: SUCCESS ⍰  (19.5727s)
example-java: Releasing Application: SUCCESS ⍰  (4.6979s)
Inspect the logs with `draft logs 01CMZAR1F4T1TJZ8SWJQ70HCNH`
```

If you encounter issues pushing the Docker image, ensure that you have successfully logged in to your ACR registry with az acr login, then try the `draft up` command again.

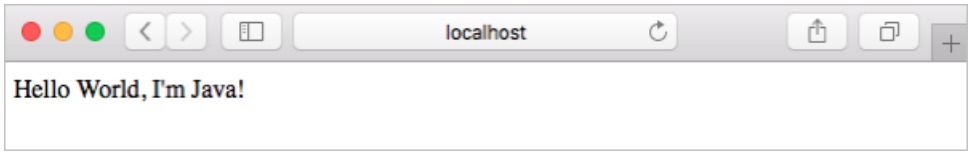
Test the application locally

To test the application, use the `draft connect` command. This command proxies a secure connection to the Kubernetes pod. When complete, the application can be accessed on the provided URL.

> **NOTE**
> It may take a few minutes for the container image to be downloaded and the application to start. If you receive an error when accessing the application, retry the connection.

```
$ draft connect

Connect to java:4567 on localhost:49804
[java]: SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
[java]: SLF4J: Defaulting to no-operation (NOP) logger implementation
[java]: SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
[java]: == Spark has ignited ...
[java]: >> Listening on 0.0.0.0:4567
```

To access your application, open a web browser to the address and port specified in the `draft connect` output, such as *http://localhost:49804*.



Use `Control+C` to stop the proxy connection.

> **NOTE**
> You can also use the `draft up --auto-connect` command to build and deploy your application then immediately connect to the first running container.

Access the application on the internet

The previous step created a proxy connection to the application pod in your AKS cluster. As you develop and test your application, you may want to make the application available on the internet. To expose an application on the internet, you create a Kubernetes service with a type of LoadBalancer, or create an ingress controller. Let's create a *LoadBalancer* service.

First, update the *values.yaml* Draft pack to specify that a service with a type *LoadBalancer* should be created:

```
vi charts/java/values.yaml
```

Locate the *service.type* property and update the value from *ClusterIP* to *LoadBalancer*, as shown in the following condensed example:

```
[...]
service:
  name: java
  type: LoadBalancer
  externalPort: 80
  internalPort: 4567
[...]
```

Save and close the file, then use `draft up` to rerun the application:

```
draft up
```

It takes a few minutes for the service to return a public IP address. To monitor the progress, use the `kubectl get service` command with the *watch* parameter:

```
kubectl get service --watch
```

Initially, the *EXTERNAL-IP* for the service appears as *pending*:

```
NAME                TYPE          CLUSTER-IP    EXTERNAL-IP   PORT(S)        AGE
example-java-java   LoadBalancer  10.0.141.72   <pending>     80:32150/TCP   2m
```

Once the EXTERNAL-IP address has changed from *pending* to an IP address, use `Control+C` to stop the `kubectl` watch process:

```
NAME                TYPE           CLUSTER-IP    EXTERNAL-IP     PORT(S)        AGE
example-java-java   LoadBalancer   10.0.141.72   52.175.224.118  80:32150/TCP   7m
```

To see the application, browse to the external IP address of your load balancer with `curl`:

```
$ curl 52.175.224.118

Hello World, I'm Java
```

Iterate on the application

Now that Draft has been configured and the application is running in Kubernetes, you are set for code iteration. Each time you want to test updated code, run the `draft up` command to update the running application.

In this example, update the Java sample application to change the display text. Open the *Hello.java* file:

```
vi src/main/java/helloworld/Hello.java
```

Update the output text to display, *Hello World, I'm Java in AKS!*:

```
package helloworld;

import static spark.Spark.*;

public class Hello {
    public static void main(String[] args) {
        get("/", (req, res) -> "Hello World, I'm Java in AKS!");
    }
}
```

Run the `draft up` command to redeploy the application:

```
$ draft up

Draft Up Started: 'example-java': 01CMZC9RF0TZT7XPWGFCJE15X4
example-java: Building Docker Image: SUCCESS ⚓  (25.0202s)
example-java: Pushing Docker Image: SUCCESS ⚓  (7.1457s)
example-java: Releasing Application: SUCCESS ⚓  (3.5773s)
Inspect the logs with `draft logs 01CMZC9RF0TZT7XPWGFCJE15X4`
```

To see the updated application, curl the IP address of your load balancer again:

```
$ curl 52.175.224.118

Hello World, I'm Java in AKS!
```

Next steps

For more information about using Draft, see the Draft documentation on GitHub.

Draft documentation

Integrate with Azure-managed services using Open Service Broker for Azure (OSBA)

7/12/2018 • 3 minutes to read • Edit Online

Together with the Kubernetes Service Catalog, Open Service Broker for Azure (OSBA) allows developers to utilize Azure-managed services in Kubernetes. This guide focuses on deploying Kubernetes Service Catalog, Open Service Broker for Azure (OSBA), and applications that use Azure-managed services using Kubernetes.

Prerequisites

- An Azure subscription
- Azure CLI: install it locally, or use it in the Azure Cloud Shell.
- Helm CLI 2.7+: install it locally, or use it in the Azure Cloud Shell.
- Permissions to create a service principal with the Contributor role on your Azure subscription
- An existing Azure Kubernetes Service (AKS) cluster. If you need an AKS cluster, follow theCreate an AKS cluster quickstart.

Install Service Catalog

The first step is to install Service Catalog in your Kubernetes cluster using a Helm chart. Upgrade your Tiller (Helm server) installation in your cluster with:

```
helm init --upgrade
```

Now, add the Service Catalog chart to the Helm repository:

```
helm repo add svc-cat https://svc-catalog-charts.storage.googleapis.com
```

Finally, install Service Catalog with the Helm chart. If your cluster is RBAC-enabled, run this command.

```
helm install svc-cat/catalog --name catalog --namespace catalog --set
controllerManager.healthcheck.enabled=false
```

If your cluster is not RBAC-enabled, run this command.

```
helm install svc-cat/catalog --name catalog --namespace catalog --set rbacEnable=false --set
controllerManager.healthcheck.enabled=false
```

After the Helm chart has been run, verify that `servicecatalog` appears in the output of the following command:

```
kubectl get apiservice
```

For example, you should see output similar to the following (show here truncated):

```
NAME                                 AGE
v1.                                  10m
v1.authentication.k8s.io             10m
...
v1beta1.servicecatalog.k8s.io        34s
v1beta1.storage.k8s.io               10
```

Install Open Service Broker for Azure

The next step is to install Open Service Broker for Azure, which includes the catalog for the Azure-managed services. Examples of available Azure services are Azure Database for PostgreSQL, Azure Database for MySQL, and Azure SQL Database.

Start by adding the Open Service Broker for Azure Helm repository:

```
helm repo add azure https://kubernetescharts.blob.core.windows.net/azure
```

Create a Service Principal with the following Azure CLI command:

```
az ad sp create-for-rbac
```

Output should be similar to the following. Take note of the `appId`, `password`, and `tenant` values, which you use in the next step.

```
{
  "appId": "7248f250-0000-0000-0000-dbdeb8400d85",
  "displayName": "azure-cli-2017-10-15-02-20-15",
  "name": "http://azure-cli-2017-10-15-02-20-15",
  "password": "77851d2c-0000-0000-0000-cb3ebc97975a",
  "tenant": "72f988bf-0000-0000-0000-2d7cd011db47"
}
```

Set the following environment variables with the preceding values:

```
AZURE_CLIENT_ID=<appId>
AZURE_CLIENT_SECRET=<password>
AZURE_TENANT_ID=<tenant>
```

Now, get your Azure subscription ID:

```
az account show --query id --output tsv
```

Again, set the following environment variable with the preceding value:

```
AZURE_SUBSCRIPTION_ID=[your Azure subscription ID from above]
```

Now that you've populated these environment variables, execute the following command to install the Open Service Broker for Azure using the Helm chart:

```
helm install azure/open-service-broker-azure --name osba --namespace osba \
    --set azure.subscriptionId=$AZURE_SUBSCRIPTION_ID \
    --set azure.tenantId=$AZURE_TENANT_ID \
    --set azure.clientId=$AZURE_CLIENT_ID \
    --set azure.clientSecret=$AZURE_CLIENT_SECRET
```

Once the OSBA deployment is complete, install the Service Catalog CLI, an easy-to-use command-line interface for querying service brokers, service classes, service plans, and more.

Execute the following commands to install the Service Catalog CLI binary:

```
curl -sLO https://servicecatalogcli.blob.core.windows.net/cli/latest/$(uname -s)/$(uname -m)/svcat
chmod +x ./svcat
```

Now, list installed service brokers:

```
./svcat get brokers
```

You should see output similar to the following:

```
  NAME                              URL                                  STATUS
+------+-----------------------------------------------------------------+--------+
  osba   http://osba-open-service-broker-azure.osba.svc.cluster.local   Ready
```

Next, list the available service classes. The displayed service classes are the available Azure-managed services that can be provisioned through Open Service Broker for Azure.

```
./svcat get classes
```

Finally, list all available service plans. Service plans are the service tiers for the Azure-managed services. For example, for Azure Database for MySQL, plans range from `basic50` for Basic tier with 50 Database Transaction Units (DTUs), to `standard800` for Standard tier with 800 DTUs.

```
./svcat get plans
```

Install WordPress from Helm chart using Azure Database for MySQL

In this step, you use Helm to install an updated Helm chart for WordPress. The chart provisions an external Azure Database for MySQL instance that WordPress can use. This process can take a few minutes.

```
helm install azure/wordpress --name wordpress --namespace wordpress --set resources.requests.cpu=0
```

In order to verify the installation has provisioned the right resources, list the installed service instances and bindings:

```
./svcat get instances -n wordpress
./svcat get bindings -n wordpress
```

List installed secrets:

```
kubectl get secrets -n wordpress -o yaml
```

Next steps

By following this article, you deployed Service Catalog to an Azure Kubernetes Service (AKS) cluster. You used Open Service Broker for Azure to deploy a WordPress installation that uses Azure-managed services, in this case Azure Database for MySQL.

Refer to the Azure/helm-charts repository to access other updated OSBA-based Helm charts. If you're interested in creating your own charts that work with OSBA, refer to Creating a New Chart.

Using OpenFaaS on AKS

9/13/2018 • 4 minutes to read • Edit Online

OpenFaaS is a framework for building serverless functions on top of containers. As an open source project, it has gained large-scale adoption within the community. This document details installing and using OpenFaas on an Azure Kubernetes Service (AKS) cluster.

Prerequisites

In order to complete the steps within this article, you need the following.

- Basic understanding of Kubernetes.
- An Azure Kubernetes Service (AKS) cluster and AKS credentials configured on your development system.
- Azure CLI installed on your development system.
- Git command-line tools installed on your system.

Get OpenFaaS

Clone the OpenFaaS project repository to your development system.

```
git clone https://github.com/openfaas/faas-netes
```

Change into the directory of the cloned repository.

```
cd faas-netes
```

Deploy OpenFaaS

As a good practice, OpenFaaS and OpenFaaS functions should be stored in their own Kubernetes namespace.

Create a namespace for the OpenFaaS system.

```
kubectl create namespace openfaas
```

Create a second namespace for OpenFaaS functions.

```
kubectl create namespace openfaas-fn
```

A Helm chart for OpenFaaS is included in the cloned repository. Use this chart to deploy OpenFaaS into your AKS cluster.

```
helm install --namespace openfaas -n openfaas \
  --set functionNamespace=openfaas-fn, \
  --set serviceType=LoadBalancer, \
  --set rbac=false chart/openfaas/
```

Output:

```
NAME:   openfaas
LAST DEPLOYED: Wed Feb 28 08:26:11 2018
NAMESPACE: openfaas
STATUS: DEPLOYED

RESOURCES:
==> v1/ConfigMap
NAME                 DATA  AGE
prometheus-config    2     20s
alertmanager-config  1     20s

{snip}

NOTES:
To verify that openfaas has started, run:

  kubectl --namespace=openfaas get deployments -l "release=openfaas, app=openfaas"
```

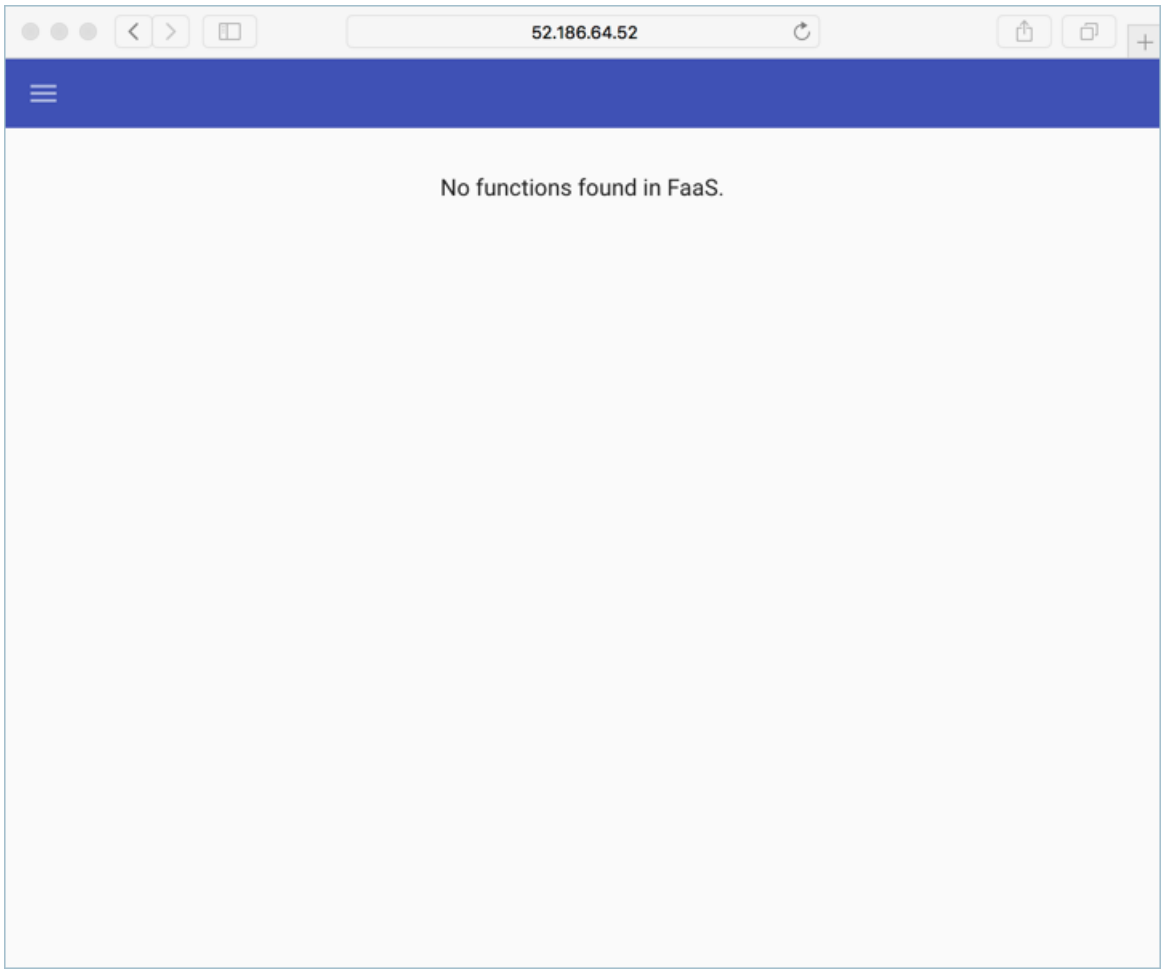
A public IP address is created for accessing the OpenFaaS gateway. To retrieve this IP address, use the kubectl get service command. It may take a minute for the IP address to be assigned to the service.

```
kubectl get service -l component=gateway --namespace openfaas
```

Output.

