

Table of Contents

[ACS with Kubernetes](#)

[Overview](#)

[About Container Service](#)

[Quickstarts](#)

[Kubernetes Linux](#)

[Tutorials](#)

[1 - Create container images](#)

[2 - Create container registry](#)

[3 - Create Kubernetes cluster](#)

[4 - Run application](#)

[5 - Scale application](#)

[6 - Update application](#)

[7 - Monitor with OMS](#)

[Samples](#)

[Azure CLI](#)

[Concepts](#)

[Secure containers](#)

[Service principal - Kubernetes](#)

[How-to guides](#)

[Kubernetes with Windows](#)

[Kubernetes web interface](#)

[Deploy Helm charts](#)

[Jenkins and ACS Kubernetes](#)

[Use Draft with ACR and ACS](#)

[Connect with an ACS cluster](#)

[Scale an ACS cluster](#)

[Load balance a Kubernetes ACS cluster](#)

[Monitor](#)

[Monitor with OMS](#)

[Monitor with Datadog](#)

[Monitor with Sysdig](#)

[Monitor with CoScale](#)

[Deploy Spring Boot apps](#)

[Deploy a Spring Boot Application on Linux in the Azure Container Service](#)

[Deploy a Spring Boot Application on a Kubernetes Cluster in the Azure Container Service](#)

[Deploy a Spring Boot app using the fabric8 Maven plugin](#)

[Reference](#)

[Azure CLI 2.0](#)

[REST](#)

[Resources](#)

[Azure Roadmap](#)

[FAQ](#)

[Templates - ACS Engine](#)

[Pricing](#)

[Region availability](#)

[Stack Overflow](#)

[Videos](#)

Introduction to Azure Container Service for Kubernetes

8/11/2017 • 1 min to read • [Edit Online](#)

Azure Container Service for Kubernetes makes it simple to create, configure, and manage a cluster of virtual machines that are preconfigured to run containerized applications. This enables you to use your existing skills, or draw upon a large and growing body of community expertise, to deploy and manage container-based applications on Microsoft Azure.

By using Azure Container Service, you can take advantage of the enterprise-grade features of Azure, while still maintaining application portability through Kubernetes and the Docker image format.

Using Azure Container Service for Kubernetes

Our goal with Azure Container Service is to provide a container hosting environment by using open-source tools and technologies that are popular among our customers today. To this end, we expose the standard Kubernetes API endpoints. By using these standard endpoints, you can leverage any software that is capable of talking to a Kubernetes cluster. For example, you might choose [kubectl](#), [helm](#), or [draft](#).

Creating a Kubernetes cluster using Azure Container Service

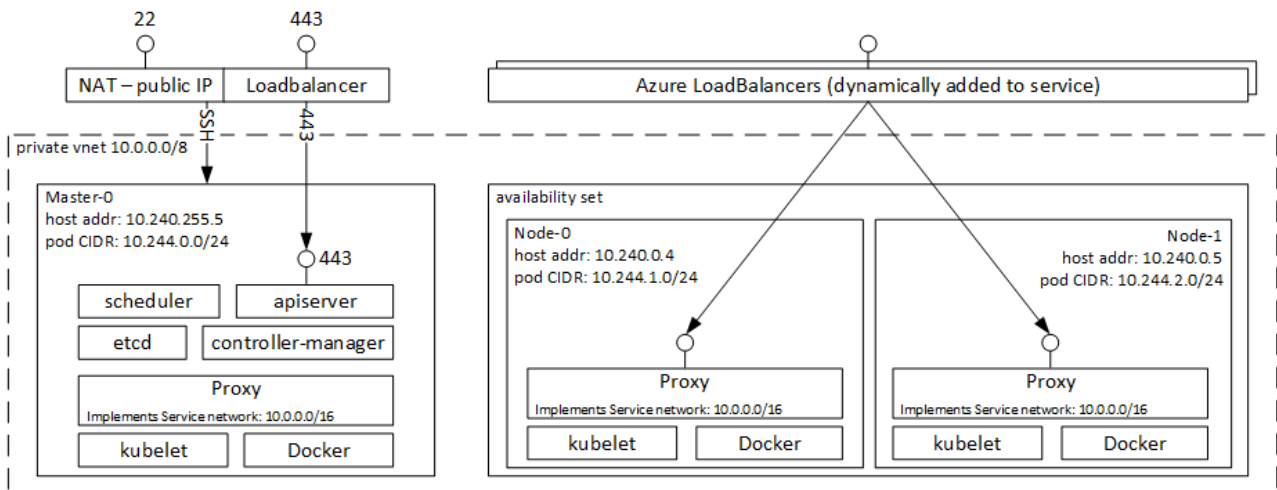
To begin using Azure Container Service, deploy an Azure Container Service cluster with the [Azure CLI 2.0](#) or via the portal (search the Marketplace for **Azure Container Service**). If you are an advanced user who needs more control over the Azure Resource Manager templates, you can use the open source [acs-engine](#) project to build your own custom Kubernetes cluster and deploy it via the `az` CLI.

Using Kubernetes

Kubernetes automates deployment, scaling, and management of containerized applications. It has a rich set of features including:

- Automatic binpacking
- Self-healing
- Horizontal scaling
- Service discovery and load balancing
- Automated rollouts and rollbacks
- Secret and configuration management
- Storage orchestration
- Batch execution

Architectural diagram of Kubernetes deployed via Azure Container Service:



Videos

Kubernetes Support in Azure Container Services (Azure Friday, January 2017):

Tools for Developing and Deploying Applications on Kubernetes (Azure OpenDev, June 2017):

Next steps

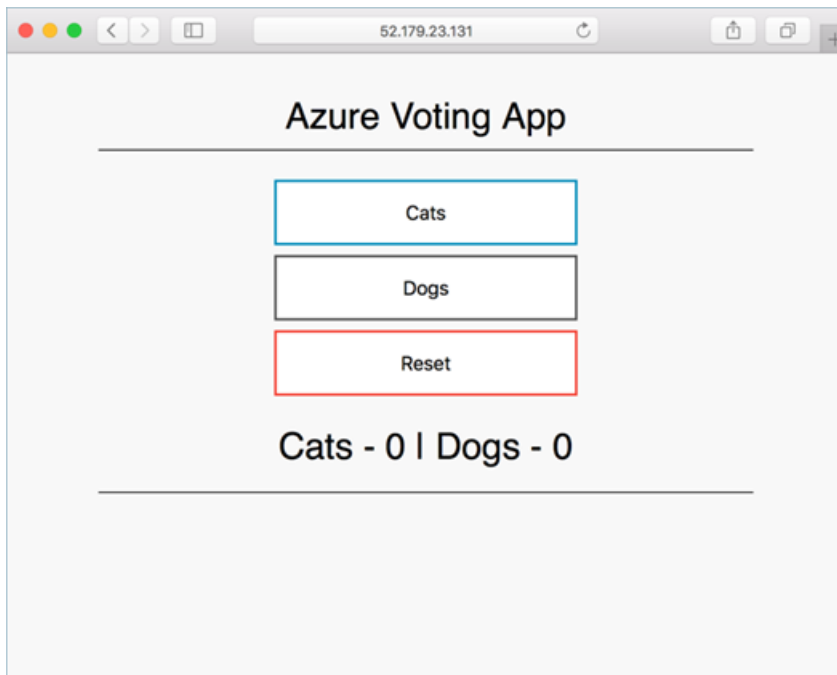
Explore the [Kubernetes Quickstart](#) to begin exploring Azure Container Service today.

Deploy Kubernetes cluster for Linux containers

9/19/2017 • 4 min to read • [Edit Online](#)

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

The example application used in this document is written in Python. The concepts and steps detailed here can be used to deploy any container image into a Kubernetes cluster. The code, Dockerfile, and pre-created Kubernetes manifest files related to this project are available on [GitHub](#).



This quick start assumes a basic understanding of Kubernetes concepts, for detailed information on Kubernetes see the [Kubernetes documentation](#).

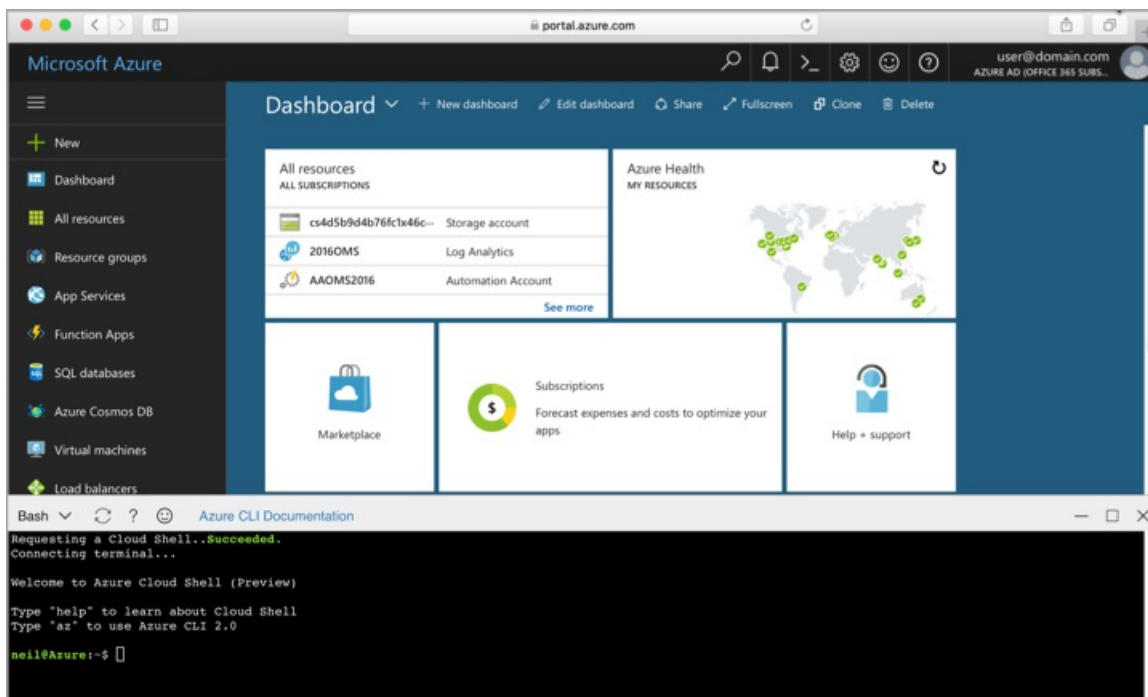
If you don't have an Azure subscription, create a [free account](#) before you begin.

Launch Azure Cloud Shell

The Azure Cloud Shell is a free Bash shell that you can run directly within the Azure portal. It has the Azure CLI preinstalled and configured to use with your account. Click the **Cloud Shell** button on the menu in the upper-right of the [Azure portal](#).



The button launches an interactive shell that you can use to run the steps in this topic:



If you choose to install and use the CLI locally, this quickstart requires that you are running the Azure CLI version 2.0.4 or later. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI 2.0](#).

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.

The following example creates a resource group named *myResourceGroup* in the *westeurope* location.

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

Output:

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

Create Kubernetes cluster

Create a Kubernetes cluster in Azure Container Service with the `az acs create` command. The following example creates a cluster named *myK8sCluster* with one Linux master node and three Linux agent nodes.

```
az acs create --orchestrator-type kubernetes --resource-group myResourceGroup --name myK8sCluster --generate-ssh-keys
```

In some cases, such as with a limited trial, an Azure subscription has limited access to Azure resources. If the deployment fails due to limited available cores, reduce the default agent count by adding `--agent-count 1` to the

[az acs create](#) command.

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 CloudShell, kubectl is already installed. If you want to install it locally, you can use the [az acs kubernetes install-cli](#) command.

To configure kubectl to connect to your Kubernetes cluster, run the [az acs kubernetes get-credentials](#) command. This step downloads credentials and configures the Kubernetes CLI to use them.

```
az acs kubernetes get-credentials --resource-group=myResourceGroup --name=myK8sCluster
```

To verify the connection to your cluster, use the [kubectl get](#) command to return a list of the cluster nodes.

```
kubectl get nodes
```

Output:

NAME	STATUS	AGE	VERSION
k8s-agent-14ad53a1-0	Ready	10m	v1.6.6
k8s-agent-14ad53a1-1	Ready	10m	v1.6.6
k8s-agent-14ad53a1-2	Ready	10m	v1.6.6
k8s-master-14ad53a1-0	Ready,SchedulingDisabled	10m	v1.6.6

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.

Create a file named `azure-vote.yml` and copy into it the following YAML. 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:redis-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 create` command to run the application.

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

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.

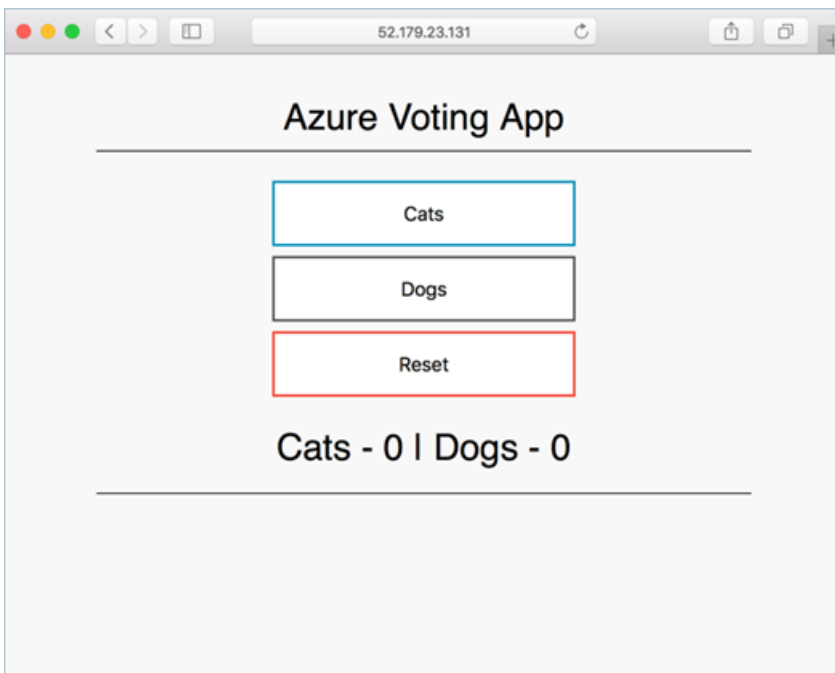
To monitor progress, use the [kubectl get service](#) command with the `--watch` argument.

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

Initially the **EXTERNAL-IP** for the *azure-vote-front* service appears as *pending*. 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  10.0.34.242  <pending>      80:30676/TCP    7s
azure-vote-front  10.0.34.242  52.179.23.131  80:30676/TCP    2m
```

You can now browse to the external IP address to see the Azure Vote App.



Delete cluster

When the cluster is no longer needed, you can 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
```

Get the code

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

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

[Manage an ACS Kubernetes cluster](#)

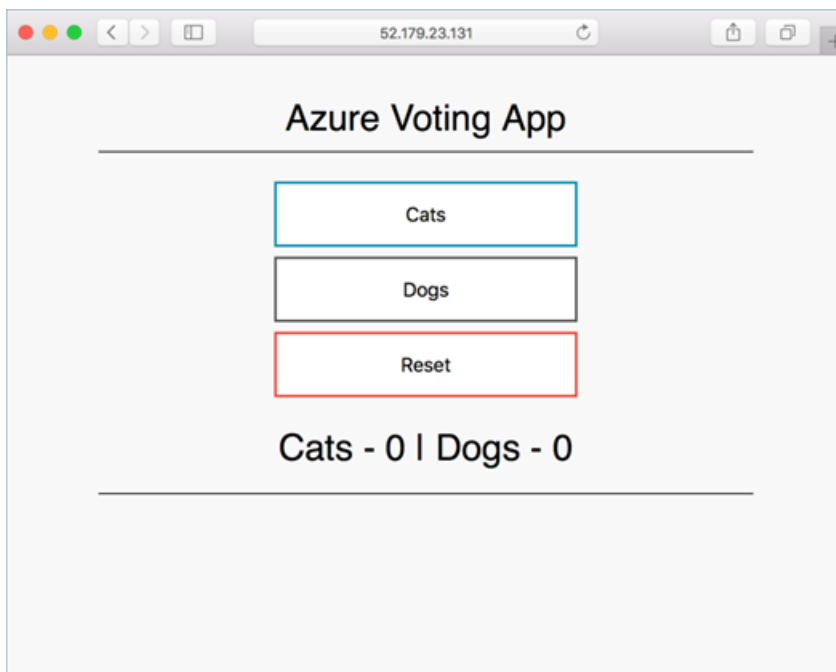
Create container images to be used with Azure Container Service

9/14/2017 • 2 min to read • [Edit Online](#)

In this tutorial, part one of seven, a multi-container application is prepared for use in Kubernetes. Steps completed include:

- Cloning application source from GitHub
- Creating a container image from the application source
- Testing the application in a local Docker environment

Once completed, the following application is accessible in your local development environment.



In subsequent tutorials, the container image is uploaded to an Azure Container Registry, and then run in an Azure hosted Kubernetes cluster.

Before you begin

This tutorial assumes a basic understanding of core Docker concepts such as containers, container images, and basic docker commands. If needed, see [Get started with Docker](#) for a primer on container basics.

To complete this tutorial, you need a Docker development environment. Docker provides packages that easily configure Docker on any [Mac](#), [Windows](#), or [Linux](#) system.

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 download a copy of the 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 the build out of container images and the deployment of multi-container applications.

Run the `docker-compose.yml` 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.

```
docker images
```

Notice that 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.

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.

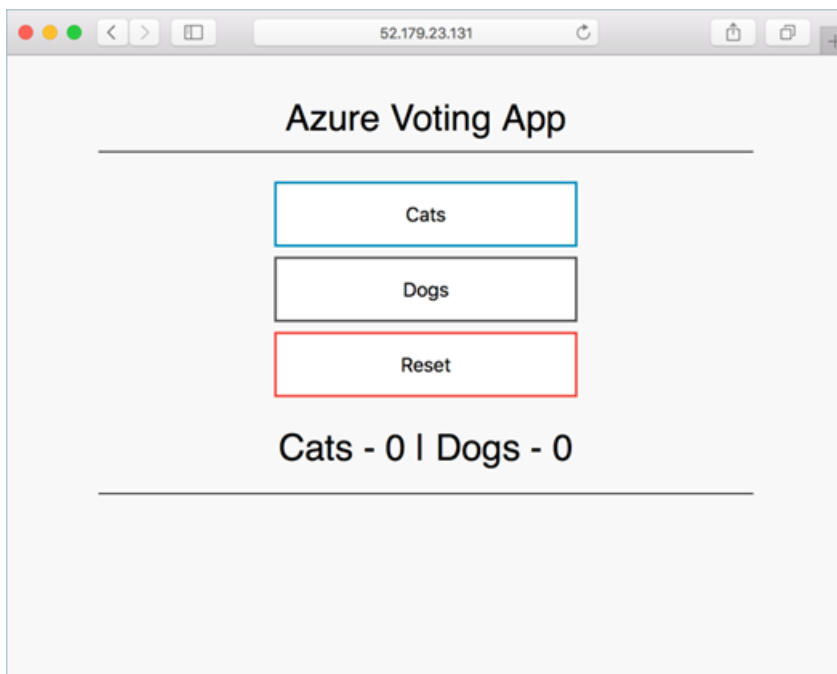
```
docker ps
```

Output:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
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

Browse to <http://localhost:8080> to see the running application.



Clean up resources

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

Run the following to stop the running containers.

```
docker-compose stop
```

Delete the stopped containers and resources with the following command.

```
docker-compose down
```

At completion, you have a container image that contains the Azure Vote application.

Next steps

In this tutorial, an application was tested and container images created for the application. The following steps were completed:

- Cloning the application source from GitHub
- Created a container image from application source
- Tested the application in a local Docker environment

Advance to the next tutorial to learn about storing container images in an Azure Container Registry.

[Push images to Azure Container Registry](#)

Deploy and use Azure Container Registry

9/14/2017 • 3 min to read • [Edit Online](#)

Azure Container Registry (ACR) is an Azure-based, private registry, for Docker container images. This tutorial, part two of seven, walks through deploying an Azure Container Registry instance, and pushing a container image to it. Steps completed include:

- Deploying an Azure Container Registry (ACR) instance
- Tagging a container image for ACR
- Uploading the image to ACR

In subsequent tutorials, this ACR instance is integrated with an Azure Container Service Kubernetes cluster.

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 2.0.4 or later. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI 2.0](#).

Deploy Azure Container Registry

When deploying 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 this example, a resource group named `myResourceGroup` is created in the `westeurope` region.

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

Create an Azure Container registry with the `az acr create` command. The name of a Container Registry **must be unique**.

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

Throughout the rest of this tutorial, we use `<acrname>` as a placeholder for the container registry name.

Container registry login

Use the `az acr login` command to log in to the ACR instance. You need to provide the unique name given to the container registry when it was created.

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

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

Tag container images

To see a list of current images, use the [docker images](#) command.

```
docker images
```

Output:

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

Each container image needs to be tagged with the loginServer name of the registry. This tag is used for routing when pushing container images to an image registry.

To get the loginServer name, run the following command.

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

Now, tag the `azure-vote-front` image with the loginServer of the container registry. Also, add `:redis-v1` to the end of the image name. This tag indicates the image version.

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

Once tagged, run [docker images](#) to verify the operation.

```
docker images
```

Output:

REPOSITORY	TAG	IMAGE ID	CREATED
SIZE			
azure-vote-front	latest	eaf2b9c57e5e	8 minutes ago
716 MB			
mycontainerregistry082.azurecr.io/azure-vote-front	redis-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

Push the `azure-vote-front` image to the registry.

Using the following example, replace the ACR loginServer name with the loginServer from your environment.

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

This takes a couple of minutes to complete.

List images in registry

To return a list of images that have been pushed to your Azure Container registry, use the [az acr repository list](#)

command. Update the command with the ACR instance name.

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

Output:

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

And then to see the tags for a specific image, use the [az acr repository show-tags](#) command.

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

Output:

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

At tutorial completion, the container image has been stored in a private Azure Container Registry instance. This image is deployed from ACR to a Kubernetes cluster in subsequent tutorials.

Next steps

In this tutorial, an Azure Container Registry was prepared for use in an ACS Kubernetes cluster. The following steps were completed:

- Deployed an Azure Container Registry instance
- Tagged a container image for ACR
- Uploaded the image to ACR

Advance to the next tutorial to learn about deploying a Kubernetes cluster in Azure.

[Deploy Kubernetes cluster](#)

Deploy a Kubernetes cluster in Azure Container Service

9/14/2017 • 2 min to read • [Edit Online](#)

Kubernetes provides a distributed platform for containerized applications. With Azure Container Service, provisioning of a production ready Kubernetes cluster is simple and quick. In this tutorial, part 3 of 7, an Azure Container Service Kubernetes cluster is deployed. Steps completed include:

- Deploying a Kubernetes ACS cluster
- Installation of the Kubernetes CLI (kubectl)
- Configuration of kubectl

In subsequent tutorials, the Azure Vote application is deployed to the cluster, scaled, updated, and Operations Management Suite is configured to monitor the Kubernetes cluster.

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](#).

Create Kubernetes cluster

Create a Kubernetes cluster in Azure Container Service with the `az acs create` command.