```
NAME               TYPE           CLUSTER-IP     EXTERNAL-IP    PORT(S)          AGE
gateway            ClusterIP      10.0.156.194   <none>         8080/TCP         7m
gateway-external   LoadBalancer   10.0.28.18     52.186.64.52   8080:30800/TCP   7m
```

To test the OpenFaaS system, browse to the external IP address on port 8080, `http://52.186.64.52:8080` in this example.



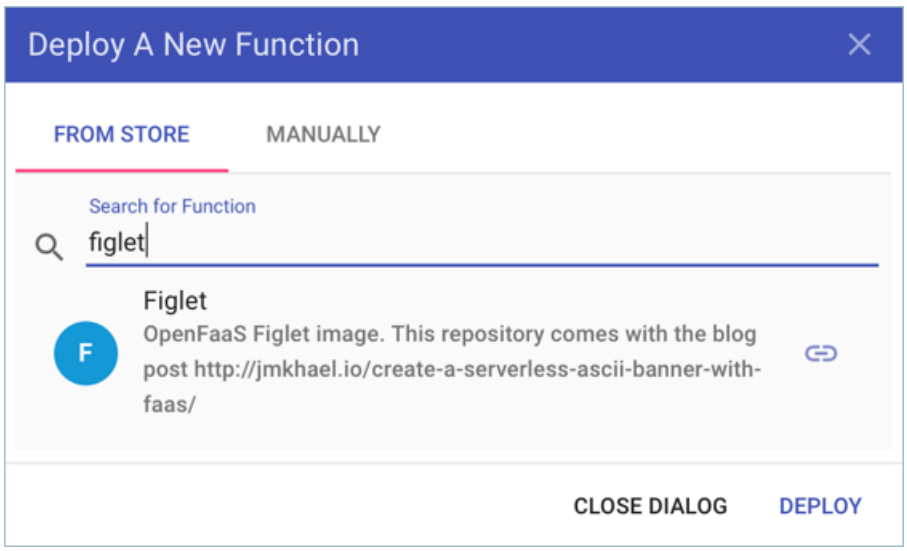
Finally, install the OpenFaaS CLI. This example used brew, see the OpenFaaS CLI documentation for more options.

```
brew install faas-cli
```

Create first function

Now that OpenFaaS is operational, create a function using the OpenFaas portal.

Click on **Deploy New Function** and search for **Figlet**. Select the Figlet function, and click **Deploy**.



Use curl to invoke the function. Replace the IP address in the following example with that of your OpenFaas gateway.

```
curl -X POST http://52.186.64.52:8080/function/figlet -d "Hello Azure"
```

Output:

```
 _   _      _ _                 _
| | | | ___| | | ___      / \    _____   _ _ __ ___
| |_| |/ _ \ | |/ _ \    / _ \  |_  / | | | '__/ _ \
|  _  |  __/ | | (_) |  / ___ \  / /| |_| | | |  __/
|_| |_|\___|_|_|\___/  /_/   \_\/___|\__,_|_|  \___|
```

Create second function

Now create a second function. This example will be deployed using the OpenFaaS CLI and includes a custom container image and retrieving data from a Cosmos DB. Several items need to be configured before creating the function.

First, create a new resource group for the Cosmos DB.

```
az group create --name serverless-backing --location eastus
```

Deploy a CosmosDB instance of kind `MongoDB`. The instance needs a unique name, update `openfaas-cosmos` to something unique to your environment.

```
az cosmosdb create --resource-group serverless-backing --name openfaas-cosmos --kind MongoDB
```

Get the Cosmos database connection string and store it in a variable.

Update the value for the `--resource-group` argument to the name of your resource group, and the `--name` argument to the name of your Cosmos DB.

```
COSMOS=$(az cosmosdb list-connection-strings \
  --resource-group serverless-backing \
  --name openfaas-cosmos \
  --query connectionStrings[0].connectionString \
  --output tsv)
```

Now populate the Cosmos DB with test data. Create a file named `plans.json` and copy in the following json.

```
{
    "name" : "two_person",
    "friendlyName" : "Two Person Plan",
    "portionSize" : "1-2 Person",
    "mealsPerWeek" : "3 Unique meals per week",
    "price" : 72,
    "description" : "Our basic plan, delivering 3 meals per week, which will feed 1-2 people.",
    "__v" : 0
}
```

Use the *mongoimport* tool to load the CosmosDB instance with data.

If needed, install the MongoDB tools. The following example installs these tools using brew, see the MongoDB documentation for other options.

```
brew install mongodb
```

Load the data into the database.

```
mongoimport --uri=$COSMOS -c plans < plans.json
```

Output:

```
2018-02-19T14:42:14.313+0000    connected to: localhost
2018-02-19T14:42:14.918+0000    imported 1 document
```

Run the following command to create the function. Update the value of the `-g` argument with your OpenFaaS gateway address.

```
faas-cli deploy -g http://52.186.64.52:8080 --image=shanepeckham/openfaascosmos --name=cosmos-query --
env=NODE_ENV=$COSMOS
```

Once deployed, you should see your newly created OpenFaaS endpoint for the function.

```
Deployed. 202 Accepted.
URL: http://52.186.64.52:8080/function/cosmos-query
```

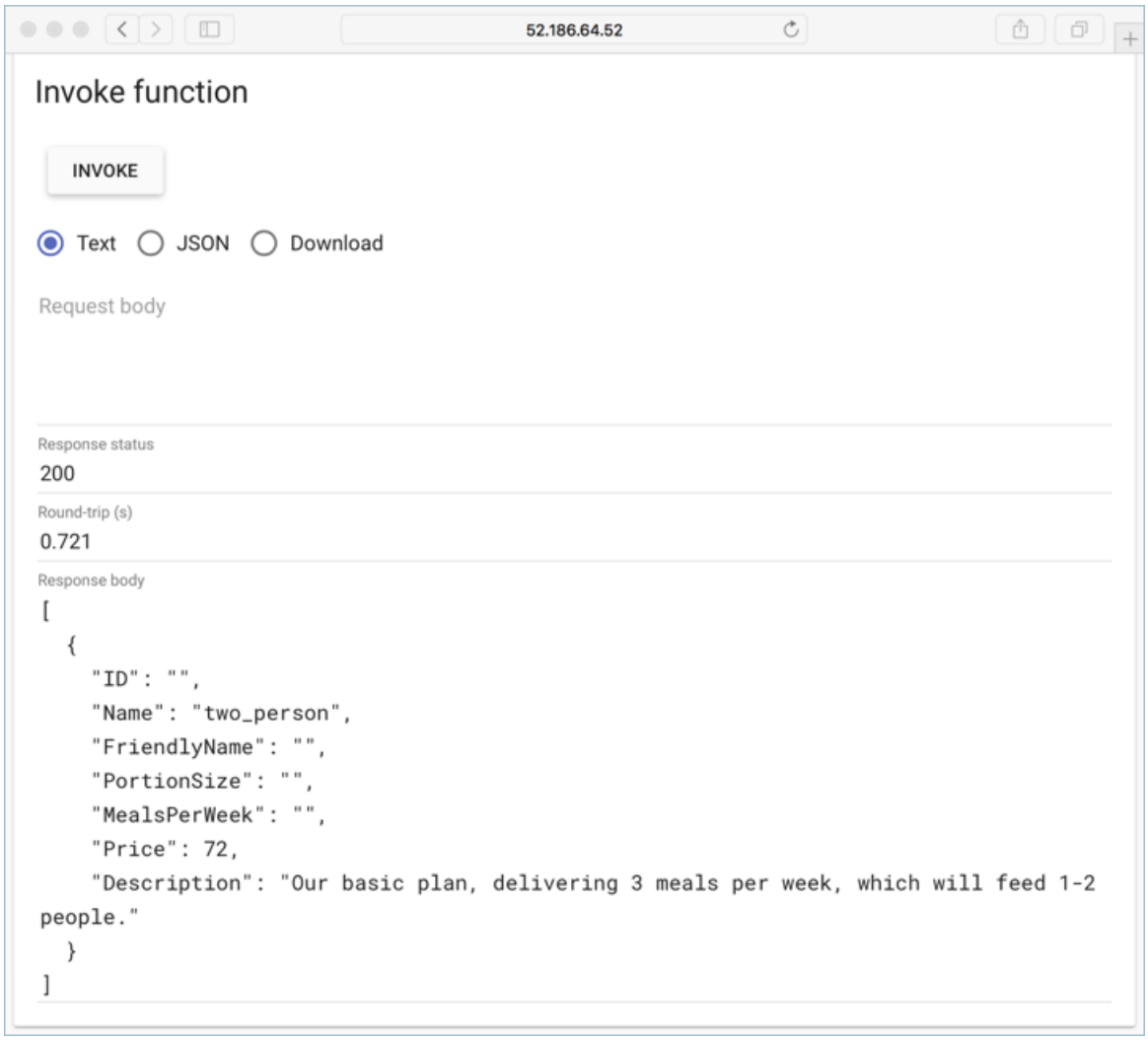
Test the function using curl. Update the IP address with the OpenFaaS gateway address.

```
curl -s http://52.186.64.52:8080/function/cosmos-query
```

Output:

```
[{"ID":"","Name":"two_person","FriendlyName":"","PortionSize":"","MealsPerWeek":"","Price":72,"Description":"Ou
r basic plan, delivering 3 meals per week, which will feed 1-2 people."}]
```

You can also test the function within the OpenFaaS UI.



Next Steps

The default deployment of OpenFaas needs to be locked down for both OpenFaaS Gateway and Functions. Alex Ellis' Blog post has more details on secure configuration options.

Running Apache Spark jobs on AKS

5/10/2018 • 6 minutes to read • Edit Online

Apache Spark is a fast engine for large-scale data processing. As of the Spark 2.3.0 release, Apache Spark supports native integration with Kubernetes clusters. Azure Kubernetes Service (AKS) is a managed Kubernetes environment running in Azure. This document details preparing and running Apache Spark jobs on an Azure Kubernetes Service (AKS) cluster.

Prerequisites

In order to complete the steps within this article, you need the following.

- Basic understanding of Kubernetes and Apache Spark.
- Docker Hub account, or an Azure Container Registry.
- Azure CLI installed on your development system.
- JDK 8 installed on your system.
- SBT (Scala Build Tool) installed on your system.
- Git command-line tools installed on your system.

Create an AKS cluster

Spark is used for large-scale data processing and requires that Kubernetes nodes are sized to meet the Spark resources requirements. We recommend a minimum size of `Standard_D3_v2` for your Azure Kubernetes Service (AKS) nodes.

If you need an AKS cluster that meets this minimum recommendation, run the following commands.

Create a resource group for the cluster.

```
az group create --name mySparkCluster --location eastus
```

Create the AKS cluster with nodes that are of size `Standard_D3_v2`.

```
az aks create --resource-group mySparkCluster --name mySparkCluster --node-vm-size Standard_D3_v2
```

Connect to the AKS cluster.

```
az aks get-credentials --resource-group mySparkCluster --name mySparkCluster
```

If you are using Azure Container Registry (ACR) to store container images, configure authentication between AKS and ACR. See the ACR authentication documentation for these steps.

Build the Spark source

Before running Spark jobs on an AKS cluster, you need to build the Spark source code and package it into a container image. The Spark source includes scripts that can be used to complete this process.

Clone the Spark project repository to your development system.

```
git clone -b branch-2.3 https://github.com/apache/spark
```

Change into the directory of the cloned repository and save the path of the Spark source to a variable.

```
cd spark
sparkdir=$(pwd)
```

If you have multiple JDK versions installed, set `JAVA_HOME` to use version 8 for the current session.

```
export JAVA_HOME=`/usr/libexec/java_home -d 64 -v "1.8*"`
```

Run the following command to build the Spark source code with Kubernetes support.

```
./build/mvn -Pkubernetes -DskipTests clean package
```

The following commands create the Spark container image and push it to a container image registry. Replace `registry.example.com` with the name of your container registry and `v1` with the tag you prefer to use. If using Docker Hub, this value is the registry name. If using Azure Container Registry (ACR), this value is the ACR login server name.

```
REGISTRY_NAME=registry.example.com
REGISTRY_TAG=v1
```

```
./bin/docker-image-tool.sh -r $REGISTRY_NAME -t $REGISTRY_TAG build
```

Push the container image to your container image registry.

```
./bin/docker-image-tool.sh -r $REGISTRY_NAME -t $REGISTRY_TAG push
```

Prepare a Spark job

Next, prepare a Spark job. A jar file is used to hold the Spark job and is needed when running the `spark-submit` command. The jar can be made accessible through a public URL or pre-packaged within a container image. In this example, a sample jar is created to calculate the value of Pi. This jar is then uploaded to Azure storage. If you have an existing jar, feel free to substitute

Create a directory where you would like to create the project for a Spark job.

```
mkdir myprojects
cd myprojects
```

Create a new Scala project from a template.

```
sbt new sbt/scala-seed.g8
```

When prompted, enter `SparkPi` for the project name.

```
name [Scala Seed Project]: SparkPi
```

Navigate to the newly created project directory.

```
cd sparkpi
```

Run the following commands to add an SBT plugin, which allows packaging the project as a jar file.

```
touch project/assembly.sbt
echo 'addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.14.6")' >> project/assembly.sbt
```

Run these commands to copy the sample code into the newly created project and add all necessary dependencies.

```
EXAMPLESDIR="src/main/scala/org/apache/spark/examples"
mkdir -p $EXAMPLESDIR
cp $sparkdir/examples/$EXAMPLESDIR/SparkPi.scala $EXAMPLESDIR/SparkPi.scala

cat <<EOT >> build.sbt
// https://mvnrepository.com/artifact/org.apache.spark/spark-sql
libraryDependencies += "org.apache.spark" %% "spark-sql" % "2.3.0" % "provided"
EOT

sed -ie 's/scalaVersion.*/scalaVersion := "2.11.11",/' build.sbt
sed -ie 's/name.*/name := "SparkPi",/' build.sbt
```

To package the project into a jar, run the following command.

```
sbt assembly
```

After successful packaging, you should see output similar to the following.

```
[info] Packaging /Users/me/myprojects/sparkpi/target/scala-2.11/SparkPi-assembly-0.1.0-SNAPSHOT.jar ...
[info] Done packaging.
[success] Total time: 10 s, completed Mar 6, 2018 11:07:54 AM
```

Copy job to storage

Create an Azure storage account and container to hold the jar file.

```
RESOURCE_GROUP=sparkdemo
STORAGE_ACCT=sparkdemo$RANDOM
az group create --name $RESOURCE_GROUP --location eastus
az storage account create --resource-group $RESOURCE_GROUP --name $STORAGE_ACCT --sku Standard_LRS
export AZURE_STORAGE_CONNECTION_STRING=`az storage account show-connection-string --resource-group
$RESOURCE_GROUP --name $STORAGE_ACCT -o tsv`
```

Upload the jar file to the Azure storage account with the following commands.

```
CONTAINER_NAME=jars
BLOB_NAME=SparkPi-assembly-0.1.0-SNAPSHOT.jar
FILE_TO_UPLOAD=target/scala-2.11/SparkPi-assembly-0.1.0-SNAPSHOT.jar

echo "Creating the container..."
az storage container create --name $CONTAINER_NAME
az storage container set-permission --name $CONTAINER_NAME --public-access blob

echo "Uploading the file..."
az storage blob upload --container-name $CONTAINER_NAME --file $FILE_TO_UPLOAD --name $BLOB_NAME

jarUrl=$(az storage blob url --container-name $CONTAINER_NAME --name $BLOB_NAME | tr -d '"')
```

Variable `jarUrl` now contains the publicly accessible path to the jar file.

Submit a Spark job

Start kube-proxy in a separate command-line with the following code.

```
kubectl proxy
```

Navigate back to the root of Spark repository.

```
cd $sparkdir
```

Submit the job using `spark-submit`.

```
./bin/spark-submit \
  --master k8s://http://127.0.0.1:8001 \
  --deploy-mode cluster \
  --name spark-pi \
  --class org.apache.spark.examples.SparkPi \
  --conf spark.executor.instances=3 \
  --conf spark.kubernetes.container.image=$REGISTRY_NAME/spark:$REGISTRY_TAG \
  $jarUrl
```

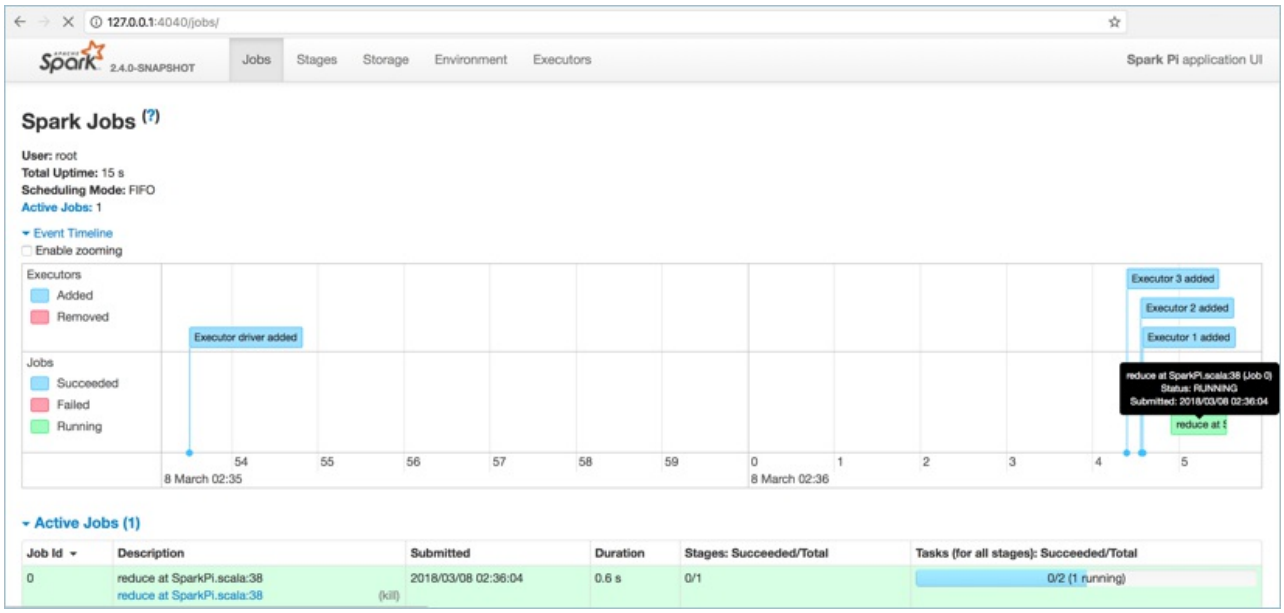
This operation starts the Spark job, which streams job status to your shell session. While the job is running, you can see Spark driver pod and executor pods using the kubectl get pods command. Open a second terminal session to run these commands.