The following example creates a cluster named `myK8sCluster` in a Resource Group named `myResourceGroup`. This Resource Group was created in the [previous tutorial](#).

```
az acs create --orchestrator-type kubernetes --resource-group myResourceGroup --name myK8sCluster --generate-ssh-keys
```

In some cases, such as with a limited trial, an Azure subscription has limited access to Azure resources. If the deployment fails due to limited available cores, reduce the default agent count by adding `--agent-count 1` to the `az acs create` command.

After several minutes, the deployment completes, and returns json formatted information about the ACS deployment.

Install the kubectl CLI

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

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

If running in Linux or macOS, you may need to run with `sudo`. On Windows, ensure your shell has been run as administrator.

```
az acs kubernetes install-cli
```

On Windows, the default installation is `c:\program files (x86)\kubectl.exe`. You may need to add this file to the

Windows path.

Connect with kubectl

To configure kubectl to connect to your Kubernetes cluster, run the [az acs kubernetes get-credentials](#) command.

```
az acs kubernetes get-credentials --resource-group myResourceGroup --name myK8SCluster
```

To verify the connection to your cluster, run the [kubectl get nodes](#) command.

```
kubectl get nodes
```

Output:

NAME	STATUS	AGE	VERSION
k8s-agent-98dc3136-0	Ready	5m	v1.6.2
k8s-agent-98dc3136-1	Ready	5m	v1.6.2
k8s-agent-98dc3136-2	Ready	5m	v1.6.2
k8s-master-98dc3136-0	Ready,SchedulingDisabled	5m	v1.6.2

At tutorial completion, you have an ACS Kubernetes cluster ready for workloads. In subsequent tutorials, a multi-container application is deployed to this cluster, scaled out, updated, and monitored.

Next steps

In this tutorial, an Azure Container Service Kubernetes cluster was deployed. The following steps were completed:

- Deployed a Kubernetes ACS cluster
- Installed the Kubernetes CLI (kubectl)
- Configured kubectl

Advance to the next tutorial to learn about running application on the cluster.

[Deploy application in Kubernetes](#)

Run applications in Kubernetes

9/14/2017 • 2 min to read • [Edit Online](#)

In this tutorial, part four of seven, a sample application is deployed into a Kubernetes cluster. Steps completed include:

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

In subsequent tutorials, this application is scaled out, updated, and Operations Management Suite configured to monitor the Kubernetes cluster.

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.yml` 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](#).

Update manifest file

In this tutorial, Azure Container Registry (ACR) has been used to store a container image. Before running the application, the ACR login server name needs to be updated in the Kubernetes manifest file.

Get the ACR login server name with the [az acr list](#) command.

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

The manifest file has been pre-created with a login server name of `microsoft`. Open the file with any text editor. In this example, the file is opened with `vi`.

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

Replace `microsoft` with the 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:redis-v1
```

Save and close the file.

Deploy application

Use the [kubectl create](#) command to run the application. This command parses the manifest file and create the defined Kubernetes objects.

```
kubectl create -f azure-vote-all-in-one-redis.yml
```

Output:

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

Test application

A [Kubernetes service](#) is created which exposes the application to the internet. This process can take a few minutes.

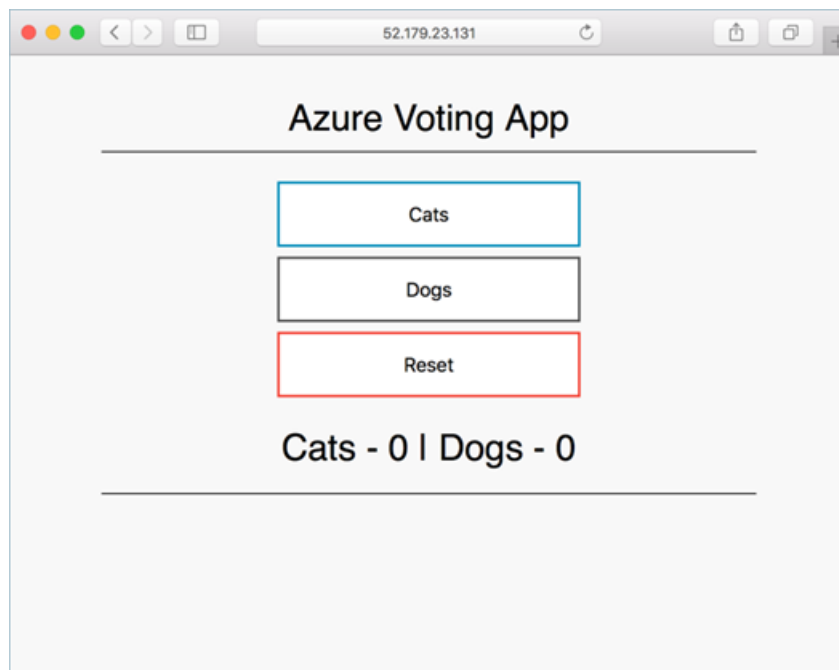
To monitor progress, use the [kubectl get service](#) command with the `--watch` argument.

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

Initially, the **EXTERNAL-IP** for the `azure-vote-front` service appears as `pending`. Once the EXTERNAL-IP address has changed from `pending` to an `IP address`, use `CTRL-C` to stop the kubectl watch process.

NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
azure-vote-front	10.0.42.158	<pending>	80:31873/TCP	1m
azure-vote-front	10.0.42.158	52.179.23.131	80:31873/TCP	2m

To see the application, browse to the external IP address.



Next steps

In this tutorial, the Azure vote application was deployed to an Azure Container Service Kubernetes cluster. Tasks completed include:

- Download Kubernetes manifest files

- Run the application in Kubernetes
- Tested the application

Advance to the next tutorial to learn about scaling both a Kubernetes application and the underlying Kubernetes infrastructure.

[Scale Kubernetes application and infrastructure](#)

Scale Kubernetes pods and Kubernetes infrastructure

9/14/2017 • 3 min to read • [Edit Online](#)

If you've been following the tutorials, you have a working Kubernetes cluster in Azure Container Service 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 agent nodes to change the cluster's capacity for hosting workloads. Tasks completed include:

- Manually scaling Kubernetes pods
- Configuring Autoscale pods running the app front end
- Scale the Kubernetes Azure agent nodes

In subsequent tutorials, the Azure Vote application is updated, and Operations Management Suite configured to monitor the Kubernetes cluster.

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](#).

Manually scale pods

Thus far, the Azure Vote front-end and Redis instance have been deployed, each with a single replica. To verify, run the `kubectl get` command.

```
kubectl get pods
```

Output:

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

Manually change the number of pods in the `azure-vote-front` deployment using the `kubectl scale` command. This example increases the number to 5.

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

Run `kubectl get pods` to verify that Kubernetes is creating the pods. After a minute or so, the additional pods are running:

```
kubectl get pods
```

Output:

NAME	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.

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. Here, if CPU utilization exceeds 50%, the autoscaler increases the pods to a maximum of 10.

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

To see the status of the autoscaler, run the following command:

```
kubectl get hpa
```

Output:

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 3.

Scale the agents

If you created your Kubernetes cluster using default commands in the previous tutorial, it has three agent nodes. You can adjust the number of agents manually if you plan more or fewer container workloads on your cluster. Use the `az acs scale` command, and specify the number of agents with the `--new-agent-count` parameter.

The following example increases the number of agent nodes to 4 in the Kubernetes cluster named `myK8sCluster`. The command takes a couple of minutes to complete.

```
az acs scale --resource-group=myResourceGroup --name=myK8sCluster --new-agent-count 4
```

The command output shows the number of agent nodes in the value of `agentPoolProfiles:count`:

```
{
  "agentPoolProfiles": [
    {
      "count": 4,
      "dnsPrefix": "myK8Scluster-myK8Scluster-e44f25-k8s-agents",
      "fqdn": "",
      "name": "agentpools",
      "vmSize": "Standard_D2_v2"
    }
  ],
  ...
}
```

Next steps

In this tutorial, you used different scaling features in your Kubernetes cluster. Tasks covered included:

- Manually scaling Kubernetes pods
- Configuring Autoscale pods running the app front end
- Scale the Kubernetes Azure agent nodes

Advance to the next tutorial to learn about updating application in Kubernetes.

[Update an application in Kubernetes](#)

Update an application in Kubernetes

9/14/2017 • 3 min 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. Tasks that you complete include:

- Updating the front-end application code
- Creating an updated container image
- Pushing the container image to Azure Container Registry
- Deploying the updated container image

In subsequent tutorials, Operations Management Suite is configured to monitor the Kubernetes cluster.

Before you begin

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

An application repository was also cloned which 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. Inside is a directory named `azure-vote` and a file named `docker-compose.yml`.

If you haven't completed these steps, and want to follow along, return to [Tutorial 1 – Create container images](#).

Update application

For this tutorial, a change is made to the application, and the updated application deployed to the Kubernetes cluster.

The application source code can be found inside of the `azure-vote` directory. Open the `config_file.cfg` file with any code or text editor. In this example `vi` is used.

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

Change the values for `VOTE1VALUE` and `VOTE2VALUE`, and then save the file.

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

Save and close the file.

Update container image

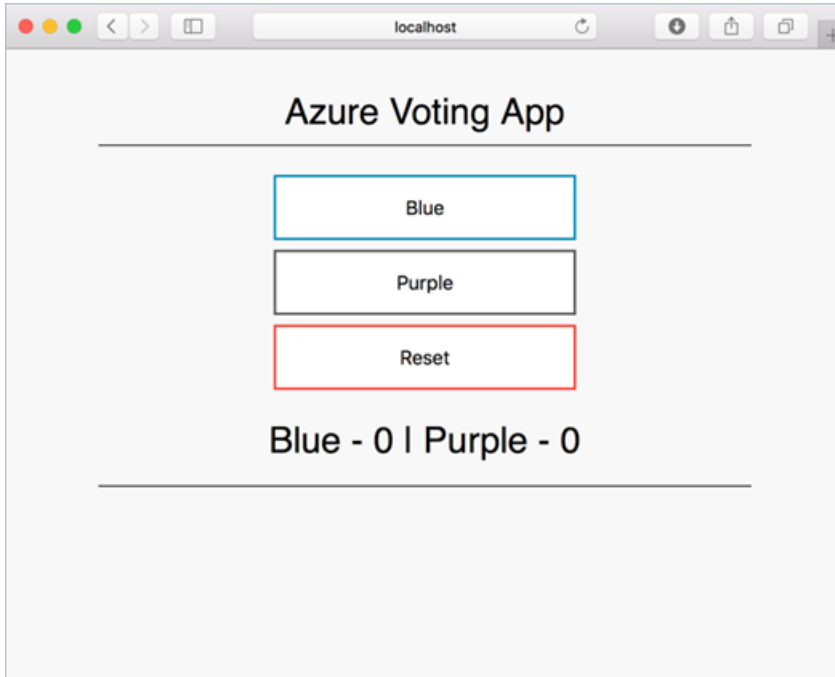
Use `docker-compose` to re-create the front-end image and run the updated application. The `--build` argument is

used to instruct Docker Compose to re-create the application image.

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

Test application locally

Browse to <http://localhost:8080> to see the updated application.



Tag and push images

Tag the `azure-vote-front` image with the loginServer of the container 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 Azure Container Registry login server name or public registry hostname. Also notice that the image version is updated to `redis-v2`.

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

Use `docker push` to upload the image to your registry. Replace `<acrLoginServer>` with your Azure Container Registry login server name.

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

Deploy update application

To ensure maximum uptime, multiple instances of the application pod must be running. Verify this configuration with the `kubectl get pod` command.

```
kubectl get pod
```

Output:

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 pods running the azure-vote-front image, scale the `azure-vote-front` deployment.

```
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.

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

Output:

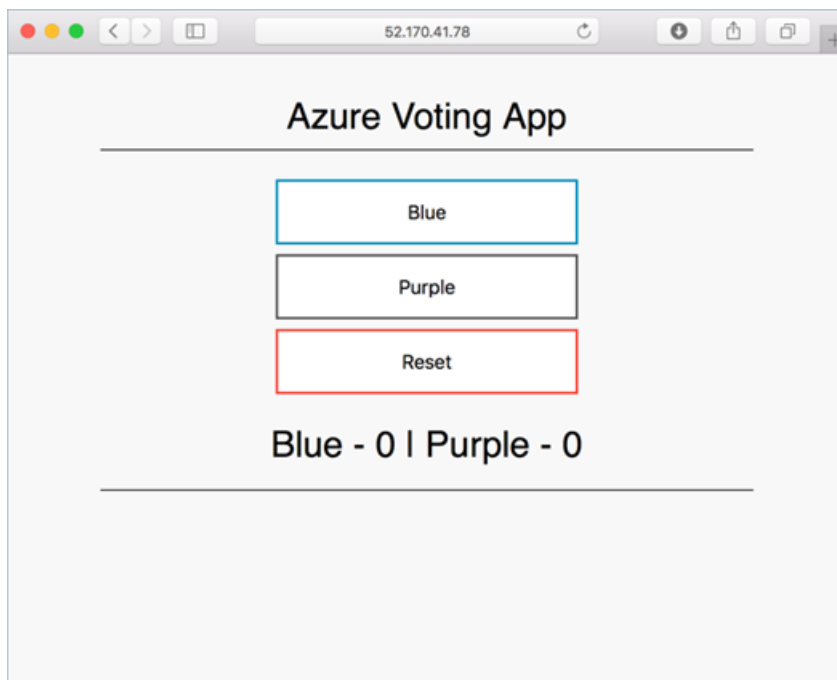
NAME	READY	STATUS	RESTARTS	AGE
azure-vote-back-2978095810-gq9g0	1/1	Running	0	5m
azure-vote-front-1297194256-tpjlg	1/1	Running	0	1m
azure-vote-front-1297194256-tptnx	1/1	Running	0	5m
azure-vote-front-1297194256-zktw9	1/1	Terminating	0	1m

Test updated application

Get the external IP address of the `azure-vote-front` service.

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

Browse to the IP address to see the updated application.



Next steps

In this tutorial, you updated an application and rolled out this update to a Kubernetes cluster. The following tasks were completed:

- Updated the front-end application code
- Created an updated container image
- Pushed the container image to Azure Container Registry
- Deployed the updated application

Advance to the next tutorial to learn about how to monitor Kubernetes with Operations Management Suite.

[Monitor Kubernetes with OMS](#)

Monitor a Kubernetes cluster with Operations Management Suite

9/25/2017 • 3 min to read • [Edit Online](#)

Monitoring your Kubernetes cluster and containers is critical, especially when you manage a production cluster at scale with multiple apps.

You can take advantage of several Kubernetes monitoring solutions, either from Microsoft or other providers. In this tutorial, you monitor your Kubernetes cluster by using the Containers solution in [Operations Management Suite](#), Microsoft's cloud-based IT management solution. (The OMS Containers solution is in preview.)

This tutorial, part seven of seven, covers the following tasks:

- Get OMS Workspace settings
- Set up OMS agents on the Kubernetes nodes
- Access monitoring information in the OMS portal or Azure portal

Before you begin

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

If you have not done these steps, and would like to follow along, return to [Tutorial 1 – Create container images](#).

Get Workspace settings

When you can access the [OMS portal](#), go to **Settings > Connected Sources > Linux Servers**. There, you can find the *Workspace ID* and a primary or secondary *Workspace Key*. Take note of these values, which you need to set up OMS agents on the cluster.

Set up OMS agents

Here is a YAML file to set up OMS agents on the Linux cluster nodes. It creates a Kubernetes [DaemonSet](#), which runs a single identical pod on each cluster node. The DaemonSet resource is ideal for deploying a monitoring agent.

Save the following text to a file named `oms-daemonset.yaml`, and replace the placeholder values for *myWorkspaceID* and *myWorkspaceKey* with your OMS Workspace ID and Key. (In production, you can encode these values as secrets.)

```

apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: omsagent
spec:
  template:
    metadata:
      labels:
        app: omsagent
        agentVersion: v1.3.4-127
        dockerProviderVersion: 10.0.0-25
    spec:
      containers:
        - name: omsagent
          image: "microsoft/oms"
          imagePullPolicy: Always
          env:
            - name: WSID
              value: myWorkspaceID
            - name: KEY
              value: myWorkspaceKey
            - name: DOMAIN
              value: opinsights.azure.com
          securityContext:
            privileged: true
          ports:
            - containerPort: 25225
              protocol: TCP
            - containerPort: 25224
              protocol: UDP
          volumeMounts:
            - mountPath: /var/run/docker.sock
              name: docker-sock
            - mountPath: /var/log
              name: host-log
          livenessProbe:
            exec:
              command:
                - /bin/bash
                - -c
                - ps -ef | grep omsagent | grep -v "grep"
            initialDelaySeconds: 60
            periodSeconds: 60
      volumes:
        - name: docker-sock
          hostPath:
            path: /var/run/docker.sock
        - name: host-log
          hostPath:
            path: /var/log

```

Create the DaemonSet with the following command:

```
kubectl create -f oms-daemonset.yaml
```

To see that the DaemonSet is created, run:

```
kubectl get daemonset
```

Output is similar to the following:

NAME	DESIRED	CURRENT	READY	UP-TO-DATE	AVAILABLE	NODE-SELECTOR	AGE
omsagent	3	3	3	0	3	<none>	5m

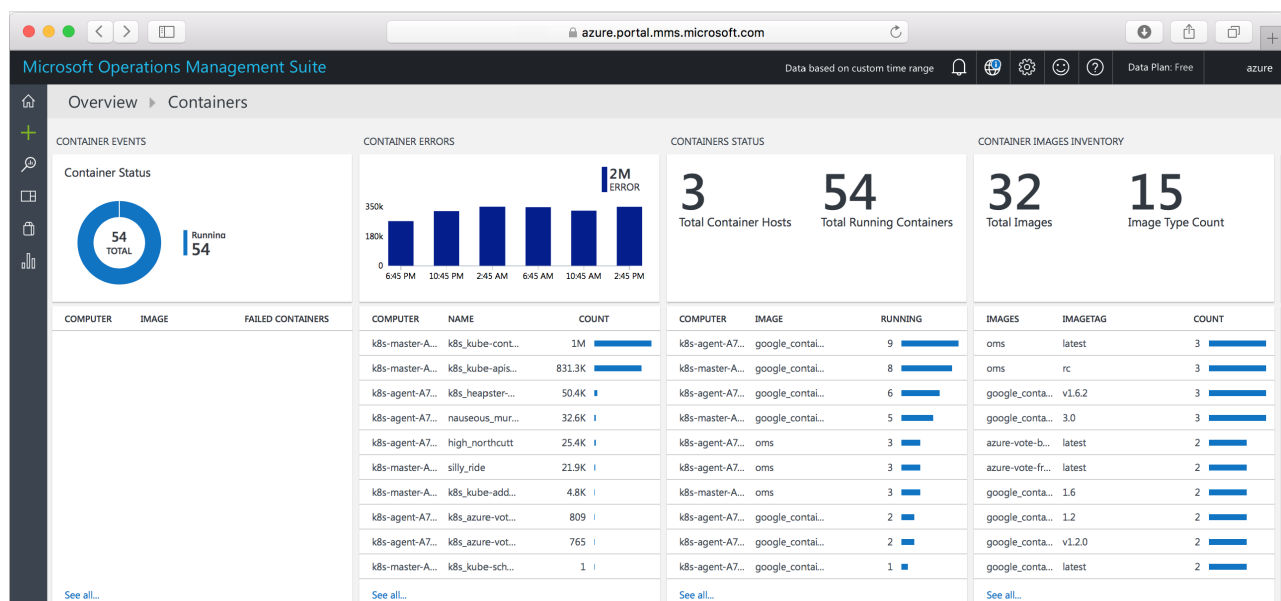
After the agents are running, it takes several minutes for OMS to ingest and process the data.

Access monitoring data

View and analyze the OMS container monitoring data with the [Container solution](#) in either the OMS portal or the Azure portal.

To install the Container solution using the [OMS portal](#), go to **Solutions Gallery**. Then add **Container Solution**. Alternatively, add the Containers solution from the [Azure Marketplace](#).

In the OMS portal, look for a **Containers** summary tile on the OMS dashboard. Click the tile for details including: container events, errors, status, image inventory, and CPU and memory usage. For more granular information, click a row on any tile, or perform a [log search](#).



Similarly, in the Azure portal, go to **Log Analytics** and select your workspace name. To see the **Containers** summary tile, click **Solutions > Containers**. To see details, click the tile.

See the [Azure Log Analytics documentation](#) for detailed guidance on querying and analyzing monitoring data.

Next steps

In this tutorial, you monitored your Kubernetes cluster with OMS. Tasks covered included:

- Get OMS Workspace settings
- Set up OMS agents on the Kubernetes nodes
- Access monitoring information in the OMS portal or Azure portal

Follow this link to see pre-built script samples for Container Service.

[Azure Container Service script samples](#)

Azure CLI Samples for Azure Container Service

8/11/2017 • 1 min to read • [Edit Online](#)

The following table includes links to bash scripts built using the Azure CLI.

Create virtual machines	
Create an ACS Kubernetes Linux cluster	Create a Kubernetes cluster for Linux based containers.
Create an ACS Kubernetes Windows cluster	Create a Kubernetes cluster for Windows based containers.
Scale an ACS cluster	Scale an ACS cluster.

Securing Docker containers in Azure Container Service

8/11/2017 • 5 min to read • [Edit Online](#)

This article introduces considerations and recommendations for securing Docker containers deployed in Azure Container Service. Many of these considerations apply generally to Docker containers deployed in Azure or other environments.

Image security

Containers are built from images that are stored in one or more repositories. These repositories can belong to public or private container registries. An example of a public registry is [Docker Hub](#). An example of a private registry is the [Docker Trusted Registry](#), which can be installed on-premises or in a virtual private cloud. There are also cloud-based private container registry services including [Azure Container Registry](#).

Public and private images

In general, as with any publicly published software package, a publicly available container image does not guarantee security. Container images consist of multiple software layers, and each software layer could have vulnerabilities. It is key to understand the origin of the container image, including the owner of the image (to determine if it is a reliable source or not), the software layers it consists of, and the software versions.

For example, if you go to the official [nginx repository](#) in Docker Hub and navigate to the **Tags** tab, you see color-coded vulnerabilities in each image. Each color depicts the vulnerability of a software layer of the image. For more about vulnerability scanning in Docker Hub, see [Understanding official repos on Docker Hub](#).

OFFICIAL REPOSITORY nginx ☆ Last pushed: 2 hours ago		
Repo Info	Tags	
Scanned Images ⓘ		
1.10 Compressed size: 71 MB Scanned 2 hours ago	ⓘ This image has vulnerabilities 	
stable Compressed size: 71 MB Scanned 2 hours ago	ⓘ This image has vulnerabilities 	
1.10.3 Compressed size: 71 MB Scanned 2 hours ago	ⓘ This image has vulnerabilities 	
alpine Compressed size: 17 MB Scanned 2 hours ago	ⓘ This image has vulnerabilities 	
1.11-alpine Compressed size: 17 MB Scanned 2 hours ago	ⓘ This image has vulnerabilities 	
1-alpine Compressed size: 17 MB Scanned 2 hours ago	ⓘ This image has vulnerabilities 	

Enterprises care deeply about security, and to protect themselves from security attack should store and retrieve images from a private registry, such as Azure Container Registry or Docker Trusted Registry. In addition to providing a managed private registry, Azure Container Registry supports [service principal-based authentication](#) through Azure Active Directory for basic authentication flows, including role-based access for read-only, write, and

owner permissions.