```
$ kubectl get pods

NAME                                            READY     STATUS     RESTARTS   AGE
spark-pi-2232778d0f663768ab27edc35cb73040-driver   1/1       Running    0          16s
spark-pi-2232778d0f663768ab27edc35cb73040-exec-1   0/1       Init:0/1   0          4s
spark-pi-2232778d0f663768ab27edc35cb73040-exec-2   0/1       Init:0/1   0          4s
spark-pi-2232778d0f663768ab27edc35cb73040-exec-3   0/1       Init:0/1   0          4s
```

While the job is running, you can also access the Spark UI. In the second terminal session, use the `kubectl port-forward` command provide access to Spark UI.

```
kubectl port-forward spark-pi-2232778d0f663768ab27edc35cb73040-driver 4040:4040
```

To access Spark UI, open the address `127.0.0.1:4040` in a browser.



Get job results and logs

After the job has finished, the driver pod will be in a "Completed" state. Get the name of the pod with the following command.

```
kubectl get pods --show-all
```

Output:

```
NAME                                              READY     STATUS      RESTARTS   AGE
spark-pi-2232778d0f663768ab27edc35cb73040-driver   0/1       Completed   0          1m
```

Use the `kubectl logs` command to get logs from the spark driver pod. Replace the pod name with your driver pod's name.

```
kubectl logs spark-pi-2232778d0f663768ab27edc35cb73040-driver
```

Within these logs, you can see the result of the Spark job, which is the value of Pi.

```
Pi is roughly 3.152155760778804
```

Package jar with container image

In the above example, the Spark jar file was uploaded to Azure storage. Another option is to package the jar file into custom-built Docker images.

To do so, find the `dockerfile` for the Spark image located at `$sparkdir/resource-managers/kubernetes/docker/src/main/dockerfiles/spark/` directory. Add am `ADD` statement for the Spark job `jar` somewhere between `WORKDIR` and `ENTRYPOINT` declarations.

Update the jar path to the location of the `SparkPi-assembly-0.1.0-SNAPSHOT.jar` file on your development system. You can also use your own custom jar file.

```
WORKDIR /opt/spark/work-dir

ADD /path/to/SparkPi-assembly-0.1.0-SNAPSHOT.jar SparkPi-assembly-0.1.0-SNAPSHOT.jar

ENTRYPOINT [ "/opt/entrypoint.sh" ]
```

Build and push the image with the included Spark scripts.

```
./bin/docker-image-tool.sh -r <your container repository name> -t <tag> build
./bin/docker-image-tool.sh -r <your container repository name> -t <tag> push
```

When running the job, instead of indicating a remote jar URL, the `local://` scheme can be used with the path to the jar file in the Docker image.

```
./bin/spark-submit \
    --master k8s://https://<k8s-apiserver-host>:<k8s-apiserver-port> \
    --deploy-mode cluster \
    --name spark-pi \
    --class org.apache.spark.examples.SparkPi \
    --conf spark.executor.instances=3 \
    --conf spark.kubernetes.container.image=<spark-image> \
    local:///opt/spark/work-dir/<your-jar-name>.jar
```

WARNING

From Spark documentation: "The Kubernetes scheduler is currently experimental. In future versions, there may be behavioral changes around configuration, container images and entrypoints".

Next steps

Check out Spark documentation for more details.

Spark documentation

Using GPUs on AKS

10/2/2018 • 7 minutes to read • Edit Online

AKS supports the creation of GPU enabled node pools. Azure currently provides single or multiple GPU enabled VMs. GPU enabled VMs are designed for compute-intensive, graphics-intensive, and visualization workloads. A list of GPU enabled VMs can be found here.

Create an AKS cluster

GPUs are typically needed for compute-intensive workloads such as graphics-intensive, and visualization workloads. Refer to the following document to determine the right VM size for your workload. We recommend a minimum size of `Standard_NC6` for your Azure Kubernetes Service (AKS) nodes.

> **NOTE**
> GPU enabled VMs contain specialized hardware that is subject to higher pricing and region availability. For more information, see the pricing tool and region availability site for more information.

If you need an AKS cluster that meets this minimum recommendation, run the following commands.

Create a resource group for the cluster.

```
az group create --name myGPUCluster --location eastus
```

Create the AKS cluster with nodes that are of size `Standard_NC6`.

```
az aks create --resource-group myGPUCluster --name myGPUCluster --node-vm-size Standard_NC6
```

Connect to the AKS cluster.

```
az aks get-credentials --resource-group myGPUCluster --name myGPUCluster
```

Confirm GPUs are schedulable

Run the following commands to confirm the GPUs are schedulable via Kubernetes.

Get the current list of nodes.

```
$ kubectl get nodes
NAME                       STATUS    ROLES     AGE       VERSION
aks-nodepool1-22139053-0   Ready     agent     10h       v1.9.6
aks-nodepool1-22139053-1   Ready     agent     10h       v1.9.6
aks-nodepool1-22139053-2   Ready     agent     10h       v1.9.6
```

Describe one of the nodes to confirm the GPUs are schedulable. This can be found under the `Capacity` section. For example, `nvidia.com/gpu: 1`. If you do not see the GPUs, consult the **Troubleshoot** section below.

```
$ kubectl describe node aks-nodepool1-22139053-0
Name:               aks-nodepool1-22139053-0
```

```
Roles:              agent
Labels:             agentpool=nodepool1
                    beta.kubernetes.io/arch=amd64
                    beta.kubernetes.io/instance-type=Standard_NC6
                    beta.kubernetes.io/os=linux
                    failure-domain.beta.kubernetes.io/region=eastus
                    failure-domain.beta.kubernetes.io/zone=1
                    kubernetes.azure.com/cluster=MC_myGPUCluster_myGPUCluster
                    kubernetes.io/hostname=aks-nodepool1-22139053-0
                    kubernetes.io/role=agent
                    storageprofile=managed
                    storagetier=Standard_LRS
Annotations:        node.alpha.kubernetes.io/ttl=0
                    volumes.kubernetes.io/controller-managed-attach-detach=true
Taints:             <none>
CreationTimestamp:  Thu, 05 Apr 2018 12:13:20 -0700
Conditions:
  Type                 Status  LastHeartbeatTime                 LastTransitionTime                Reason
Message
  ----                 ------  -----------------                 ------------------                ------
-------
  NetworkUnavailable   False   Thu, 05 Apr 2018 12:15:07 -0700   Thu, 05 Apr 2018 12:15:07 -0700   RouteCreated
RouteController created a route
  OutOfDisk            False   Thu, 05 Apr 2018 22:14:33 -0700   Thu, 05 Apr 2018 12:13:20 -0700
KubeletHasSufficientDisk     kubelet has sufficient disk space available
  MemoryPressure       False   Thu, 05 Apr 2018 22:14:33 -0700   Thu, 05 Apr 2018 12:13:20 -0700
KubeletHasSufficientMemory   kubelet has sufficient memory available
  DiskPressure         False   Thu, 05 Apr 2018 22:14:33 -0700   Thu, 05 Apr 2018 12:13:20 -0700
KubeletHasNoDiskPressure     kubelet has no disk pressure
  Ready                True    Thu, 05 Apr 2018 22:14:33 -0700   Thu, 05 Apr 2018 12:15:10 -0700   KubeletReady
kubelet is posting ready status. AppArmor enabled
Addresses:
  InternalIP:  10.240.0.4
  Hostname:    aks-nodepool1-22139053-0
Capacity:
 nvidia.com/gpu:                 1
 cpu:                            6
 memory:                         57691688Ki
 pods:                           110
Allocatable:
 nvidia.com/gpu:                 1
 cpu:                            6
 memory:                         57589288Ki
 pods:                           110
System Info:
 Machine ID:                 2eb0e90bd1fe450ba3cf83479443a511
 System UUID:                CFB485B6-CB49-A545-A2C9-8E4C592C3273
 Boot ID:                    fea24544-596d-4246-b8c3-610fc7ac7280
 Kernel Version:             4.13.0-1011-azure
 OS Image:                   Debian GNU/Linux 9 (stretch)
 Operating System:           linux
 Architecture:               amd64
 Container Runtime Version:  docker://1.13.1
 Kubelet Version:            v1.9.6
 Kube-Proxy Version:         v1.9.6
PodCIDR:                     10.244.1.0/24
ExternalID:                  /subscriptions/8ecadfc9-d1a3-4ea4-b844-
0d9f87e4d7c8/resourceGroups/MC_myGPUCluster_myGPUCluster/providers/Microsoft.Compute/virtualMachines/aks-
nodepool1-22139053-0
Non-terminated Pods:         (2 in total)
  Namespace                  Name                       CPU Requests  CPU Limits  Memory Requests  Memory
Limits
  ---------                  ----                       ------------  ----------  ---------------  ------------
-
  kube-system                kube-proxy-pwffr           100m (1%)     0 (0%)      0 (0%)           0 (0%)
  kube-system                kube-svc-redirect-mkpf4    0 (0%)        0 (0%)      0 (0%)           0 (0%)
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  CPU Requests  CPU Limits  Memory Requests  Memory Limits
```

```
  CPU Requests  CPU Limits  Memory Requests  Memory Limits
  ------------  ----------  ---------------  -------------
  100m (1%)     0 (0%)      0 (0%)           0 (0%)
Events:         <none>
```

Run a GPU enabled workload

In order to demonstrate the GPUs are indeed working, schedule a GPU enabled workload with the appropriate resource request. This example will run a Tensorflow job against the MNIST dataset.

The following job manifest includes a resource limit of `nvidia.com/gpu: 1`.

Copy the manifest and save as **samples-tf-mnist-demo.yaml**.

```
apiVersion: batch/v1
kind: Job
metadata:
  labels:
    app: samples-tf-mnist-demo
  name: samples-tf-mnist-demo
spec:
  template:
    metadata:
      labels:
        app: samples-tf-mnist-demo
    spec:
      containers:
      - name: samples-tf-mnist-demo
        image: microsoft/samples-tf-mnist-demo:gpu
        args: ["--max_steps", "500"]
        imagePullPolicy: IfNotPresent
        resources:
          limits:
           nvidia.com/gpu: 1
      restartPolicy: OnFailure
```

Use the kubectl apply command to run the job. This command parses the manifest file and creates the defined Kubernetes objects.

```
$ kubectl apply -f samples-tf-mnist-demo.yaml
job "samples-tf-mnist-demo" created
```

Monitor the progress of the job until successful completion using the kubectl get jobs command with the `--watch` argument.

```
$ kubectl get jobs samples-tf-mnist-demo --watch
NAME                    DESIRED   SUCCESSFUL   AGE
samples-tf-mnist-demo   1         0            8s
samples-tf-mnist-demo   1         1            35s
```

Determine the pod name to view the logs by showing completed pods.

```
$ kubectl get pods --selector app=samples-tf-mnist-demo --show-all
NAME                          READY     STATUS      RESTARTS   AGE
samples-tf-mnist-demo-smnr6   0/1       Completed   0          4m
```

Using the pod name from the output of the command above, refer to the pod logs to confirm that the appropriate GPU device has been discovered in this case, `Tesla K80`.

```
$ kubectl logs samples-tf-mnist-demo-smnr6
2018-04-13 04:11:08.710863: I tensorflow/core/platform/cpu_feature_guard.cc:137] Your CPU supports instructions
that this TensorFlow binary was not compiled to use: SSE4.1 SSE4.2 AVX AVX2 FMA
2018-04-13 04:11:15.824349: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1030] Found device 0 with
properties:
name: Tesla K80 major: 3 minor: 7 memoryClockRate(GHz): 0.8235
pciBusID: 04e1:00:00.0
totalMemory: 11.17GiB freeMemory: 11.10GiB
2018-04-13 04:11:15.824394: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1120] Creating TensorFlow device
(/device:GPU:0) -> (device: 0, name: Tesla K80, pci bus id: 04e1:00:00.0, compute capability: 3.7)
2018-04-13 04:11:20.891910: I tensorflow/stream_executor/dso_loader.cc:139] successfully opened CUDA library
libcupti.so.8.0 locally
Successfully downloaded train-images-idx3-ubyte.gz 9912422 bytes.
Extracting /tmp/tensorflow/input_data/train-images-idx3-ubyte.gz
Successfully downloaded train-labels-idx1-ubyte.gz 28881 bytes.
Extracting /tmp/tensorflow/input_data/train-labels-idx1-ubyte.gz
Successfully downloaded t10k-images-idx3-ubyte.gz 1648877 bytes.
Extracting /tmp/tensorflow/input_data/t10k-images-idx3-ubyte.gz
Successfully downloaded t10k-labels-idx1-ubyte.gz 4542 bytes.
Extracting /tmp/tensorflow/input_data/t10k-labels-idx1-ubyte.gz
Accuracy at step 0: 0.0487
Accuracy at step 10: 0.6571
Accuracy at step 20: 0.8111
Accuracy at step 30: 0.8562
Accuracy at step 40: 0.8786
Accuracy at step 50: 0.8911
Accuracy at step 60: 0.8986
Accuracy at step 70: 0.9017
Accuracy at step 80: 0.9049
Accuracy at step 90: 0.9114
Adding run metadata for 99
Accuracy at step 100: 0.9109
Accuracy at step 110: 0.9143
Accuracy at step 120: 0.9188
Accuracy at step 130: 0.9194
Accuracy at step 140: 0.9237
Accuracy at step 150: 0.9231
Accuracy at step 160: 0.9158
Accuracy at step 170: 0.9259
Accuracy at step 180: 0.9303
Accuracy at step 190: 0.9315
Adding run metadata for 199
Accuracy at step 200: 0.9334
Accuracy at step 210: 0.9342
Accuracy at step 220: 0.9359
Accuracy at step 230: 0.9353
Accuracy at step 240: 0.933
Accuracy at step 250: 0.9353
Accuracy at step 260: 0.9408
Accuracy at step 270: 0.9396
Accuracy at step 280: 0.9406
Accuracy at step 290: 0.9444
Adding run metadata for 299
Accuracy at step 300: 0.9453
Accuracy at step 310: 0.946
Accuracy at step 320: 0.9464
Accuracy at step 330: 0.9472
Accuracy at step 340: 0.9516
Accuracy at step 350: 0.9473
Accuracy at step 360: 0.9502
Accuracy at step 370: 0.9483
Accuracy at step 380: 0.9481
Accuracy at step 390: 0.9467
Adding run metadata for 399
Accuracy at step 400: 0.9477
Accuracy at step 410: 0.948
Accuracy at step 420: 0.9496
Accuracy at step 430: 0.9501
```

```
Accuracy at step 440: 0.9534
Accuracy at step 450: 0.9551
Accuracy at step 460: 0.9518
Accuracy at step 470: 0.9562
Accuracy at step 480: 0.9583
Accuracy at step 490: 0.9575
Adding run metadata for 499
```

Cleanup

Remove the associated Kubernetes objects created in this step.