Image security scanning

Even when using a private registry, it is a good idea to use image scanning solutions for additional security validation. Each software layer in a container image is potentially prone to vulnerabilities independent of other layers in the container image. As increasingly companies start deploying their production workloads based on container technologies, image scanning becomes important to ensure prevention of security threats against their organizations.

Security monitoring and scanning solutions such as [Twistlock](#) and [Aqua Security](#), among others, can be used to scan container images in a private registry and identify potential vulnerabilities. It is important to understand the depth of scanning that the different solutions provide. For example, some solutions might only cross-verify image layers against known vulnerabilities. These solutions might not be able to verify image-layer software built through certain package manager software. Other solutions have deeper scanning integration and can find vulnerabilities in any packaged software.

Production deployment rules and audit

Once an application is deployed in production, it is essential to set a few rules to ensure that images used in production environments are secure and contain no vulnerabilities.

- As a rule, images with vulnerabilities, even minor, should not be allowed to run in a production environment. In addition, all images deployed in production should ideally be saved in a private registry accessible to a select few. It is also important to keep the number of production images small to ensure that they can be managed effectively.
- Since it is hard to pinpoint the origin of software from a publicly available container image, it is a good practice to build images from source to ensure knowledge of the origin of the layer. When a vulnerability surfaces in a self-built container image, customers can find a quicker path to a resolution. With a public image, customers would need to find the root of a public image to fix it or obtain another secure image from the publisher.
- A thoroughly scanned image deployed in production is not guaranteed to be up to date for the lifetime of the application. Security vulnerabilities might be reported for layers of the image that were not previously known or were introduced after the production deployment. Periodic auditing of images deployed in production is necessary to identify images that are out of date or have not been updated in a while. One might use blue-green deployment methodologies and rolling upgrade mechanisms to update container images without downtime. Image scanning can be done with tools described in the preceding section.
- A continuous integration (CI) pipeline to build images and integrated security scanning can help maintain secure private registries with secure container images. The vulnerability scanning built into the CI solution ensures that images that pass all the tests are pushed to the private registry from which production workloads are deployed. A CI pipeline failure ensures that vulnerable images are not pushed into the private registry used for production workload deployments. It also automates image security scanning if there are a significant number of images. Otherwise, manually auditing images for security vulnerabilities can be painstakingly lengthy and error prone.

Host-level container isolation

When a customer deploys container applications on Azure resources, they are deployed at a subscription level in resource groups and are not multi-tenant. This means that if a customer shares a subscription with others, there are no boundaries that can be built between two deployments in the same subscription. Therefore, container-level security is not guaranteed.

It is also critical to understand that containers share the kernel and the resources of the host (which in Azure Container Service is an Azure VM in a cluster). Therefore, containers running in production must be run in non-

privileged user mode. Running a container with root privileges can compromise the entire environment. With root-level access in a container, a hacker can gain access to the full root privileges on the host. In addition, it is important to run containers with read-only file systems. This prevents someone who has access to the compromised container to write malicious scripts to the file system and gain access to other files. Similarly, it is important to limit the resources (such as memory, CPU, and network bandwidth) allocated to a container. This helps prevent hackers from hogging resources and pursuing illegal activities such as credit card fraud or bitcoin mining, which could prevent other containers from running on the host or cluster.

Orchestrator considerations

Each orchestrator available in Azure Container Service has its own security considerations. For example, you should limit direct SSH access to orchestrator nodes in Container Service. Instead, you should use each orchestrator's UI or command-line tools (such as `kubect1` for Kubernetes) to manage the container environment without accessing the hosts. For more information, see [Make a remote connection to a Kubernetes, DC/OS, or Docker Swarm cluster](#).

For additional orchestrator-specific security information, see the following resources:

- **Kubernetes:** [Security Best Practices for Kubernetes Deployment](#)
- **DC/OS:** [Securing Your Cluster](#)
- **Docker Swarm:** [Docker Security](#)

Next steps

- For more about Docker architecture and container security, see [Introduction to Container Security](#).
- For information about Azure platform security, see the [Azure Security Center](#).

Set up an Azure AD service principal for a Kubernetes cluster in Container Service

9/25/2017 • 4 min to read • [Edit Online](#)

In Azure Container Service, a Kubernetes cluster requires an [Azure Active Directory service principal](#) to interact with Azure APIs. The service principal is needed to dynamically manage resources such as [user-defined routes](#) and the [Layer 4 Azure Load Balancer](#).

This article shows different options to set up a service principal for your Kubernetes cluster. For example, if you installed and set up the [Azure CLI 2.0](#), you can run the `az acs create` command to create the Kubernetes cluster and the service principal at the same time.

Requirements for the service principal

You can use an existing Azure AD service principal that meets the following requirements, or create a new one.

- **Scope:** the subscription used to deploy the cluster.
- **Role: Contributor**
- **Client secret:** must be a password. Currently, you can't use a service principal set up for certificate authentication.

IMPORTANT

To create a 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. To see if you have the required permissions, [check in the Portal](#).

Option 1: Create a service principal in Azure AD

If you want to create an Azure AD service principal before you deploy your Kubernetes cluster, Azure provides several methods.

The following example commands show you how to do this with the [Azure CLI 2.0](#). You can alternatively create a service principal using [Azure PowerShell](#), the [portal](#), or other methods.

```
az login

az account set --subscription "mySubscriptionID"

az group create --name "myResourceGroup" --location "westus"

az ad sp create-for-rbac --role="Contributor" --scopes="/subscriptions/mySubscriptionID"
```

Output is similar to the following (shown here redacted):

```
{
  "appId": "2a120104-xxxx-xxxx-xxxx-64fce3f77756",
  "displayName": "azure-cli-2017-09-21-22-00-41",
  "name": "http://azure-cli-2017-09-21-22-00-41",
  "password": "39969d0b-xxxx-xxxx-xxxx-5ace3399701f",
  "tenant": "72f988bf-xxxx-xxxx-xxxx-2d7cd011db47"
}
```

Highlighted are the **client ID** (`appId`) and the **client secret** (`password`) that you use as service principal parameters for cluster deployment.

Specify service principal when creating the Kubernetes cluster

Provide the **client ID** (also called the `appId`, for Application ID) and **client secret** (`password`) of an existing service principal as parameters when you create the Kubernetes cluster. Make sure the service principal meets the requirements at the beginning this article.

You can specify these parameters when deploying the Kubernetes cluster using the [Azure Command-Line Interface \(CLI\) 2.0](#), [Azure portal](#), or other methods.

TIP

When specifying the **client ID**, be sure to use the `appId`, not the `ObjectId`, of the service principal.

The following example shows one way to pass the parameters with the Azure CLI 2.0. This example uses the [Kubernetes quickstart template](#).

1. [Download](#) the template parameters file `azuredeploy.parameters.json` from GitHub.
2. To specify the service principal, enter values for `servicePrincipalClientId` and `servicePrincipalClientSecret` in the file. (You also need to provide your own values for `dnsNamePrefix` and `sshRSAPublicKey`. The latter is the SSH public key to access the cluster.) Save the file.

```
{
  "$schema": "http://schema.management.azure.com/schemas/2015-01-01/deploymentParameters.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "dnsNamePrefix": {
      "value": "myClusterName"
    },
    "adminUsername": {
      "value": "azureuser"
    },
    "sshRSAPublicKey": {
      "value": "ssh-rsa mySSHkeystringxxxxxxxxxxxxxxxxxxxxx azureuser@contoso"
    },
    "servicePrincipalClientId": {
      "value": "2a120104-xxxx-xxxx-xxxx-64fce3f77756"
    },
    "servicePrincipalClientSecret": {
      "value": "39969d0b-xxxx-xxxx-xxxx-5ace3399701f"
    },
    "orchestratorType": {
      "value": "Kubernetes"
    }
  }
}
```

3. Run the following command, using `--parameters` to set the path to the `azuredeploy.parameters.json` file. This command deploys the cluster in a resource group you create called `myResourceGroup` in the West US region.

```
az login

az account set --subscription "mySubscriptionID"

az group create --name "myResourceGroup" --location "westus"

az group deployment create -g "myResourceGroup" --template-uri
"https://raw.githubusercontent.com/Azure/azure-quickstart-templates/master/101-acs-
kubernetes/azuredeploy.json" --parameters @azuredeploy.parameters.json
```

Option 2: Generate a service principal when creating the cluster with

```
az acs create
```

If you run the `az acs create` command to create the Kubernetes cluster, you have the option to generate a service principal automatically.

As with other Kubernetes cluster creation options, you can specify parameters for an existing service principal when you run `az acs create`. However, when you omit these parameters, the Azure CLI creates one automatically for use with Container Service. This takes place transparently during the deployment.

The following command creates a Kubernetes cluster and generates both SSH keys and service principal credentials:

```
az acs create -n myClusterName -d myDNSPrefix -g myResourceGroup --generate-ssh-keys --orchestrator-type
kubernetes
```

IMPORTANT

If your account doesn't have the Azure AD and subscription permissions to create a service principal, the command generates an error similar to `Insufficient privileges to complete the operation.`

Additional considerations

- If you don't have permissions to create a service principal in your subscription, you might need to ask your Azure AD or subscription administrator to assign the necessary permissions, or ask them for a service principal to use with Azure Container Service.
- 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 specifying the service principal **Client ID**, you can use the value of the `appId` (as shown in this article) or the corresponding service principal `name` (for example, `https://www.contoso.org/example`).
- On the master and agent VMs in the Kubernetes cluster, the service principal credentials are stored in the file `/etc/kubernetes/azure.json`.
- When you use the `az acs create` command to generate the service principal automatically, the service principal credentials are written to the file `~/azure/acsServicePrincipal.json` on the machine used to run the command.

- When you use the `az acs create` command to generate the service principal automatically, the service principal can also authenticate with an [Azure container registry](#) created in the same subscription.

Next steps

- [Get started with Kubernetes](#) in your container service cluster.
- To troubleshoot the service principal for Kubernetes, see the [ACS Engine documentation](#).

Deploy Kubernetes cluster for Windows containers

8/11/2017 • 4 min to read • [Edit Online](#)

The Azure CLI is used to create and manage Azure resources from the command line or in scripts. This guide details using the Azure CLI to deploy a [Kubernetes](#) cluster in [Azure Container Service](#). Once the cluster is deployed, you connect to it with the Kubernetes `kubect1` command-line tool, and you deploy your first Windows container.

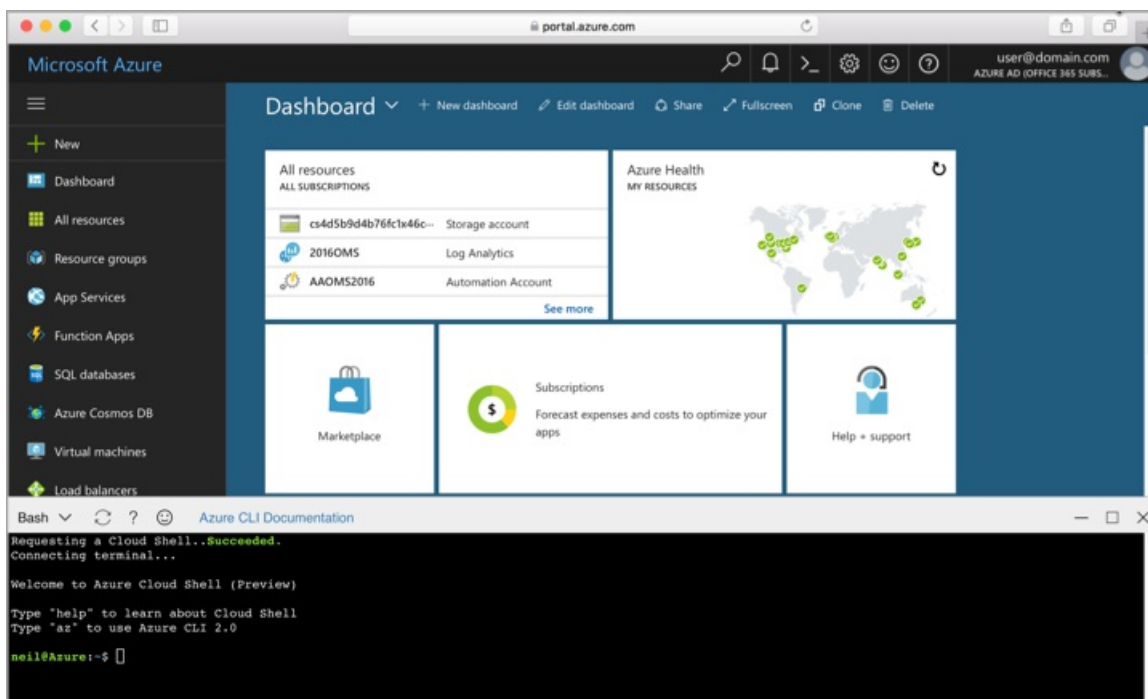
If you don't have an Azure subscription, create a [free account](#) before you begin.