```
$ kubectl delete jobs samples-tf-mnist-demo
job "samples-tf-mnist-demo" deleted
```

Troubleshoot

In some scenarios, you might not see GPU resources under Capacity. For example: After upgrading a cluster to Kubernetes version 1.10 or creating a new Kubernetes version 1.10 cluster, the expected `nvidia.com/gpu` resource is missing from `Capacity` when running `kubectl describe node <node-name>`.

To resolve this, apply the following daemonset post provision or upgrade, then you will see `nvidia.com/gpu` as a schedulable resource.

Copy the manifest and save as **nvidia-device-plugin-ds.yaml**. For the image tag of `image: nvidia/k8s-device-plugin:1.10` below, update the tag to match your Kubernetes version. For example, use tag `1.11` for Kubernetes version 1.11.

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  labels:
    kubernetes.io/cluster-service: "true"
  name: nvidia-device-plugin
  namespace: kube-system
spec:
  template:
    metadata:
      # Mark this pod as a critical add-on; when enabled, the critical add-on scheduler
      # reserves resources for critical add-on pods so that they can be rescheduled after
      # a failure.  This annotation works in tandem with the toleration below.
      annotations:
        scheduler.alpha.kubernetes.io/critical-pod: ""
      labels:
        name: nvidia-device-plugin-ds
    spec:
      tolerations:
      # Allow this pod to be rescheduled while the node is in "critical add-ons only" mode.
      # This, along with the annotation above marks this pod as a critical add-on.
      - key: CriticalAddonsOnly
        operator: Exists
      containers:
      - image: nvidia/k8s-device-plugin:1.10 # Update this tag to match your Kubernetes version
        name: nvidia-device-plugin-ctr
        securityContext:
          allowPrivilegeEscalation: false
          capabilities:
            drop: ["ALL"]
        volumeMounts:
          - name: device-plugin
            mountPath: /var/lib/kubelet/device-plugins
      volumes:
        - name: device-plugin
          hostPath:
            path: /var/lib/kubelet/device-plugins
      nodeSelector:
        beta.kubernetes.io/os: linux
        accelerator: nvidia
```

Use the kubectl apply command to create the daemonset.

```
$ kubectl apply -f nvidia-device-plugin-ds.yaml
daemonset "nvidia-device-plugin" created
```

Next steps

Interested in running Machine Learning workloads on Kubernetes? Refer to the Kubeflow labs for more detail.

Kubeflow Labs

Create a continuous deployment pipeline with Jenkins and Azure Kubernetes Service (AKS)

10/1/2018 • 11 minutes to read • Edit Online

To quickly deploy updates to applications in Azure Kubernetes Service (AKS), you often use a continuous integration and continuous delivery (CI/CD) platform. In a CI/CD platform, a code commit can trigger a new container build that is then used to deploy an updated application instance. In this article, you use Jenkins as the CI/CD platform to build and push container images to Azure Container Registry (ACR) and then run those applications in AKS. You learn how to:

- Deploy a sample Azure vote application to an AKS cluster
- Create a basic Jenkins instance
- Configure credentials for Jenkins to interact with ACR
- Create a Jenkins build job and GitHub webhook for automated builds
- Test the CI/CD pipeline to update an application in AKS based on GitHub code commits

Before you begin

You need the following items in order to complete the steps in this article.

- Basic understanding of Kubernetes, Git, CI/CD, and container images
- An AKS cluster and `kubectl` configured with the AKS cluster credentials.
- An Azure Container Registry (ACR) registry, the ACR login server name, and the AKS cluster configured to authenticate with the ACR registry.
- The Azure CLI version 2.0.46 or later installed and configured. Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI.
- Docker installed on your development system.
- A GitHub account, GitHub personal access token, and Git client installed on your development system.

Prepare the application

In this article, you use a sample Azure vote application that contains a web interface hosted in one or more pods, and a second pod hosting Redis for temporary data storage. Before you integrate Jenkins and AKS for automated deployments, first manually prepare and deploy the Azure vote application to your AKS cluster. This manual deployment is version one of the application, and lets you see the application in action.

Fork the following GitHub repository for the sample application - https://github.com/Azure-Samples/azure-voting-app-redis. To fork the repo to your own GitHub account, select the **Fork** button in the top right-hand corner.

Clone the fork to your development system. Make sure you use the URL of your fork when cloning this repo:

```
git clone https://github.com/<your-github-account>/azure-voting-app-redis.git
```

Change to the directory of your cloned fork:

```
cd azure-voting-app-redis
```

To create the container images needed for the sample application, use the *docker-compose.yaml* file with `docker-compose`:

```
docker-compose up -d
```

The required base images are pulled and the application containers built. You can then use the docker images command to see the created image. Three images have been downloaded or created. The `azure-vote-front` image contains the application and uses the `nginx-flask` image as a base. The `redis` image is used to start a Redis instance:

```
$ docker images

REPOSITORY                    TAG        IMAGE ID            CREATED             SIZE
azure-vote-front              latest     9cc914e25834        40 seconds ago      694MB
redis                         latest     a1b99da73d05        7 days ago          106MB
tiangolo/uwsgi-nginx-flask    flask      788ca94b2313        9 months ago        694MB
```

Before you can push the *azure-vote-front* container image to ACR, get your ACR login server with the az acr list command. The following example gets the ACR login server address for a registry in the resource group named *myResourceGroup*:

```
az acr list --resource-group myResourceGroup --query "[].{acrLoginServer:loginServer}" --output table
```

Use the docker tag command to tag the image with the ACR login server name and a version number of `v1`. Provide your own `<acrLoginServer>` name obtained in the previous step:

```
docker tag azure-vote-front <acrLoginServer>/azure-vote-front:v1
```

Finally, push the *azure-vote-front* image to your ACR registry. Again, replace `<acrLoginServer>` with the login server name of your own ACR registry, such as `myacrregistry.azurecr.io`:

```
docker push <acrLoginServer>/azure-vote-front:v1
```

Deploy the sample application to AKS

To deploy the sample application to your AKS cluster, you can use the Kubernetes manifest file in the root of the Azure vote repository repo. Open the *azure-vote-all-in-one-redis.yaml* manifest file with an editor such as `vi`. Replace `microsoft` with your ACR login server name. This value is found on line **47** of the manifest file:

```
containers:
- name: azure-vote-front
  image: microsoft/azure-vote-front:v1
```

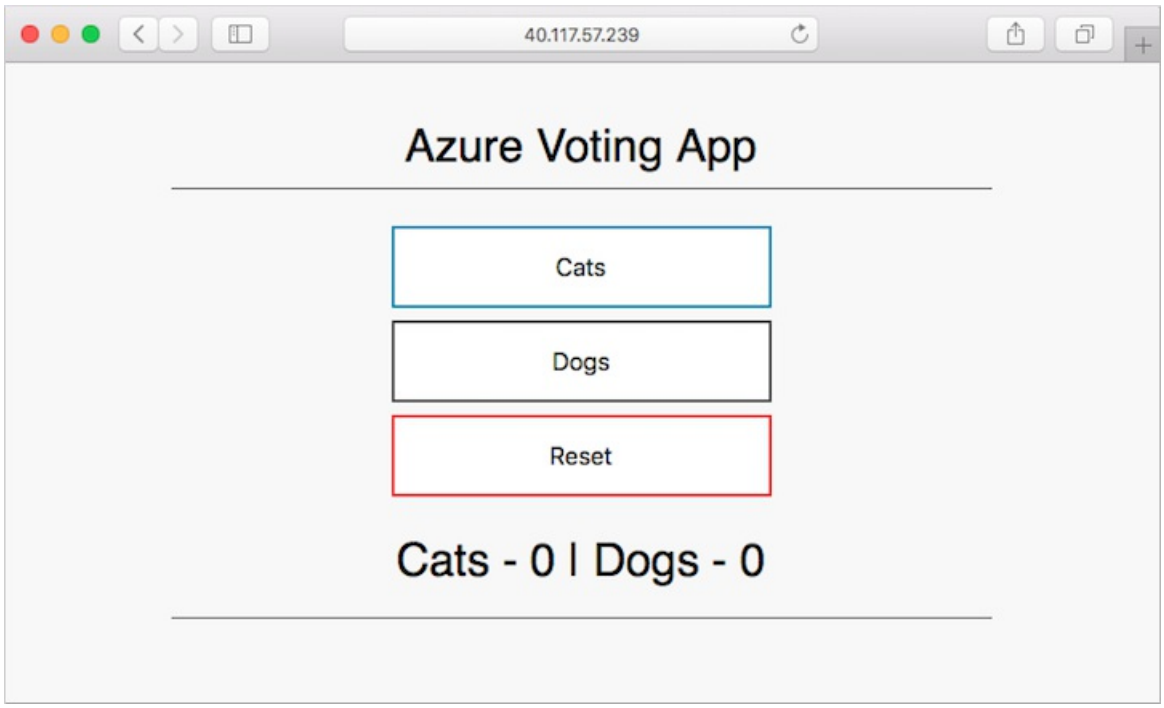
Next, use the kubectl apply command to deploy the application to your AKS cluster:

```
kubectl apply -f azure-vote-all-in-one-redis.yaml
```

A Kubernetes load balancer service is created to expose the application to the internet. This process can take a few minutes. To monitor the progress of the load balancer deployment, use the kubectl get service command with the `--watch` argument. Once the *EXTERNAL-IP* address has changed from *pending* to an *IP address*, use `Control + C` to stop the kubectl watch process.

```
$ kubectl get service azure-vote-front --watch

NAME               TYPE           CLUSTER-IP    EXTERNAL-IP   PORT(S)        AGE
azure-vote-front   LoadBalancer   10.0.215.27   <pending>     80:30747/TCP   22s
azure-vote-front   LoadBalancer   10.0.215.27   40.117.57.239   80:30747/TCP   2m
```

To see the application in action, open a web browser to the external IP address of your service. The Azure vote application is displayed, as shown in the following example:



Deploy Jenkins to an Azure VM

To quickly deploy Jenkins for use in this article, you can use the following script to deploy an Azure virtual machine, configure network access, and complete a basic installation of Jenkins. For authentication between Jenkins and the AKS cluster, the script copies your Kubernetes configuration file from your development system to the Jenkins system.

> **WARNING**
> This sample script is for demo purposes to quickly provision a Jenkins environment that runs on an Azure VM. It uses the Azure custom script extension to configure a VM and then display the required credentials. Your *~/.kube/config* is copied to the Jenkins VM.

Run the following commands to download and run the script. You should review the contents of any script prior to running it - https://raw.githubusercontent.com/Azure-Samples/azure-voting-app-redis/master/jenkins-tutorial/deploy-jenkins-vm.sh.

```
curl https://raw.githubusercontent.com/Azure-Samples/azure-voting-app-redis/master/jenkins-tutorial/deploy-jenkins-vm.sh > azure-jenkins.sh
sh azure-jenkins.sh
```

It takes a few minutes to create the VM and deploy the required components for Docker and Jenkins. When the script has completed, it outputs an address for the Jenkins server and a key to unlock the dashboard, as shown in the following example output:

```
Open a browser to http://40.115.43.83:8080
Enter the following to Unlock Jenkins:
667e24bba78f4de6b51d330ad89ec6c6
```

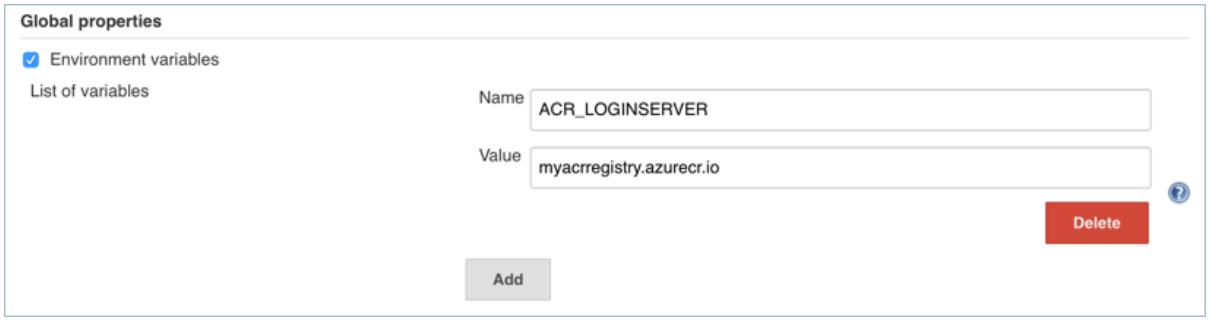
Open a web browser to the URL displayed and enter the unlock key. Follow the on-screen prompts to complete the Jenkins configuration:

- Choose **Install suggested plugins**
- Create the first admin user. Enter a username, such as *azureuser*, then provide your own secure password. Finally, type a full name and e-mail address.
- Select **Save and Finish**
- Once Jenkins is ready, select **Start using Jenkins**
  - If your web browser displays a blank page when you start using Jenkins, restart the Jenkins service. To restart the service, SSH to the public IP address of your Jenkins instance and type `sudo service jenkins restart`. Once the service has restarted, refresh you web browser.
- Sign in to Jenkins with the username and password you created in the install process.

Create a Jenkins environment variable

A Jenkins environment variable is used to hold the ACR login server name. This variable is referenced during the Jenkins build job. To create this environment variable, complete the following steps:

- On the left-hand side of the Jenkins portal, select **Manage Jenkins** > **Configure System**
- Under **Global Properties**, select **Environment variables**. Add a variable with the name `ACR_LOGINSERVER` and the value of your ACR login server.



- When complete, click **Save** at the bottom of the Jenkins configuration page.

Create a Jenkins credential for ACR

To allow Jenkins to build and then push updated container images to ACR, you need to specify credentials for ACR. This authentication can use Azure Active Directory service principals. In the pre-requisites, you configured the service principal for your AKS cluster with *Reader* permissions to your ACR registry. These permissions allow the AKS cluster to *pull* images from the ACR registry. During the CI/CD process, Jenkins builds new container images based on application updates, and needs to then *push* those images to the ACR registry. For separation of roles and permissions, now configure a service principal for Jenkins with *Contributor* permissions to your ACR registry.

Create a service principal for Jenkins to use ACR

First, create a service principal using the az ad sp create-for-rbac command:

```
$ az ad sp create-for-rbac --skip-assignment

{
  "appId": "626dd8ea-042d-4043-a8df-4ef56273670f",
  "displayName": "azure-cli-2018-09-28-22-19-34",
  "name": "http://azure-cli-2018-09-28-22-19-34",
  "password": "1ceb4df3-c567-4fb6-955e-f95ac9460297",
  "tenant": "72f988bf-86f1-41af-91ab-2d7cd011db48"
}
```

Make a note of the *appId* and *password* shown in your output. These values are used in following steps to configure the credential resource in Jenkins.

Get the resource ID of your ACR registry using the az acr show command, and store it as a variable. Provide your resource group name and ACR name:

```
ACR_ID=$(az acr show --resource-group myResourceGroup --name <acrLoginServer> --query "id" --output tsv)
```

Now create a role assignment to assign the service principal *Contributor* rights to the ACR registry. In the following example, provide your own *appId* shown in the output a previous command to create the service principal:

```
az role assignment create --assignee 626dd8ea-042d-4043-a8df-4ef56273670f --role Contributor --scope $ACR_ID
```

Create a credential resource in Jenkins for the ACR service principal

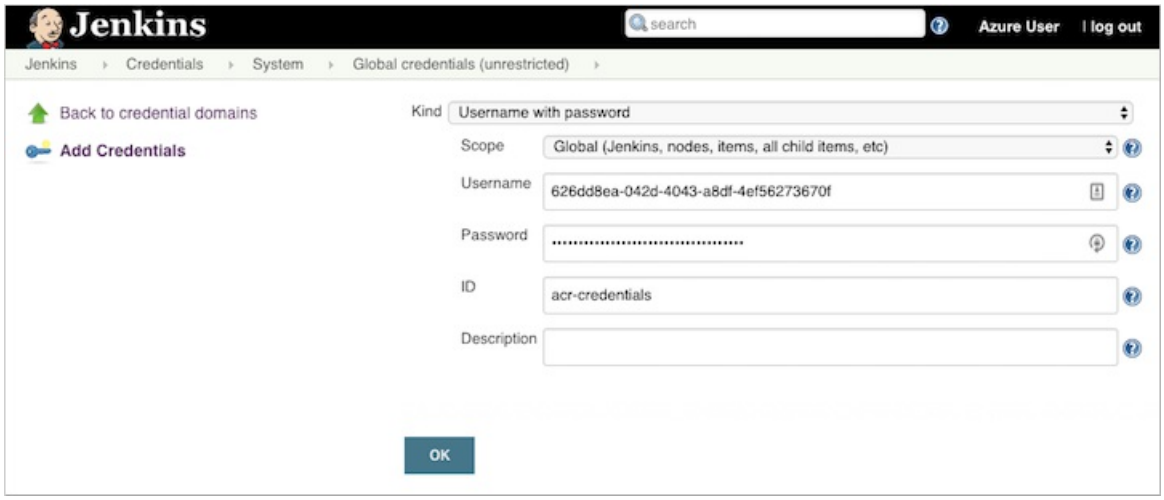
With the role assignment created in Azure, now store your ACR credentials in a Jenkins credential object. These credentials are referenced during the Jenkins build job.

Back on the left-hand side of the Jenkins portal, click **Credentials** > **Jenkins** > **Global credentials (unrestricted)** > **Add Credentials**

Ensure that the credential kind is **Username with password** and enter the following items:

- **Username** - The *appId* of the service principal created for authentication with your ACR registry.
- **Password** - The *password* of the service principal created for authentication with your ACR registry.
- **ID** - Credential identifier such as *acr-credentials*

When complete, the credentials form looks like the following example:

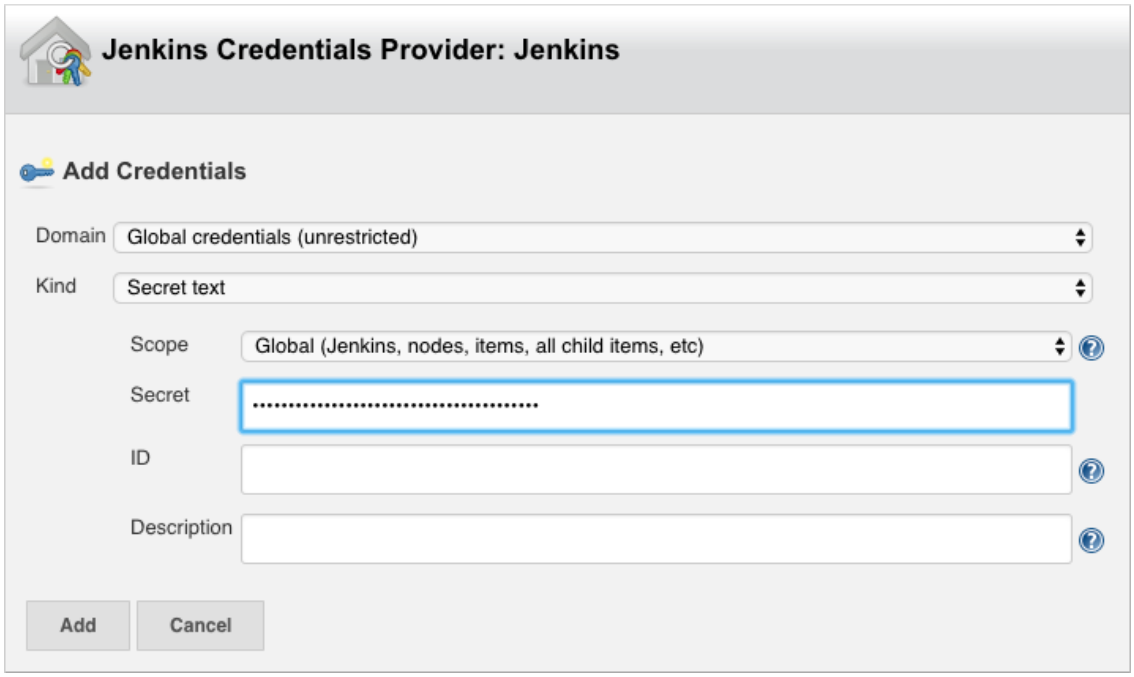


Click **OK** and return to the Jenkins portal.

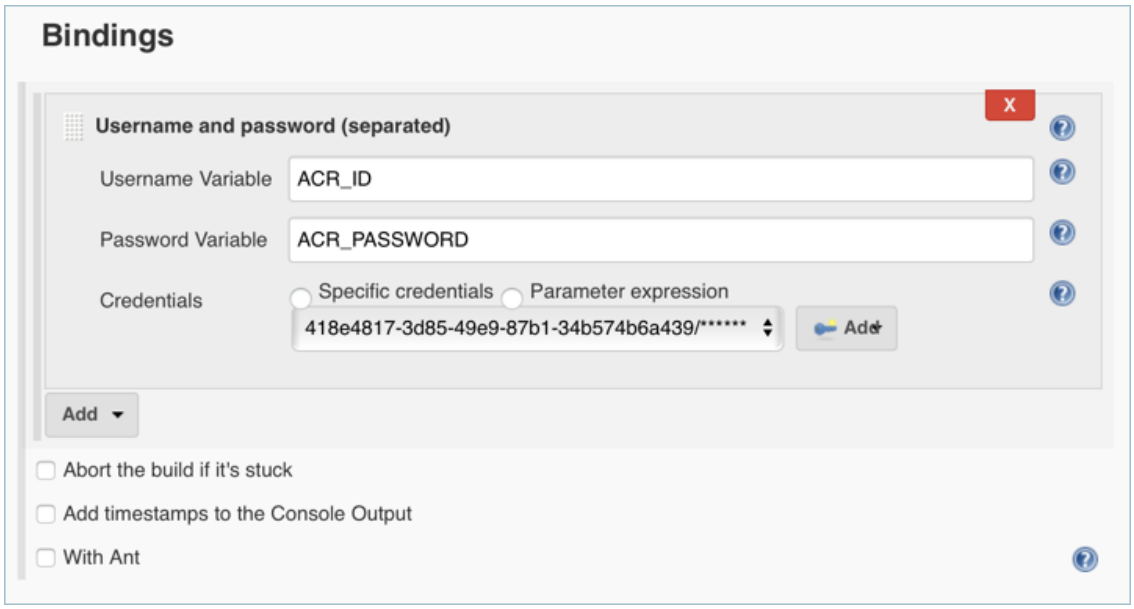
Create a Jenkins project

From the home page of your Jenkins portal, select **New item** on the left-hand side:

1. Enter *azure-vote* as job name. Choose **Freestyle project**, then select **OK**
2. Under the **General** section, select **GitHub project** and enter your forked repo URL, such as *https://github.com/<your-github-account>/azure-voting-app-redis*
3. Under the **Source code management** section, select **Git**, enter your forked repo *.git* URL, such as *https://github.com/<your-github-account>/azure-voting-app-redis.git*
    - For the credentials, click on and **Add** > **Jenkins**
    - Under **Kind**, select **Secret text** and enter your GitHub personal access token as the secret.
    - Select **Add** when done.



4. Under the **Build Triggers** section, select **GitHub hook trigger for GITscm polling**
5. Under **Build Environment**, select **Use secret texts or files**
6. Under **Bindings**, select **Add** > **Username and password (separated)**
    - Enter `ACR_ID` for the **Username Variable**, and `ACR_PASSWORD` for the **Password Variable**



7. Choose to add a **Build Step** of type **Execute shell** and use the following text. This script builds a new container image and pushes it to your ACR registry.

```
# Build new image and push to ACR.
WEB_IMAGE_NAME="${ACR_LOGINSERVER}/azure-vote-front:kube${BUILD_NUMBER}"
docker build -t $WEB_IMAGE_NAME ./azure-vote
docker login ${ACR_LOGINSERVER} -u ${ACR_ID} -p ${ACR_PASSWORD}
docker push $WEB_IMAGE_NAME
```

8. Add another **Build Step** of type **Execute shell** and use the following text. This script updates the application deployment in AKS with the new container image from ACR.

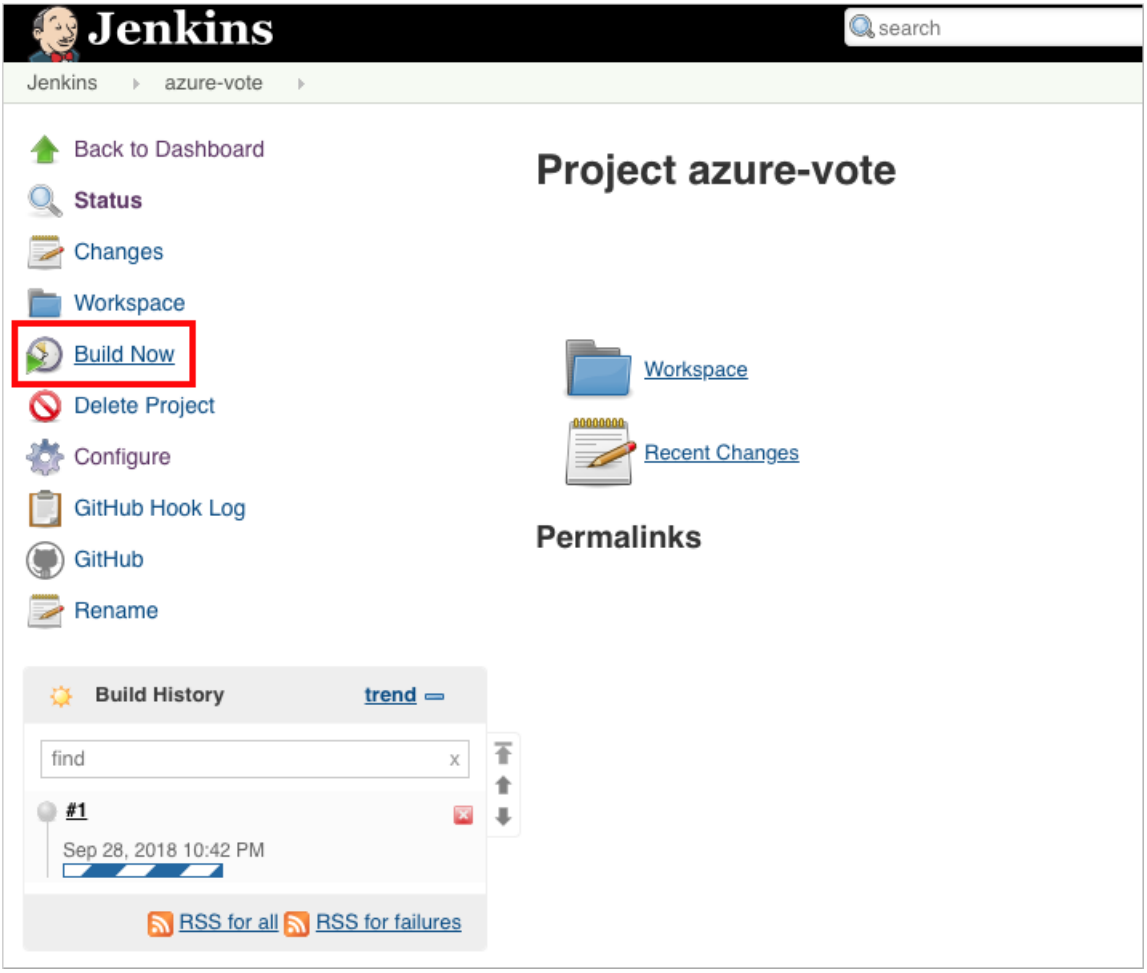
```
# Update kubernetes deployment with new image.
WEB_IMAGE_NAME="${ACR_LOGINSERVER}/azure-vote-front:kube${BUILD_NUMBER}"
kubectl set image deployment/azure-vote-front azure-vote-front=$WEB_IMAGE_NAME --kubeconfig
/var/lib/jenkins/config
```

9. Once completed, click **Save**.

Test the Jenkins build

Before you automate the job based on GitHub commits, first manually test the Jenkins build. This manual build validates that the job has been correctly configured, the proper Kubernetes authentication file is in place, and that the authentication with ACR works.

On the left-hand menu of the project, select **Build Now**.



The first build takes a minute or two as the Docker image layers are pulled down to the Jenkins server. Subsequent builds can use the cached image layers to improve the build times.

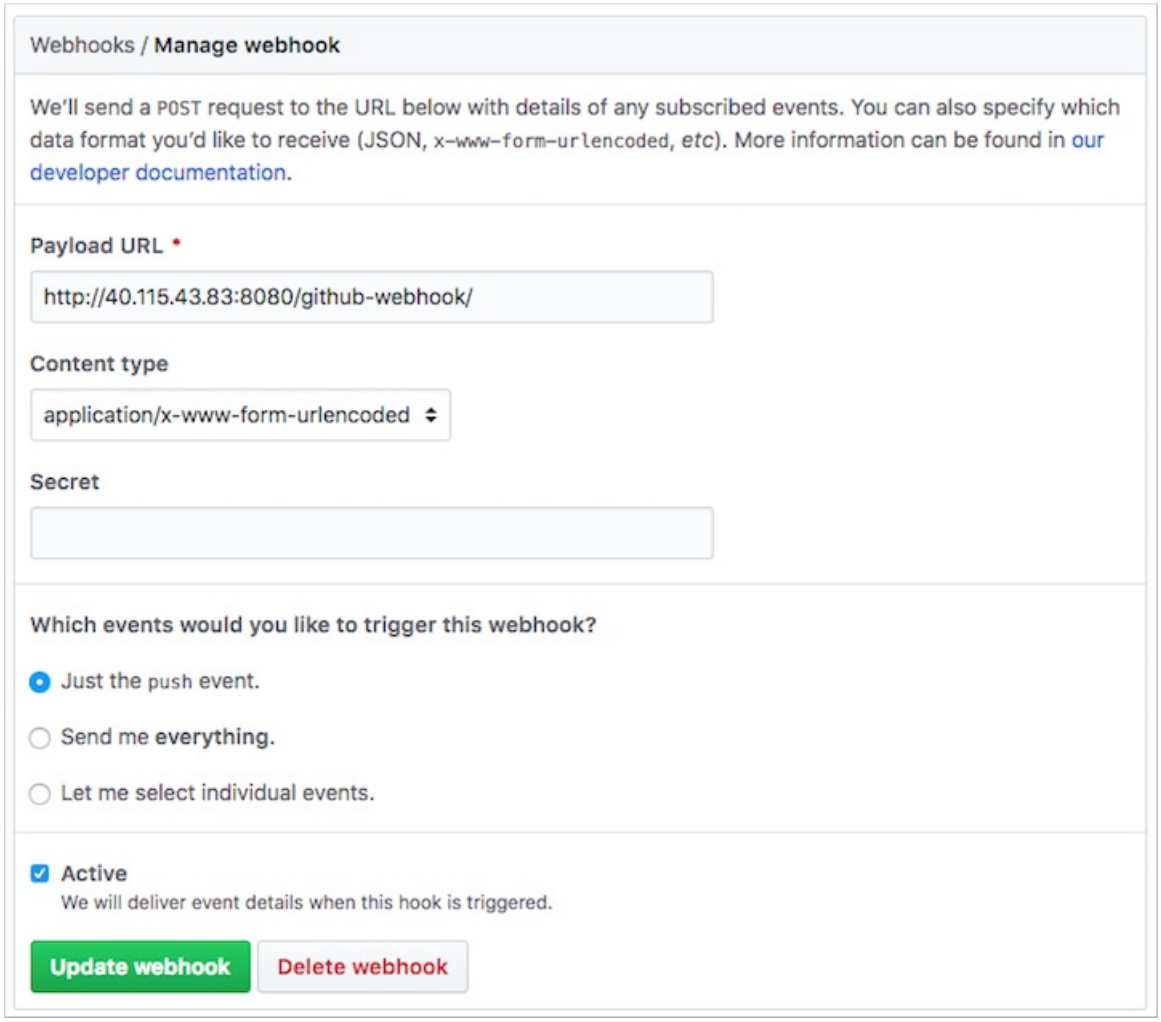
During the build process, the GitHub repository is cloned to the Jenkins build server. A new container image is built and pushed to the ACR registry. Finally, the Azure vote application running on the AKS cluster is updated to use the new image. Because no changes have been made to the application code, the application is not changed if you view the sample app in a web browser.

Once the build job is complete, click on **build #1** under build history. Select **Console Output** and view the output from the build process. The final line should indicate a successful build.

Create a GitHub webhook

With a successful manual build complete, now integrate GitHub into the Jenkins build. A webhook can be used to run the Jenkins build job each time a code commit is made in GitHub. To create the GitHub webhook, complete the following steps:

1. Browse to your forked GitHub repository in a web browser.
2. Select **Settings**, then select **Webhooks** on the left-hand side.
3. Choose to **Add webhook**. For the *Payload URL*, enter `http://<publicIp:8080>/github-webhook/`, where `<publicIp>` is the IP address of the Jenkins server. Make sure to include the trailing /. Leave the other defaults for content type and to trigger on *push* events.
4. Select **Add webhook**.



Test the complete CI/CD pipeline

Now you can test the whole CI/CD pipeline. When you push a code commit to GitHub, the following steps happen:

1. The GitHub webhook reaches out to Jenkins.
2. Jenkins starts the build job and pulls the latest code commit from GitHub.