Launch Azure Cloud Shell

The Azure Cloud Shell is a free Bash shell that you can run directly within the Azure portal. It has the Azure CLI preinstalled and configured to use with your account. Click the **Cloud Shell** button on the menu in the upper-right of the [Azure portal](#).



The button launches an interactive shell that you can use to run the steps in this topic:



If you choose to install and use the CLI locally, this quickstart requires that you are running the Azure CLI version 2.0.4 or later. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI 2.0](#).

NOTE

Support for Windows containers on Kubernetes in Azure Container Service is in preview.

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.

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

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

Create Kubernetes cluster

Create a Kubernetes cluster in Azure Container Service with the [az acs create](#) command.

The following example creates a cluster named *myK8sCluster* with one Linux master node and two Windows agent nodes. This example creates SSH keys needed to connect to the Linux master. This example uses *azureuser* for an administrative user name and *myPassword12* as the password on the Windows nodes. Update these values to something appropriate to your environment.

```
az acs create --orchestrator-type=kubernetes \
  --resource-group myResourceGroup \
  --name=myK8sCluster \
  --agent-count=2 \
  --generate-ssh-keys \
  --windows --admin-username azureuser \
  --admin-password myPassword12
```

After several minutes, the command completes, and shows you information about your deployment.

Install kubectl

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

If you're using Azure CloudShell, `kubectl` is already installed. If you want to install it locally, you can use the [az acs kubernetes install-cli](#) command.

The following Azure CLI example installs `kubectl` to your system. On Windows, run this command as an administrator.

```
az acs kubernetes install-cli
```

Connect with kubectl

To configure `kubectl` to connect to your Kubernetes cluster, run the [az acs kubernetes get-credentials](#) command. The following example downloads the cluster configuration for your Kubernetes cluster.

```
az acs kubernetes get-credentials --resource-group=myResourceGroup --name=myK8sCluster
```

To verify the connection to your cluster from your machine, try running:

```
kubectl get nodes
```

`kubectl` lists the master and agent nodes.

NAME	STATUS	AGE	VERSION
k8s-agent-98dc3136-0	Ready	5m	v1.5.3
k8s-agent-98dc3136-1	Ready	5m	v1.5.3
k8s-master-98dc3136-0	Ready,SchedulingDisabled	5m	v1.5.3

Deploy a Windows IIS container

You can run a Docker container inside a Kubernetes *pod*, which contains one or more containers.

This basic example uses a JSON file to specify a Microsoft Internet Information Server (IIS) container, and then creates the pod using the `kubectl apply` command.

Create a local file named `iis.json` and copy the following text. This file tells Kubernetes to run IIS on Windows Server 2016 Nano Server, using a public container image from [Docker Hub](#). The container uses port 80, but initially is only accessible within the cluster network.

```
{
  "apiVersion": "v1",
  "kind": "Pod",
  "metadata": {
    "name": "iis",
    "labels": {
      "name": "iis"
    }
  },
  "spec": {
    "containers": [
      {
        "name": "iis",
        "image": "nanoserver/iis",
        "ports": [
          {
            "containerPort": 80
          }
        ]
      }
    ],
    "nodeSelector": {
      "beta.kubernetes.io/os": "windows"
    }
  }
}
```

To start the pod, type:

```
kubectl apply -f iis.json
```

To track the deployment, type:

```
kubectl get pods
```

While the pod is deploying, the status is `ContainerCreating`. It can take a few minutes for the container to enter the `Running` state.

NAME	READY	STATUS	RESTARTS	AGE
iis	1/1	Running	0	32s

View the IIS welcome page

To expose the pod to the world with a public IP address, type the following command:

```
kubectl expose pods iis --port=80 --type=LoadBalancer
```

With this command, Kubernetes creates a service and an [Azure load balancer rule](#) with a public IP address for the service.

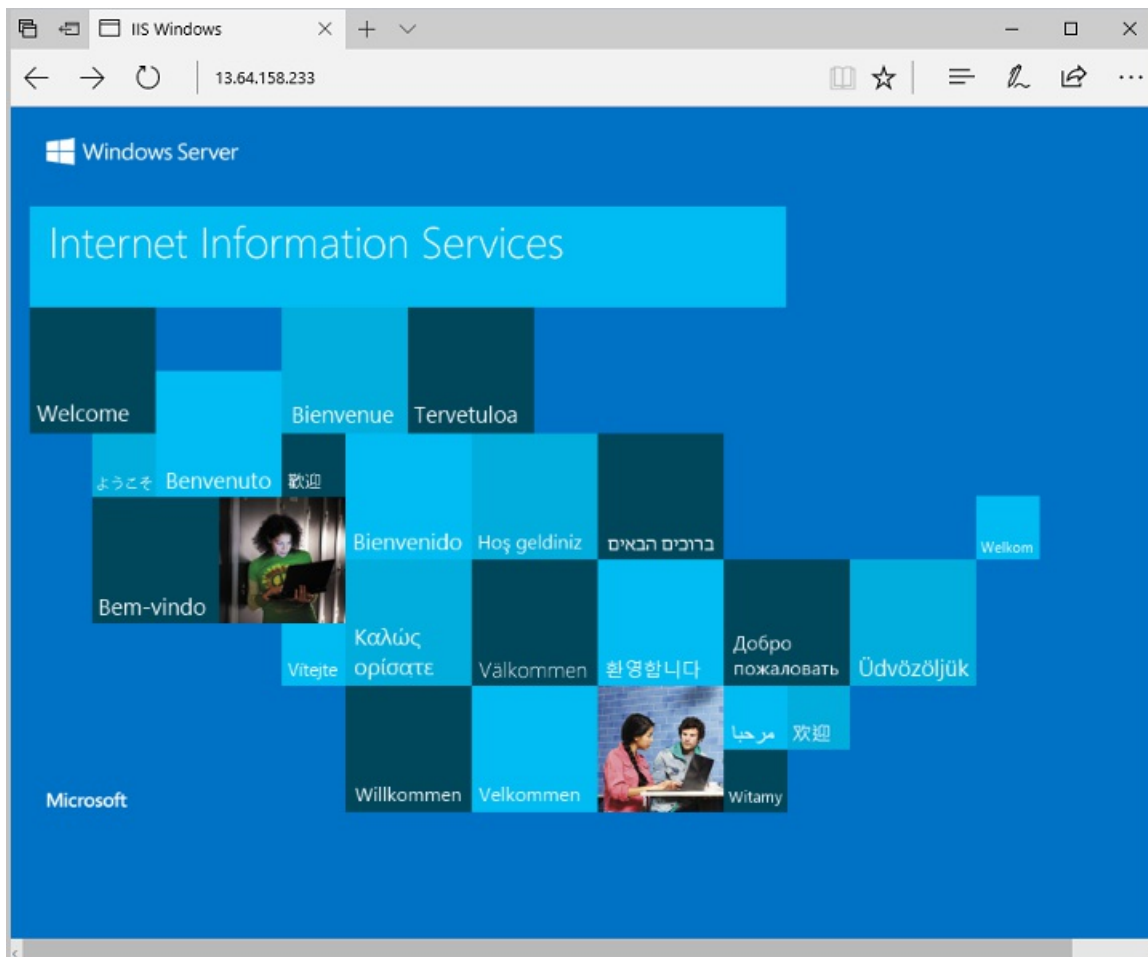
Run the following command to see the status of the service.

```
kubectl get svc
```

Initially the IP address appears as `pending`. After a few minutes, the external IP address of the `iis` pod is set:

NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	10.0.0.1	<none>	443/TCP	21h
iis	10.0.111.25	13.64.158.233	80/TCP	22m

You can use a web browser of your choice to see the default IIS welcome page at the external IP address:



Delete cluster

When the cluster is no longer needed, you can use the [az group delete](#) command to remove the resource group, container service, and all related resources.

```
az group delete --name myResourceGroup
```

Next steps

In this quick start, you deployed a Kubernetes cluster, connected with `kubect1`, and deployed a pod with an IIS container. To learn more about Azure Container Service, continue to the Kubernetes tutorial.

[Manage an ACS Kubernetes cluster](#)

Using the Kubernetes web UI with Azure Container Service

8/11/2017 • 2 min to read • [Edit Online](#)

Prerequisites

This walkthrough assumes that you have [created a Kubernetes cluster using Azure Container Service](#).

It also assumes that you have the Azure CLI 2.0 and `kubectl` tools installed.

You can test if you have the `az` tool installed by running:

```
$ az --version
```

If you don't have the `az` tool installed, there are instructions [here](#).

You can test if you have the `kubectl` tool installed by running:

```
$ kubectl version
```

If you don't have `kubectl` installed, you can run:

```
$ az acs kubernetes install-cli
```

Overview

Connect to the web UI

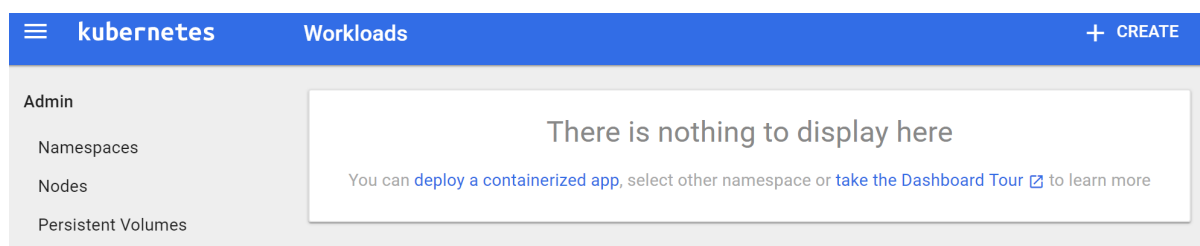
You can launch the Kubernetes web UI by running:

```
$ az acs kubernetes browse -g [Resource Group] -n [Container service instance name]
```

This should open a web browser configured to talk to a secure proxy connecting your local machine to the Kubernetes web UI.

Create and expose a service

1. In the Kubernetes web UI, click **Create** button in the upper right window.



A dialog box opens where you can start creating your application.

2. Give it the name `hello-nginx`. Use the `nginx` [container from Docker](#) and deploy three replicas of this web service.

Deploy a Containerized App

☒ Specify app details below
 ☐ Upload a YAML or JSON file

To learn more, [take the Dashboard Tour](#)

An 'app' label with this value will be added to the Deployment and Service that get deployed. [Learn more](#)

Enter the URL of a public image on any registry, or a private image hosted on Docker Hub or Google Container Registry. [Learn more](#)

A Deployment will be created to maintain the desired number of pods across your cluster. [Learn more](#)

App name *

hello-nginx

11 / 24

Container image *

nginx

Number of pods *

3

- Under **Service**, select **External** and enter port 80.

This setting load-balances traffic to the three replicas.