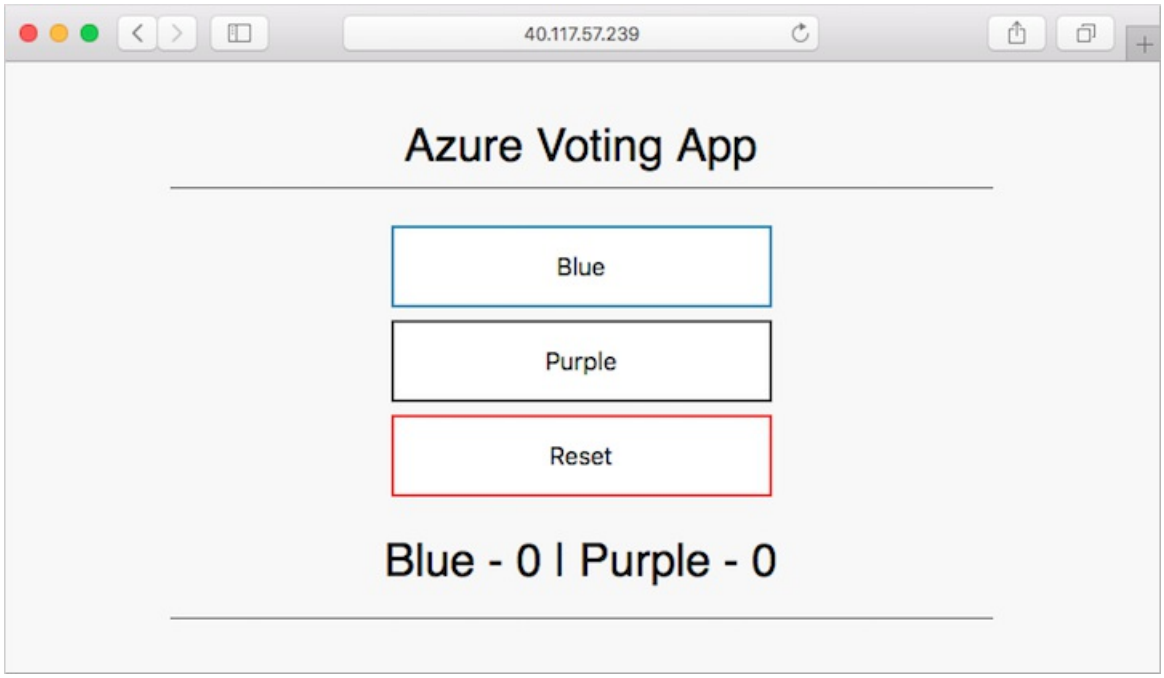
3. A Docker build is started using the updated code, and the new container image is tagged with the latest build number.
4. This new container image is pushed to Azure Container Registry.
5. Your application deployed to Azure Kubernetes Service updates with the latest container image from the Azure Container Registry registry.

On your development machine, open up the cloned application with a code editor. Under the */azure-vote/azure-vote* directory, open the file named **config_file.cfg**. Update the vote values in this file to something other than cats and dogs, as shown in the following example:

```
# UI Configurations
TITLE = 'Azure Voting App'
VOTE1VALUE = 'Blue'
VOTE2VALUE = 'Purple'
SHOWHOST = 'false'
```

When updated, save the file, commit the changes, and push these to your fork of the GitHub repository. The GitHub webhook triggers a new build job in Jenkins. In the Jenkins web dashboard, monitor the build process. It takes a few seconds to pull the latest code, create and push the updated image, and deploy the updated application in AKS.

Once the build is complete, refresh your web browser of the sample Azure vote application. Your changes are displayed, as shown in the following example:



Next steps

In this article, you learned how to use Jenkins as part of a CI/CD solution. AKS can integrate with other CI/CD solutions and automation tools, such as the Azure DevOps Project or creating an AKS cluster with Ansible.

Tutorial: Deploy your ASP.NET Core App to Azure Kubernetes Service (AKS) with the Azure DevOps Project

9/10/2018 • 7 minutes to read • Edit Online

The Azure DevOps Project presents a simplified experience where you bring your existing code and Git repository, or choose from one of the sample applications to create a continuous integration (CI) and continuous delivery (CD) pipeline to Azure. The DevOps Project automatically creates Azure resources such as AKS, creates and configures a release pipeline in Azure DevOps Services that includes a build and release pipeline for CI/CD, and then creates an Azure Application Insights resource for monitoring.

You will:

- Create an Azure DevOps Project for an ASP.NET Core App and AKS
- Configure Azure DevOps Services and an Azure subscription
- Examine the AKS cluster
- Examine the Azure DevOps Services CI pipeline
- Examine the Azure DevOps Services CD pipeline
- Commit changes to Git and automatically deploy to Azure
- Clean up resources

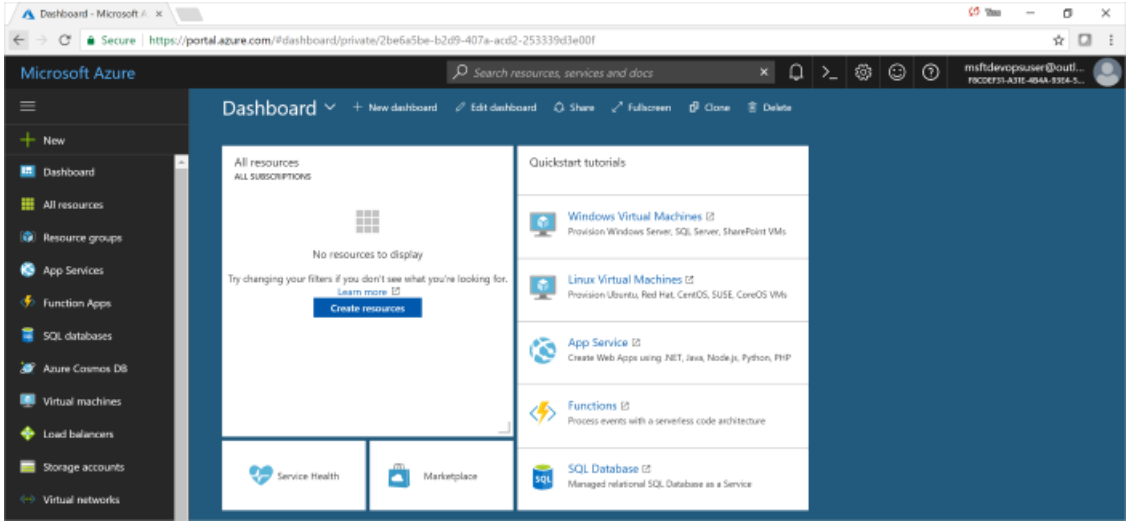
Prerequisites

- An Azure subscription. You can get one free through Visual Studio Dev Essentials.

Create an Azure DevOps Project for an ASP.NET Core App and AKS

The Azure DevOps Project creates a CI/CD pipeline in Azure. You can create a **new Azure DevOps Services** organization or use an **existing organization**. The Azure DevOps Project also creates **Azure resources** such as an AKS cluster in the **Azure subscription** of your choice.

1. Sign into the Microsoft Azure portal.
2. Choose the **Create a resource** icon in the left navigation bar, then search for **DevOps Project**. Choose **Create**.



3. Select **.NET**, and then choose **Next**.
4. For **Choose an application Framework**, select **ASP.NET Core**, and then select **Next**.
5. Select **Kubernetes Service**, then choose **Next**.

Configure Azure DevOps Services and an Azure subscription

1. Create a **new** Azure DevOps organization or choose an **existing** organization. Choose a **name** for your project.
2. Select your **Azure subscription**.
3. Select the **Change** link to see additional Azure configuration settings, and identify the **number of nodes** for the **Kubernetes cluster**. There are various options here for configuring the type and location of Azure services.
4. Exit the Azure configuration area, and choose **Done**.
5. It will take several minutes for the process to complete. A sample ASP.NET Core application is set up in an Azure Repos Git repository in your Azure DevOps Services organization, an AKS cluster is created, a CI/CD pipeline executes, and your application deploys to Azure.

   Once complete, the Azure DevOps **Project dashboard** loads in the Azure portal. You can also navigate to the **Azure DevOps Project Dashboard** directly from **All resources** in the **Azure portal**.

   This dashboard provides visibility into your Azure Repos **code repository**, **Azure DevOps Services CI/CD pipeline**, and **AKS cluster**. You can further configure additional CI/CD options in your Azure DevOps Services pipeline. On the right-side of the dashboard, select **Browse** to view your running application.

Examine the AKS cluster

The Azure DevOps Project automatically configures an AKS cluster. You can explore and customize the cluster. Follow the steps below to familiarize yourself with the AKS.

1. Navigate to the **Azure DevOps Project dashboard**.
2. On the right-side of the DevOps Project dashboard, select the **Kubernetes service**.
3. A blade opens for the AKS cluster. From this view you can perform various actions such as **monitor container health**, **search logs**, and open the **Kubernetes dashboard**.
4. On the right-side of the screen, select **View Kubernetes dashboard**. Optionally follow the steps to open

the Kubernetes dashboard.

Examine the Azure DevOps Services CI pipeline

The Azure DevOps Project automatically configures a Azure CI/CD pipeline in your Azure DevOps organization. You can explore and customize the pipeline. Follow the steps below to familiarize yourself with the Azure DevOps Services CI/CD pipeline.

1. Navigate to the **Azure DevOps Project dashboard**.
2. Select **Build Pipelines** from the **top** of the **Azure DevOps Project dashboard**. This link opens a browser tab and opens the Azure DevOps Services build pipeline for your new project.
3. Move the mouse cursor to the right of the build pipeline next to the **Status** field. Select the **ellipsis** that appears. This action opens a menu where you can perform several activities such as **queue a new build**, **pause a build**, and **edit the build pipeline**.
4. Select **Edit**.
5. From this view, **examine the various tasks** for your. The build performs various tasks such as fetching sources from the Azure DevOps Services Repos Git repository, restoring dependencies, and publishing outputs used for deployments.
6. At the top of the build pipeline, select the **build pipeline name**.
7. Change the **name** of your build pipeline to something more descriptive. Select **Save & queue**, then select **Save**.
8. Under your build pipeline name, select **History**. You see an audit trail of your recent changes for the build. Azure DevOps Services keeps track of any changes made to the build pipeline and allows you to compare versions.
9. Select **Triggers**. The Azure DevOps Project automatically created a CI trigger, and every commit to the repository starts a new build. You can optionally choose to include or exclude branches from the CI process.
10. Select **Retention**. Based on your scenario, you can specify policies to keep or remove a certain number of builds.

Examine the Azure DevOps Services CD Release pipeline

The Azure DevOps Project automatically creates and configures the necessary steps to deploy from your Azure DevOps Services organization to your Azure subscription. These steps include configuring an Azure service connection to authenticate Azure DevOps Services with your Azure subscription. The automation also creates an Azure DevOps Services Release pipeline, and the Release pipeline provides the CD to the Azure. Follow the steps below to examine more about the Azure DevOps Services Release pipeline.

1. Select **Build and Release**, then choose **Releases**. The Azure DevOps Project created an Azure DevOps Services release pipeline to manage deployments to Azure.
2. On the left-hand side of the browser, select the **ellipsis** next to your release pipeline, then choose **Edit**.
3. The release pipeline contains a **pipeline**, which defines the release process. Under **Artifacts**, select **Drop**. The build pipeline you examined in the previous steps produces the output used for the artifact.
4. To the right-hand side of the **Drop** icon, select the **Continuous deployment trigger icon** (which appears as a lightning bolt.) This release pipeline has an enabled CD trigger. The trigger creates a deployment every time there is a new build artifact available. Optionally, you can disable the trigger, so your deployments will then require manual execution.

5. On the right-hand side of the browser, select **View releases**. This view shows a history of releases.
6. Select the **ellipsis** next to one of your releases, and choose **Open**. There are several menus to explore from this view such as a **release summary**, **associated work items**, and **Tests**.
7. Select **Commits**. This view shows code commits associated with the specific deployment. You can compare releases to view the commit differences between deployments.
8. Select **Logs**. The logs contain useful information about the deployment process. They can be viewed both during and after deployments.

Commit changes to Azure DevOps Services and automatically deploy to Azure

> **NOTE**
> The steps below test the CI/CD pipeline with a simple text change to your web app.

You're now ready to collaborate with a team on your app with a CI/CD process that automatically deploys your latest work to your web site. Each change to the Azure DevOps Services Git repo starts a build in Azure DevOps Services, and a CD pipeline deploys your changes to Azure. Follow the steps below, or use other techniques to commit changes to your repository. For example, you can **clone** the Git repository in your favorite tool or IDE, and then push changes to this repo.

1. Select **Code** and then **Files** from the Azure DevOps Services menu, and navigate to your repository.
2. Navigate to the **Views\Home** directory, then select the **ellipsis** next to the **Index.cshtml** file, and then choose **Edit**.
3. Make a change to the file such as some text inside one of the **div tags**. At the top right, select **Commit**. Select **Commit** again to push your change.
4. In a few moments, a **build starts in Azure DevOps Services**, and then a release executes to deploy the changes. You can monitor the **build status** with the DevOps Project dashboard or in the browser with your Azure DevOps Services organization.
5. Once the release completes, **refresh your application** in the browser to verify you see your changes.

Clean up resources

> **NOTE**
> The steps below will permanently delete resources. Only use this functionality after carefully reading the prompts.

If you are testing, you can clean up resources to avoid creating billing charges. When no longer needed, you can delete the Azure Kubernetes cluster and related resources created in this tutorial by using the **Delete** functionality on the Azure DevOps Project dashboard. **Be careful**, as the delete functionality destroys the data created by the Azure DevOps Project in both Azure and Azure DevOps Services, and you will not be able to retrieve it once it's gone.

1. From the **Azure portal**, navigate to the **Azure DevOps Project**.
2. On the **top right** side of the dashboard, select **Delete**. After reading the prompt, select **Yes** to **permanently delete** the resources.

Next steps

You can optionally modify these build and release pipelines to meet the needs of your team. You can also use this CI/CD pattern as a template for your other pipelines. You learned how to:

- Create an Azure DevOps Project for an ASP.NET Core App and AKS
- Configure Azure DevOps Services and an Azure subscription
- Examine the AKS cluster
- Examine the Azure DevOps Services CI pipeline
- Examine the Azure DevOps Services CD pipeline
- Commit changes to Git and automatically deploy to Azure
- Clean up resources

To learn more about using the Kubernetes dashboard see below:

Use the Kubernetes dashboard

Access the Kubernetes dashboard with Azure Kubernetes Service (AKS)

7/27/2018 • 3 minutes to read • Edit Online

Kubernetes includes a web dashboard that can be used for basic management operations. This article shows you how to access the Kubernetes dashboard using the Azure CLI, then guides you through some basic dashboard operations. For more information on the Kubernetes dashboard, see Kubernetes Web UI Dashboard.

Before you begin

The steps detailed in this document assume that you have created an AKS cluster and have established a `kubectl` connection with the cluster. If you need to create an AKS cluster, see the AKS quickstart.

You also need the Azure CLI version 2.0.27 or later installed and configured. Run `az --version` to find the version. If you need to install or upgrade, see Install Azure CLI.

Start Kubernetes dashboard

To start the Kubernetes dashboard, use the az aks browse command. The following example opens the dashboard for the cluster named *myAKSCluster* in the resource group named *myResourceGroup*:

```
az aks browse --resource-group myResourceGroup --name myAKSCluster
```

This command creates a proxy between your development system and the Kubernetes API, and opens a web browser to the Kubernetes dashboard.

For RBAC-enabled clusters

If your AKS cluster uses RBAC, a *ClusterRoleBinding* must be created before you can correctly access the dashboard. By default, the Kubernetes dashboard is deployed with minimal read access and displays RBAC access errors. The Kubernetes dashboard does not currently support user-provided credentials to determine the level of access, rather it uses the roles granted to the service account. A cluster administrator can choose to grant additional access to the *kubernetes-dashboard* service account, however this can be a vector for privilege escalation. You can also integrate Azure Active Directory authentication to provide a more granular level of access.

To create a binding, use the kubectl create clusterrolebinding command as shown in the following example.

WARNING

This sample binding does not apply any additional authentication components and may lead to insecure use. The Kubernetes dashboard is open to anyone with access to the URL. Do not expose the Kubernetes dashboard publicly.

For more information on using the different authentication methods, see the Kubernetes dashboard wiki on access controls.

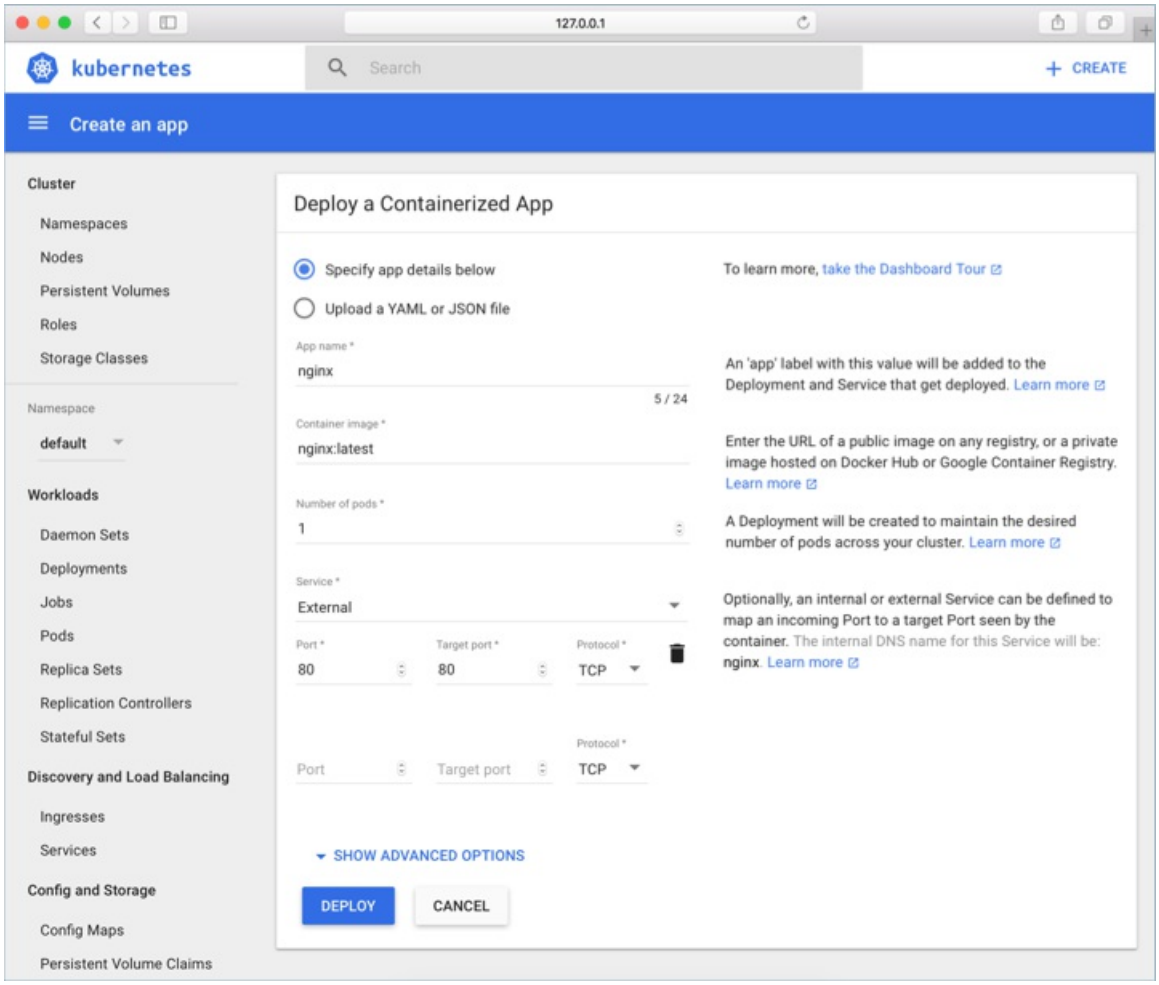
```
kubectl create clusterrolebinding kubernetes-dashboard --clusterrole=cluster-admin --serviceaccount=kube-system:kubernetes-dashboard
```

You can now access the Kubernetes dashboard in your RBAC-enabled cluster. To start the Kubernetes dashboard, use the az aks browse command as detailed in the previous step.

Run an application

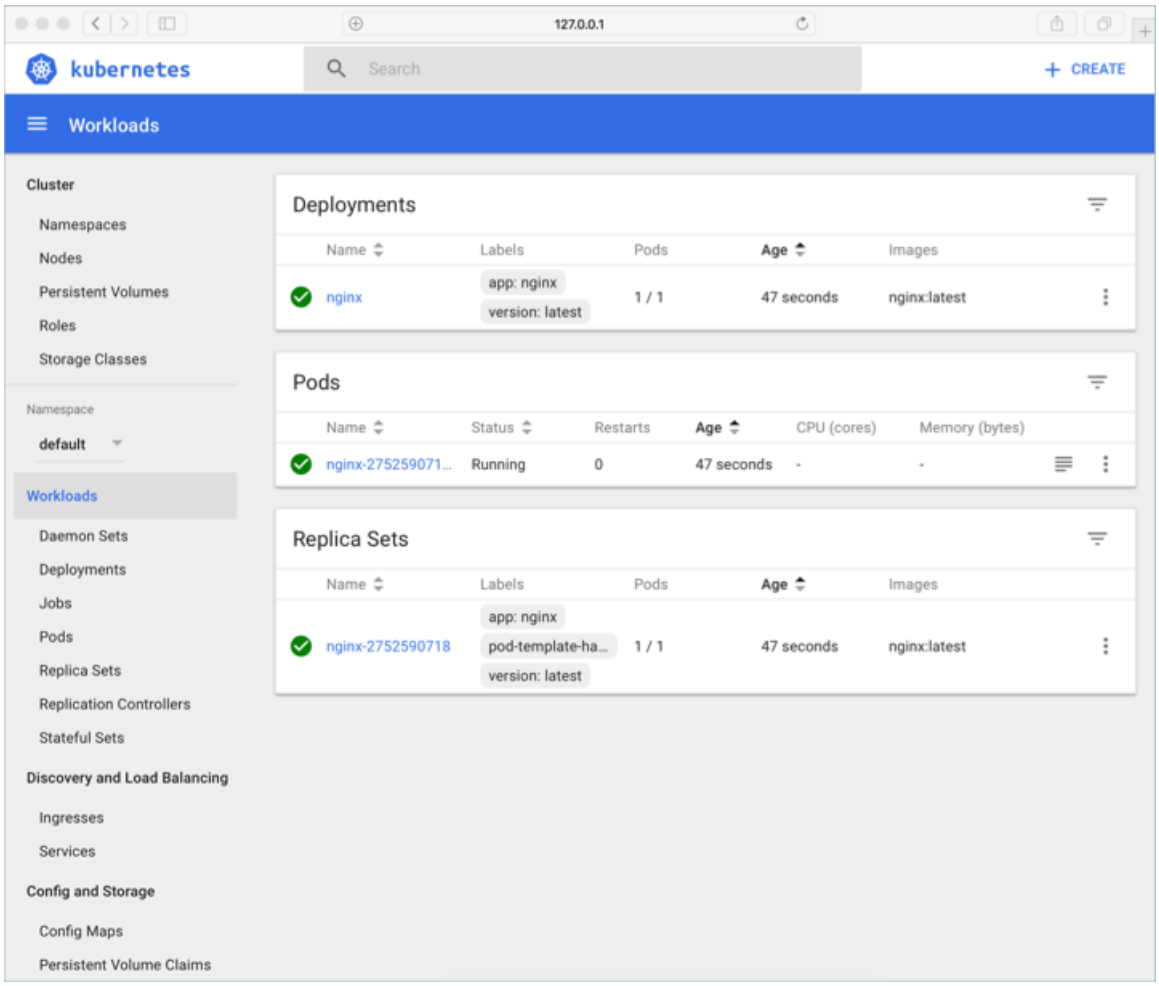
In the Kubernetes dashboard, click the **Create** button in the upper right window. Give the deployment the name `nginx` and enter `nginx:latest` for the container image name. Under **Service**, select **External** and enter `80` for both the port and target port.

When ready, click **Deploy** to create the deployment.

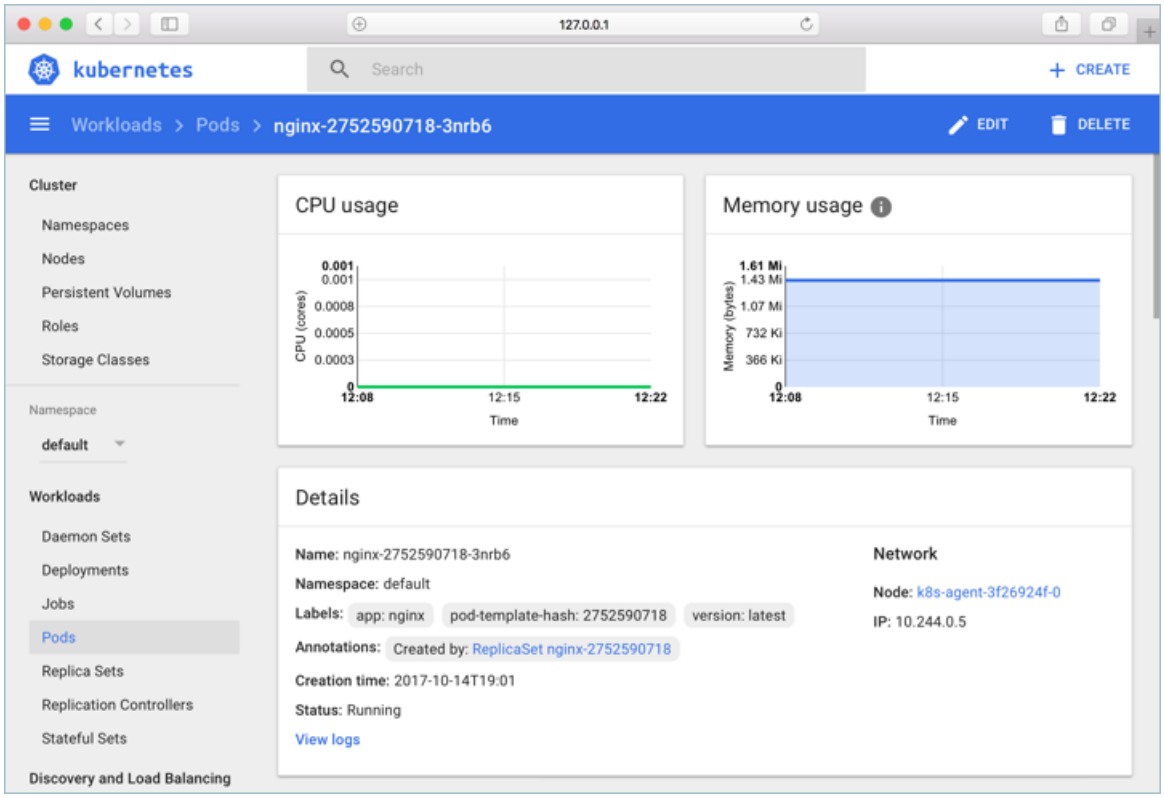


View the application

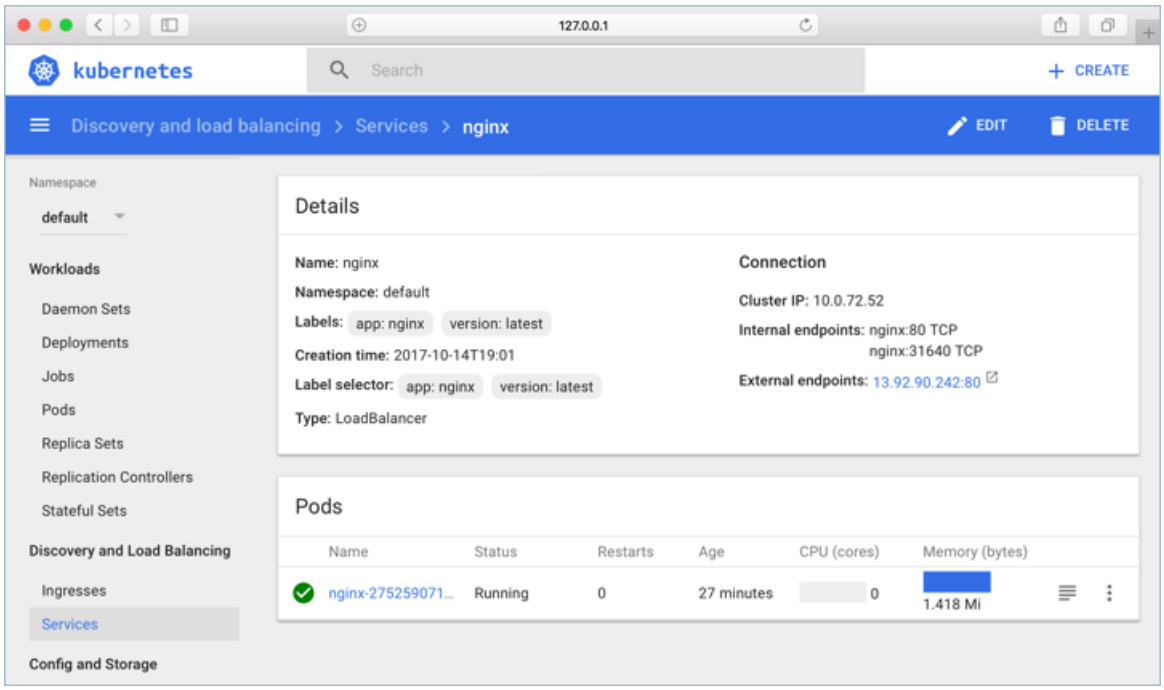
Status about the application can be seen on the Kubernetes dashboard. Once the application is running, each component has a green checkbox next to it.



To see more information about the application pods, click on **Pods** in the left-hand menu, and then select the **NGINX** pod. Here you can see pod-specific information such as resource consumption.



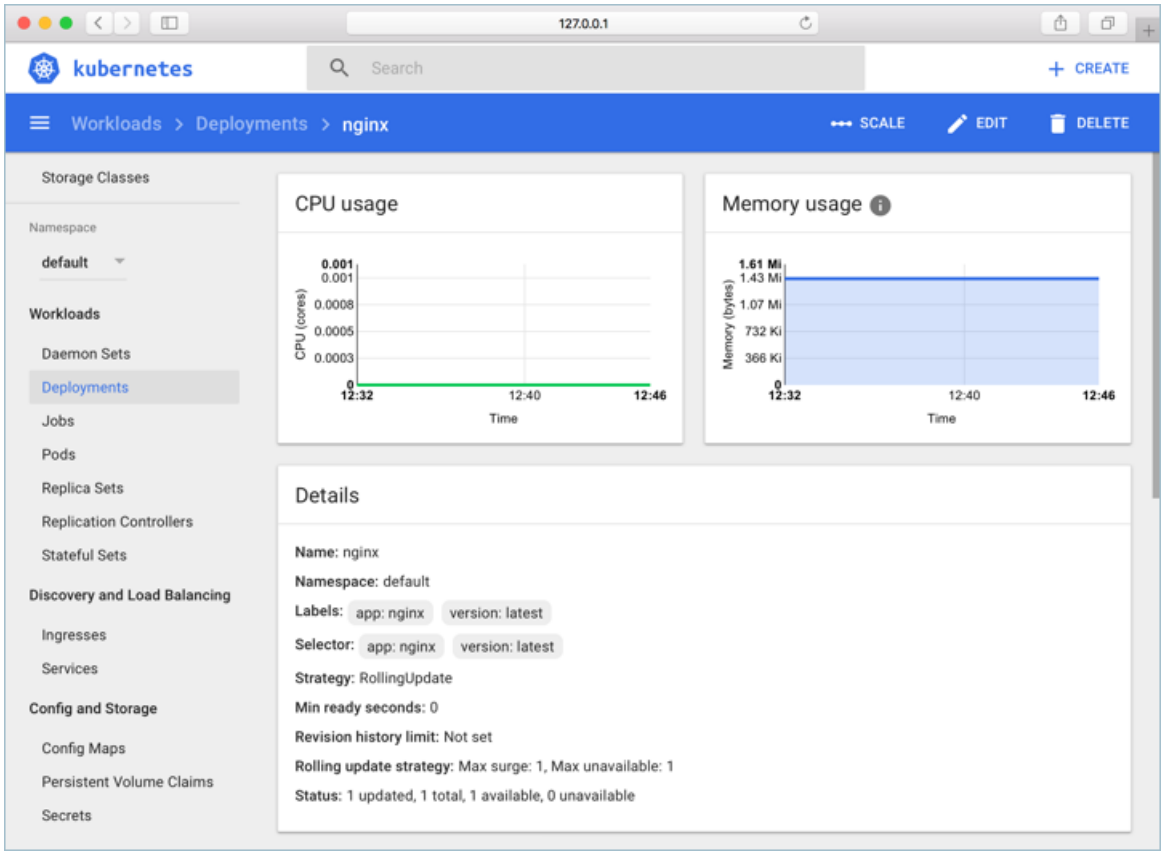
To find the IP address of the application, click on **Services** in the left-hand menu, and then select the **NGINX** service.



Edit the application

In addition to creating and viewing applications, the Kubernetes dashboard can be used to edit and update application deployments.