Service *

External

Port *

80

Target port *

80

Protocol *

TCP

Optionally, an internal or external Service can be defined to map an incoming Port to a target Port seen by the container. The internal DNS name for this Service will be: **hello-nginx**. [Learn more](#)


- Click **Deploy** to deploy these containers and services.


DEPLOY




CANCEL

View your containers

After you click **Deploy**, the UI shows a view of your service as it deploys:

Deployments						
Name	Labels	Pods	Age	Images		
 hello-nginx	app: hello-nginx	0 / 3	-	nginx		

Replica sets						
Name	Labels	Pods	Age	Images		
 hello-nginx-1648616287	app: hello-nginx pod-template-hash: 1...	0 / 3	-	nginx		

Pods						
Name	Status	Restarts	Age	Cluster IP	CPU (cores)	Memory (bytes)
 hello-nginx...	Waiting: ContainerCrē	0	-	-	-	-
 hello-nginx...	Waiting: ContainerCrē	0	-	-	-	-
 hello-nginx...	Waiting: ContainerCrē	0	-	-	-	-

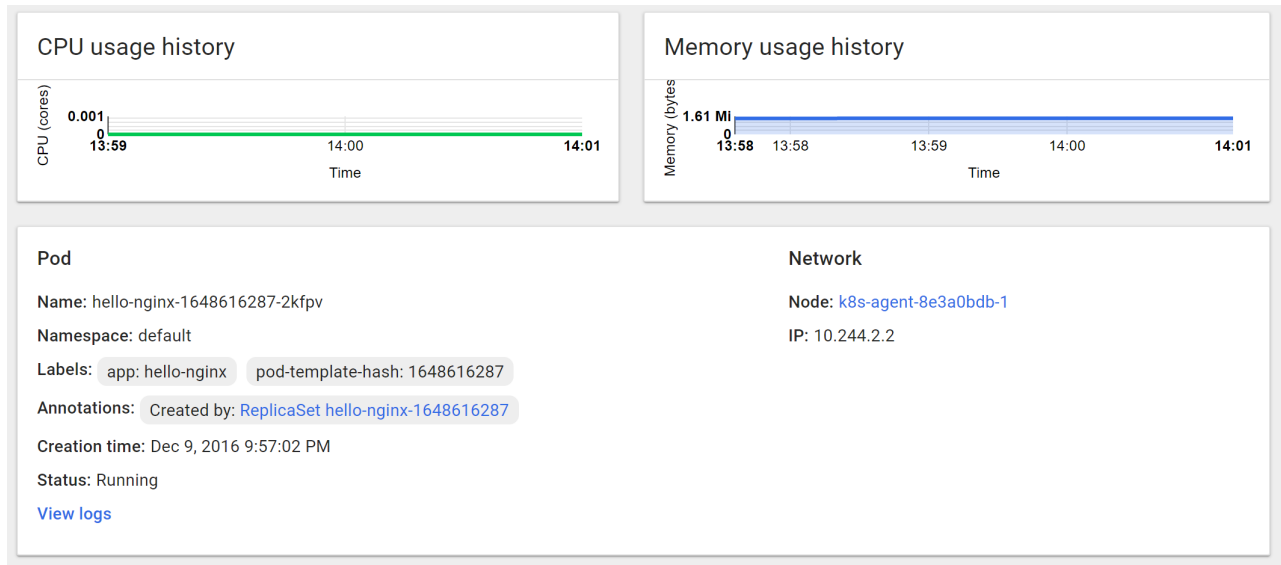
You can see the status of each Kubernetes object in the circle on the left-hand side of the UI, under **Pods**. If it is a partially full circle, then the object is still deploying. When an object is fully deployed, it displays a green check mark:



Once everything is running, click one of your pods to see details about the running web service.

Pods						
Name	Status	Restarts	Age	Cluster IP	CPU (cores)	Memory (bytes)
hello-nginx-1...	Running	0	2 minutes	10.244.2.2	<div><div></div></div> 0	<div><div></div></div> 1.430 Mi

In the **Pods** view, you can see information about the containers in the pod as well as the CPU and memory resources used by those containers:



If you don't see the resources, you may need to wait a few minutes for the monitoring data to propagate.

To see the logs for your container, click **View logs**.

Logs from hello-nginx in hello-nginx-1648616287-2kfpv

```
2016-12-09T22:28:18.460054024Z 10.244.2.1 - - [09/Dec/2016:22:28:18 +0000] "GET / HTTP/1.1" 304 0
"http://127.0.0.1:8001/api/v1/proxy/namespaces/kube-system/services/kubernetes-dashboard/" "Mozilla/5.0 (Windows NT
10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/54.0.2840.99 Safari/537.36" "-"
2016-12-09T22:28:19.219043156Z 10.244.2.1 - - [09/Dec/2016:22:28:19 +0000] "GET / HTTP/1.1" 304 0
"http://127.0.0.1:8001/api/v1/proxy/namespaces/kube-system/services/kubernetes-dashboard/" "Mozilla/5.0 (Windows NT
10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/54.0.2840.99 Safari/537.36" "-"
2016-12-09T22:28:19.974854146Z 10.244.2.1 - - [09/Dec/2016:22:28:19 +0000] "GET / HTTP/1.1" 304 0
"http://127.0.0.1:8001/api/v1/proxy/namespaces/kube-system/services/kubernetes-dashboard/" "Mozilla/5.0 (Windows NT
10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/54.0.2840.99 Safari/537.36" "-"
2016-12-09T22:28:20.684697233Z 10.244.2.1 - - [09/Dec/2016:22:28:20 +0000] "GET / HTTP/1.1" 304 0
"http://127.0.0.1:8001/api/v1/proxy/namespaces/kube-system/services/kubernetes-dashboard/" "Mozilla/5.0 (Windows NT
10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/54.0.2840.99 Safari/537.36" "-"
```

Viewing your service

In addition to running your containers, the Kubernetes UI has created an external `Service` which provisions a load balancer to bring traffic to the containers in your cluster.

In the left navigation pane, click **Services** to view all services (there should be only one).

Services				
Name	Labels	Cluster IP	Internal endpoints	External endpoints
hello-nginx	app: hello-nginx	10.0.74.250	hello-nginx:80 TCP hello-nginx:31000 TCP	13.68.245.175:80

In that view, you should see an external endpoint (IP address) that has been allocated to your service. If you click that IP address, you should see your Nginx container running behind the load balancer.



Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

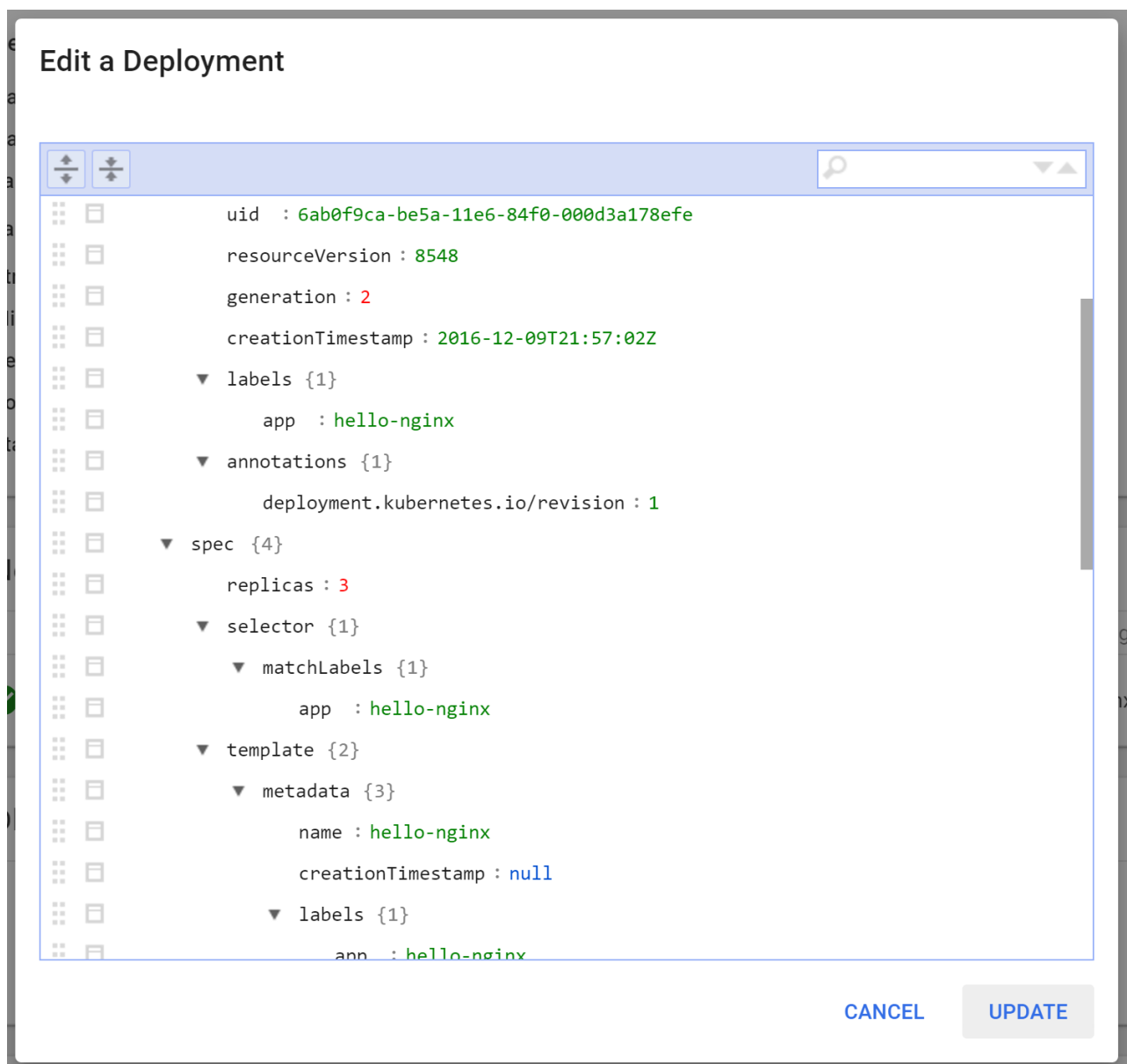
Thank you for using nginx.

Resizing your service

In addition to viewing your objects in the UI, you can edit and update the Kubernetes API objects.

First, click **Deployments** in the left navigation pane to see the deployment for your service.

Once you are in that view, click on the replica set, and then click **Edit** in the upper navigation bar:



Edit the `spec.replicas` field to be `2`, and click **Update**.

This causes the number of replicas to drop to two by deleting one of your pods.

Use Helm to deploy containers on a Kubernetes cluster

8/11/2017 • 2 min 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-get and Yum, Helm is used to manage Kubernetes charts, which are packages of preconfigured Kubernetes resources. This article shows you how to work with Helm on a Kubernetes cluster deployed in Azure Container Service.