To edit a deployment, click **Deployments** in the left-hand menu, and then select the **NGINX** deployment. Finally, select **Edit** in the upper right-hand navigation bar.



Locate the `spec.replica` value, which should be 1, change this value to 3. In doing so, the replica count of the NGINX deployment is increased from 1 to 3.

Select **Update** when ready.

Next steps

For more information about the Kubernetes dashboard, see the Kubernetes documentation.

Kubernetes Web UI Dashboard

AKS troubleshooting

8/30/2018 • 3 minutes to read • Edit Online

When you create or manager AKS clusters, you may occasionally encounter issues. This article details some common issues and troubleshooting steps.

In general, where do I find information about debugging Kubernetes issues?

Here is an official link to troubleshooting kubernetes clusters. Here is a link to a troubleshooting guide published by a Microsoft engineer around troubleshooting pods, nodes, clusters, etc.

I am getting a quota exceeded error during create or upgrade. What should I do?

You will need to request cores here.

What is the max pods per node setting for AKS?

The max pods per node are set to 30 by default if you deploy an AKS cluster in the Azure portal. The max pods per node are set to 110 by default if you deploy an AKS cluster in the Azure CLI. (Ensure you are using the latest version of the Azure CLI). This default setting can be changed using the –max-nodes-per-pod flag in the az aks create command.

I am getting "insufficientSubnetSize" error while deploying an AKS cluster with Advanced networking. What should I do?

In Custom VNET option selected for networking during AKS creates, the Azure CNI is used for IPAM. The number of nodes in an AKS cluster can be anywhere between 1 and 100. Based upon 2) above the subnet size should be greater than product of the number of nodes and the max pod per node Subnet size > no of nodes in the cluster * max pods per node.

My pod is stuck in 'CrashLoopBackOff' mode. What should I do?

There might be various reasons for the pod being stuck in that mode. You might want to look into the

- The pod itself using `kubectl describe pod <pod-name>`
- The logs using `kubectl log <pod-name>`

I am trying to enable RBAC on an existing cluster. Can you tell me how I can do that?

Unfortunately enabling RBAC on existing clusters is not supported at this time. You will need to explicitly create new clusters. If you use the CLI, RBAC is enabled by default whereas a toggle button to enable it is available in the AKS portal create workflow.

I created a cluster using the Azure CLI with defaults or the Azure portal with RBAC enabled and numerous warnings in the kubernetes dashboard. The dashboard used to work before without any warnings. What should I do?

The reason for getting warnings on the dashboard is that now it is enabled with RBAC'ed and access to it has been disabled by default. In general, this approach is considered good practice since the default exposure of the dashboard to all users of the cluster can lead to security threats. If you still want to enable the dashboard, follow this blog to enable it.

I can't seem to connect to the dashboard. What should I do?

The easiest way to access your service outside the cluster is to run kubectl proxy, which will proxy requests to your localhost port 8001 to the Kubernetes API server. From there, the apiserver can proxy to your service: http://localhost:8001/api/v1/namespaces/kube-system/services/kubernetes-dashboard/proxy/#!/node?namespace=default

If you don't see the kubernetes dashboard, then check if the kube-proxy pod is running in the kube-system

namespace. If it is not in running state, delete the pod and it will restart.

I could not get logs using Kubectl logs or cannot connect to the api server getting the "Error from server: error dialing backend: dial tcp…". What should I do?

Make sure that the default NSG is not modified and port 22 is open for connection to the API server. Check if the tunnelfront pod is running in the kube-system namespace. If it is not, force delete it and it will restart.

I am trying to upgrade or scale and am getting "message": "Changing property 'imageReference' is not allowed." Error. How do I fix this issue?

It is possible that you are getting this error because you have modified the tags in the agent nodes inside the AKS cluster. Modifying and deleting tags and other properties of resources in the MC_* resource group can lead to unexpected results. Modifying the resources under the MC_* in the AKS cluster breaks the SLO.

Checking for Kubernetes best practices in your cluster

9/18/2018 • 2 minutes to read • Edit Online

There are several best practices that you should follow on your Kubernetes deployments to ensure the best performance and resilience for your applications. You can use the kube-advisor tool to look for deployments that aren't following those suggestions.

About kube-advisor

The kube-advisor tool is a single container designed to be run on your cluster. It queries the Kubernetes API server for information about your deployments and returns a set of suggested improvements.

> **NOTE**
> The kube-advisor tool is supported by Microsoft on a best-effort basis. Issues and suggestions should be filed on GitHub.

Running kube-advisor

To run the tool on a cluster that is configured for role-based access control (RBAC), using the following commands. The first command creates a Kubernetes service account. The second command runs the tool in a pod using that service account and configures the pod for deletion after it exits.

```
kubectl apply -f https://raw.githubusercontent.com/Azure/kube-advisor/master/sa.yaml?
token=ABLLDrNcuHMro9jQ0xduCaEbpzLupzQUks5bh3RhwA%3D%3D

kubectl run --rm -i -t kubeadvisor --image=mcr.microsoft.com/aks/kubeadvisor --restart=Never --overrides="{
\"apiVersion\": \"v1\", \"spec\": { \"serviceAccountName\": \"kube-advisor\" } }"
```

If you aren't using RBAC, you can run the command as follows:

```
kubectl run --rm -i -t kubeadvisor --image=mcr.microsoft.com/aks/kubeadvisor --restart=Never
```

Within a few seconds, you should see a table describing potential improvements to your deployments.

```
sean@Azure:~$ kubectl run -i -t --rm kubeadvisor --image mcr.microsoft.com/aks/kubeadvisor --restart=Never
If you don't see a command prompt, try pressing enter.

+-------------+-------------------------------------------------+---------------+-----------------------+-------------------------------+
| NAMESPACE   | POD NAME                                        | POD CPU/MEMORY | CONTAINER             | ISSUE                         |
+-------------+-------------------------------------------------+---------------+-----------------------+-------------------------------+
| postgresdemo | postgresdemo-worker-0                          | 1m / 9440Ki   | postgresdemo-worker   | CPU Resource Limits Missing   |
+             +                                                 +               +                       +-------------------------------+
|             |                                                 |               |                       | Memory Resource Limits Missing |
+             +-------------------------------------------------+               +                       +                               +
|             | postgresdemo-worker-1                           | 1m / 9152Ki   |                       |                               |
+             +                                                 +               +                       +-------------------------------+
|             |                                                 |               |                       | CPU Resource Limits Missing   |
+-------------+-------------------------------------------------+---------------+-----------------------+-------------------------------+
```

Checks performed

The tool validates several Kubernetes best practices, each with their own suggested remediation.

Resource requests and limits

Kubernetes supports defining resource requests and limits on pod specifications. The request defines the minimum CPU and memory required to run the container. The limit defines the maximum CPU and memory that should be allowed.

By default, no requests or limits are set on pod specifications. This can lead to nodes being overscheduled and containers being starved. The kube-advisor tool highlights pods without requests and limits set.

Cleaning up

If your cluster has RBAC enabled, you can clean up the `ClusterRoleBinding` after you've run the tool using the following command:

```
kubectl delete -f https://raw.githubusercontent.com/Azure/kube-advisor/master/sa.yaml?
token=ABLLDrNcuHMro9jQ0xduCaEbpzLupzQUks5bh3RhwA%3D%3D
```

If you are running the tool against a cluster that is not RBAC-enabled, no cleanup is required.

Next steps

- Troubleshoot issues with Azure Kubernetes Service

Connect with SSH to Azure Kubernetes Service (AKS) cluster nodes for maintenance or troubleshooting

8/22/2018 • 3 minutes to read • Edit Online

Throughout the lifecycle of your Azure Kubernetes Service (AKS) cluster, you may need to access an AKS node. This access could be for maintenance, log collection, or other troubleshooting operations. The AKS nodes are Linux VMs, so you can access them using SSH. For security purposes, the AKS nodes are not exposed to the internet.

This article shows you how to create an SSH connection with an AKS node using their private IP addresses.

Add your public SSH key

By default, SSH keys are generated when you create an AKS cluster. If you did not specify your own SSH keys when you created your AKS cluster, add your public SSH keys to the AKS nodes.

To add your SSH key to an AKS node, complete the following steps:

1. Get the resource group name for your AKS cluster resources using az aks show. Provide your own core resource group and AKS cluster name:

   ```
   az aks show --resource-group myResourceGroup --name myAKSCluster --query nodeResourceGroup -o tsv
   ```

2. List the VMs in the AKS cluster resource group using the az vm list command. These VMs are your AKS nodes:

   ```
   az vm list --resource-group MC_myResourceGroup_myAKSCluster_eastus -o table
   ```

   The following example output shows the AKS nodes:

   ```
   Name                       ResourceGroup                                Location
   -------------------------  -------------------------------------------  ----------
   aks-nodepool1-79590246-0  MC_myResourceGroupAKS_myAKSClusterRBAC_eastus  eastus
   ```

3. To add your SSH keys to the node, use the az vm user update command. Provide the resource group name and then one of the AKS nodes obtained in the previous step. By default, the username for the AKS nodes is *azureuser*. Provide the location of your own SSH public key location, such as *~/.ssh/id_rsa.pub*, or paste the contents of your SSH public key:

   ```
   az vm user update \
     --resource-group MC_myResourceGroup_myAKSCluster_eastus \
     --name aks-nodepool1-79590246-0 \
     --username azureuser \
     --ssh-key-value ~/.ssh/id_rsa.pub
   ```

Get the AKS node address

The AKS nodes are not publicly exposed to the internet. To SSH to the AKS nodes, you use the private IP address.

View the private IP address of an AKS cluster node using the az vm list-ip-addresses command. Provide your own AKS cluster resource group name obtained in a previous az-aks-show step:

```
az vm list-ip-addresses --resource-group MC_myAKSCluster_myAKSCluster_eastus -o table
```

The following example output shows the private IP addresses of the AKS nodes:

```
VirtualMachine            PrivateIPAddresses
------------------------  --------------------
aks-nodepool1-79590246-0  10.240.0.4
```

Create the SSH connection

To create an SSH connection to an AKS node, you run a helper pod in your AKS cluster. This helper pod provides you with SSH access into the cluster and then additional SSH node access. To create and use this helper pod, complete the following steps:

1. Run a `debian` container image and attach a terminal session to it. This container can be used to create an SSH session with any node in the AKS cluster:

   ```
   kubectl run -it --rm aks-ssh --image=debian
   ```

2. The base Debian image doesn't include SSH components. Once the terminal session is connected to the container, install an SSH client using `apt-get` as follows:

   ```
   apt-get update && apt-get install openssh-client -y
   ```

3. In a new terminal window, not connected to your container, list the pods on your AKS cluster using the kubectl get pods command. The pod created in the previous step starts with the name *aks-ssh*, as shown in the following example:

   ```
   $ kubectl get pods

   NAME                       READY     STATUS    RESTARTS   AGE
   aks-ssh-554b746bcf-kbwvf   1/1       Running   0          1m
   ```

4. In the first step of this article, you added your public SSH key the AKS node. Now, copy your private SSH key into the pod. This private key is used to create the SSH into the AKS nodes.

   Provide your own *aks-ssh* pod name obtained in the previous step. If needed, change *~/.ssh/id_rsa* to location of your private SSH key:

   ```
   kubectl cp ~/.ssh/id_rsa aks-ssh-554b746bcf-kbwvf:/id_rsa
   ```

5. Back in the terminal session to your container, update the permissions on the copied `id_rsa` private SSH key so that it is user read-only:

   ```
   chmod 0600 id_rsa
   ```

6. Now create an SSH connection to your AKS node. Again, the default username for AKS nodes is *azureuser*. Accept the prompt to continue with the connection as the SSH key is first trusted. You are then provided

with the bash prompt of your AKS node:

```
$ ssh -i id_rsa azureuser@10.240.0.4

ECDSA key fingerprint is SHA256:A6rnRkfpG21TaZ8XmQCCgdi9G/MYIMc+gFAuY9RUY70.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '10.240.0.4' (ECDSA) to the list of known hosts.

Welcome to Ubuntu 16.04.5 LTS (GNU/Linux 4.15.0-1018-azure x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  Get cloud support with Ubuntu Advantage Cloud Guest:
    http://www.ubuntu.com/business/services/cloud

[...]

azureuser@aks-nodepool1-79590246-0:~$
```

Remove SSH access

When done, `exit` the SSH session and then `exit` the interactive container session. When this container session closes, the pod used for SSH access from the AKS cluster is deleted.

Next steps

If you need additional troubleshooting data, you can view the kubelet logs or view the Kubernetes master node logs.

Frequently asked questions about Azure Kubernetes Service (AKS)

9/4/2018 • 3 minutes to read • Edit Online

This article addresses frequent questions about Azure Kubernetes Service (AKS).

Which Azure regions provide the Azure Kubernetes Service (AKS) today?

For a complete list of available regions, see AKS Regions and availability.

Does AKS support node autoscaling?

Yes, autoscaling is available via the Kubernetes autoscaler as of Kubernetes 1.10. For more information on how to configure and use the cluster autoscaler, see Cluster autoscale on AKS.

Does AKS support Kubernetes role-based access control (RBAC)?

Yes, Kubernetes RBAC is enabled by default when clusters are created with the Azure CLI. RBAC can be enabled for clusters created using the Azure portal or templates.

Can I deploy AKS into my existing virtual network?

Yes, you can deploy an AKS cluster into an existing virtual network using the advanced networking feature.

Can I restrict the Kubernetes API server to only be accessible within my virtual network?

Not at this time. The Kubernetes API server is exposed as a public fully qualified domain name (FQDN). You can control access to your cluster using Kubernetes role-based access control (RBAC) and Azure Active Directory (AAD)

Are security updates applied to AKS agent nodes?

Yes, Azure automatically applies security patches to the nodes in your cluster on a nightly schedule. However, you are responsible for ensuring that nodes are rebooted as required. You have several options for performing node reboots:

- Manually, through the Azure portal or the Azure CLI.
- By upgrading your AKS cluster. Cluster upgrades automatically cordon and drain nodes, then bring each node back up with the latest Ubuntu image and a new patch version or a minor Kubernetes version. For more information, see Upgrade an AKS cluster.
- Using Kured, an open-source reboot daemon for Kubernetes. Kured runs as a DaemonSet and monitors each node for the presence of a file indicating that a reboot is required. OS reboots are managed across the cluster using the same cordon and drain process as a cluster upgrade.

Why are two resource groups created with AKS?

Each AKS deployment spans two resource groups:

- The first resource group is created by you and contains only the Kubernetes service resource. The AKS resource provider automatically creates the second one during deployment, such as *MC_myResourceGroup_myAKSCluster_eastus*.
- This second resource group, such as *MC_myResourceGroup_myAKSCluster_eastus*, contains all of the infrastructure resources associated with the cluster. These resources include the Kubernetes node VMs, virtual networking, and storage. This separate resource group is created to simplify resource cleanup.

If you create resources for use with your AKS cluster, such as storage accounts or reserved public IP addresses, place them in the automatically generated resource group.

Can I modify tags and other properties of the AKS resources in the MC_* resource group?

Modifying and deleting the Azure-created tags and other properties of resources in the *MC_** resource group can lead to unexpected results such as scaling and upgrading errors. It is supported to create and modify additional custom tags, such as to assign a business unit or cost center. Modifying the resources under the *MC_** in the AKS cluster breaks the SLO.

What Kubernetes admission controllers does AKS support? Can admission controllers be added or removed?

AKS supports the following admission controllers:

- *NamespaceLifecycle*
- *LimitRanger*
- *ServiceAccount*
- *DefaultStorageClass*
- *DefaultTolerationSeconds*
- *MutatingAdmissionWebhook*
- *ValidatingAdmissionWebhook*
- *ResourceQuota*
- *DenyEscalatingExec*
- *AlwaysPullImages*

It is not currently possible to modify the list of admission controllers in AKS.

Is Azure Key Vault integrated with AKS?

AKS is not currently natively integrated with Azure Key Vault. However, the Azure Key Vault FlexVolume for Kubernetes project enables direct integration from Kubernetes pods to KeyVault secrets.

Can I run Windows Server containers on AKS?

To run Windows Server containers, you need to run Windows Server-based nodes. Windows Server-based nodes are not available in AKS at this time. You can, however, use Virtual Kubelet to schedule Windows containers on Azure Container Instances and manage them as part of your AKS cluster. For more information, see Use Virtual Kubelet with AKS.

Does AKS offer a service level agreement?

In a service level agreement (SLA), the provider agrees to reimburse the customer for the cost of the service if the published service level isn't met. Since AKS itself is free, there is no cost available to reimburse and thus no formal

SLA. However, AKS seeks to maintain availability of at least 99.5% for the Kubernetes API server.