Helm has two components:

- The **Helm CLI** is a client that runs on your machine locally or in the cloud
- **Tiller** is a server that runs on the Kubernetes cluster and manages the lifecycle of your Kubernetes applications

Prerequisites

- [Create a Kubernetes cluster](#) in Azure Container Service
- [Install and configure](#) `kubectl` on a local computer
- [Install Helm](#) on a local computer

Helm basics

To view information about the Kubernetes cluster that you are installing Tiller and deploying your applications to, type the following command:

```
kubectl cluster-info
```

```
(env) sauryas-MacBook-Pro:~ sauryadas$ kubectl cluster-info
Kubernetes master is running at https://helmkubcs-helmkubrg-5abfd9.westus.cloudapp.azure.com
heapster is running at https://helmkubcs-helmkubrg-5abfd9.westus.cloudapp.azure.com/api/v1/proxy/namespaces/kube-system/services/heapster
kubernetes-dashboard is running at https://helmkubcs-helmkubrg-5abfd9.westus.cloudapp.azure.com/api/v1/proxy/namespaces/kube-system/services/kubernetes-dashbo
rd
```

After you have installed Helm, install Tiller on your Kubernetes cluster by typing the following command:

```
helm init --upgrade
```

When it completes successfully, you see output like the following:

```
(env) Sauryas-MacBook-Pro:~ sauryadas$ helm init
$HELM_HOME has been configured at /Users/sauryadas/.helm.

Tiller (the helm server side component) has been installed into your Kubernetes Cluster.
Happy Helming!
```

To view all the Helm charts available in the repository, type the following command:

```
helm search
```

You see output like the following:

```
(env) Sauryas-MacBook-Pro:~ sauryadas$ helm search
NAME                VERSION DESCRIPTION
stable/aws-cluster-autoscaler 0.2.1 Scales worker nodes within autoscaling groups.
stable/chaoskub     0.4.0 Chaoskub periodically kills random pods in you...
stable/chronograf   0.1.2 Open-source web application written in Go and R...
stable/cockroachdb  0.2.2 CockroachDB is a scalable, survivable, strongly...
stable/concourse     0.1.3 Concourse is a simple and scalable CI system.
stable/datadog       0.2.1 DataDog Agent
stable/dokuwiki      0.1.3 DokuWiki is a standards-compliant, simple to us...
stable/drupal        0.4.4 One of the most versatile open source content m...
stable/etcd-operator 0.2.0 CoreOS etcd-operator Helm chart for Kubernetes
stable/factorio      0.1.4 Factorio dedicated server.
stable/gcloud-endpoints 0.1.0 Develop, deploy, protect and monitor your APIs ...
stable/ghost         0.4.6 A simple, powerful publishing platform that all...
```

To update the charts to get the latest versions, type:

```
helm repo update
```

Deploy an Nginx ingress controller chart

To deploy an Nginx ingress controller chart, type a single command:

```
helm install stable/nginx-ingress
```

```
(env) Sauryas-MacBook-Pro:~ sauryadas$ helm install stable/nginx-ingress
NAME: harping-mongoose
LAST DEPLOYED: Wed Apr 5 14:49:22 2017
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/Service
NAME                                CLUSTER-IP    EXTERNAL-IP    PORT(S)          AGE
harping-mongoose-nginx-ingress-default-backend 10.0.148.158  <none>         80/TCP           1s
harping-mongoose-nginx-ingress-controller      10.0.58.218   <pending>      80:31121/TCP,443:30285/TCP 1s

==> extensions/v1beta1/Deployment
NAME                                DESIRED    CURRENT    UP-TO-DATE    AVAILABLE    AGE
harping-mongoose-nginx-ingress-default-backend 1          1          1              0           1s
harping-mongoose-nginx-ingress-controller      1          1          1              0           1s

==> v1/ConfigMap
NAME                                DATA    AGE
harping-mongoose-nginx-ingress-controller 1        1s

NOTES:
The nginx-ingress controller has been installed.
It may take a few minutes for the LoadBalancer IP to be available.
You can watch the status by running 'kubectl --namespace default get services -o wide -w harping-mongoose-nginx-ingress-controller'

An example Ingress that makes use of the controller:
```

If you type `kubectl get svc` to view all services that are running on the cluster, you see that an IP address is assigned to the ingress controller. (While the assignment is in progress, you see `<pending>`. It takes a couple of minutes to complete.)

After the IP address is assigned, navigate to the value of the external IP address to see the Nginx backend running.

```
(env) Sauryas-MacBook-Pro:~ sauryadas$ kubectl get svc
NAME                                CLUSTER-IP    EXTERNAL-IP    PORT(S)          AGE
harping-mongoose-nginx-ingress-controller 10.0.58.218   40.86.189.5    80/TCP,443/TCP   4m
harping-mongoose-nginx-ingress-default-backend 10.0.148.158  <none>         80/TCP           4m
kubernetes                            10.0.0.1      <none>         443/TCP          47m
```

To see a list of charts installed on your cluster, type:

```
helm list
```

You can abbreviate the command to `helm ls`.

Deploy a MariaDB chart and client

Now deploy a MariaDB chart and a MariaDB client to connect to the database.

To deploy the MariaDB chart, type the following command:

```
helm install --name v1 stable/mariadb
```

where `--name` is a tag used for releases.

TIP

If the deployment fails, run `helm repo update` and try again.

To view all the charts deployed on your cluster, type:

```
helm list
```

To view all deployments running on your cluster, type:

```
kubectl get deployments
```

Finally, to run a pod to access the client, type:

```
kubectl run v1-mariadb-client --rm --tty -i --image bitnami/mariadb --command -- bash
```

To connect to the client, type the following command, replacing `v1-mariadb` with the name of your deployment:

```
sudo mysql -h v1-mariadb
```

You can now use standard SQL commands to create databases, tables, etc. For example, `Create DATABASE testdb1;` creates an empty database.

Next steps

- For more information about managing Kubernetes charts, see the [Helm documentation](#).

Jenkins integration with Azure Container Service and Kubernetes

8/11/2017 • 5 min to read • [Edit Online](#)

In this tutorial, we walk through the process to set up continuous integration of a multi-container application into Azure Container Service Kubernetes using the Jenkins platform. The workflow updates the container image in Docker Hub and upgrades the Kubernetes pods using a deployment rollout.

High level process

The basic steps detailed in this article are:

- Install a Kubernetes cluster in Container Service
- Set up Jenkins and configure access to Container Service
- Create a Jenkins workflow
- Test the CI/CD process end to end

Install a Kubernetes cluster

Deploy the Kubernetes cluster in Azure Container Service using the following steps. Full documentation is located [here](#).

Step 1: Create a resource group

```
RESOURCE_GROUP=my-resource-group
LOCATION=westus

az group create --name=$RESOURCE_GROUP --location=$LOCATION
```

Step 2: Deploy the cluster

NOTE

The following steps require a local SSH public key stored in the ~/.ssh folder.

```
RESOURCE_GROUP=my-resource-group
DNS_PREFIX=some-unique-value
CLUSTER_NAME=any-acs-cluster-name

az acs create \
  --orchestrator-type=kubernetes \
  --resource-group $RESOURCE_GROUP \
  --name=$CLUSTER_NAME \
  --dns-prefix=$DNS_PREFIX \
  --ssh-key-value ~/.ssh/id_rsa.pub \
  --admin-username=azureuser \
  --master-count=1 \
  --agent-count=5 \
  --agent-vm-size=Standard_D1_v2
```

Set up Jenkins and configure access to Container Service

Step 1: Install Jenkins

1. Create an Azure VM with Ubuntu 16.04 LTS. Since later in the steps you will need to connect to this VM using bash on your local machine, set the 'Authentication type' to 'SSH public key' and paste the SSH public key that is stored locally in your ~/.ssh folder. Also, take note of the 'User name' that you specify since this user name will be needed to view the Jenkins dashboard and for connecting to the Jenkins VM in later steps.
2. Install Jenkins via these [instructions](#). A more detailed tutorial is at howtoforge.com.
3. To view the Jenkins dashboard on your local machine, update the Azure network security group to allow port 8080 by adding an inbound rule that allows access to port 8080. Alternatively, you may setup port forwarding by running this command:

```
ssh -i ~/.ssh/id_rsa -L 8080:localhost:8080 <your_jenkins_user>@<your_jenkins_public_ip>
```

4. Connect to your Jenkins server using the browser by navigating to the public IP (http://<public_ip>:8080) and unlock the Jenkins dashboard for the first time with the initial admin password. The admin password is stored at /var/lib/jenkins/secrets/initialAdminPassword on the Jenkins VM. An easy way to get this password is to SSH into the Jenkins VM: `ssh <your_jenkins_user>@<your_jenkins_public_ip>`. Next, run:

```
sudo cat /var/lib/jenkins/secrets/initialAdminPassword
```

5. Install Docker on the Jenkins machine via these [instructions](#). This allows for Docker commands to be run in Jenkins jobs.
6. Configure Docker permissions to allow Jenkins to access the Docker endpoint.

```
sudo chmod 777 /run/docker.sock
```

7. Install `kubectl` CLI on Jenkins. More details are at [Installing and Setting up kubectl](#). Jenkins jobs will use 'kubectl' to manage and deploy to the Kubernetes cluster.

```
curl -LO https://storage.googleapis.com/kubernetes-release/release/$(curl -s
https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/amd64/kubectl

chmod +x ./kubectl

sudo mv ./kubectl /usr/local/bin/kubectl
```

Step 2: Set up access to the Kubernetes cluster

NOTE

There are multiple approaches to accomplishing the following steps. Use the approach that is easiest for you.

1. Copy the `kubectl` config file to the Jenkins machine so that Jenkins jobs have access to the Kubernetes cluster. These instructions assume that you are using bash from a different machine than the Jenkins VM and that a local SSH public key is stored in the machine's ~/.ssh folder.

```
export KUBE_MASTER=<your_cluster_master_fqdn>
export JENKINS_USER=<your_jenkins_user>
export JENKINS_SERVER=<your_jenkins_public_ip>
sudo ssh $JENKINS_USER@$JENKINS_SERVER sudo mkdir -m 777 /home/$JENKINS_USER/.kube/ \
&& sudo ssh $JENKINS_USER@$JENKINS_SERVER sudo mkdir /var/lib/jenkins/.kube/ \
&& sudo scp -3 -i ~/.ssh/id_rsa azureuser@$KUBE_MASTER:~/.kube/config \
$JENKINS_USER@$JENKINS_SERVER:~/.kube/config \
&& sudo ssh -i ~/.ssh/id_rsa $JENKINS_USER@$JENKINS_SERVER sudo cp /home/$JENKINS_USER/.kube/config \
/var/lib/jenkins/.kube/config \
```

1. Validate from Jenkins that the Kubernetes cluster is accessible. To do this, SSH into the Jenkins VM:

```
ssh <your_jenkins_user>@<your_jenkins_public_ip> . Next, verify Jenkins can successfully connect to your cluster:  
kubectl cluster-info .
```

Create a Jenkins workflow

Prerequisites

- GitHub account for code repo.
- Docker Hub account to store and update images.
- Containerized application that can be rebuilt and updated. You can use this sample container app written in Golang: <https://github.com/chzbrgr71/go-web>

NOTE

The following steps must be performed in your own GitHub account. Feel free to clone the above repo, but you must use your own account to configure the webhooks and Jenkins access.

Step 1: Deploy initial v1 of application

1. Build the app from the developer machine with the following commands. Replace `myrepo` with your own.

```
git clone https://github.com/chzbrgr71/go-web.git  
cd go-web  
docker build -t myrepo/go-web .
```

2. Push image to Docker Hub.

```
docker login  
docker push myrepo/go-web
```

3. Deploy to the Kubernetes cluster.

NOTE

Edit the `go-web.yaml` file to update your container image and repo.

```
kubectl create -f ./go-web.yaml --record
```

Step 2: Configure Jenkins system

4. Click **Manage Jenkins > Configure System**.
5. Under **GitHub**, select **Add GitHub Server**.
6. Leave **API URL** as default.
7. Under **Credentials**, add a Jenkins credential using **Secret text**. We recommend using GitHub personal access tokens, which are configured in your GitHub user account settings. More details on this [here](#).
8. Click **Test connection** to ensure this is configured correctly.
9. Under **Global Properties**, add an environment variable `DOCKER_HUB` and provide your Docker Hub password. (This is useful in this demo, but a production scenario would require a more secure approach.)
10. Save.

GitHub

GitHub Servers

GitHub Server

API URL

Credentials

Secret text ▾

Add ▾

Test connection

Manage hooks ☒

Step 3: Create the Jenkins workflow

1. Create a Jenkins item.
2. Provide a name (for example, "go-web") and select **Freestyle Project**.
3. Check **GitHub project** and provide the URL to your GitHub repo.
4. In **Source Code Management**, provide the GitHub repo URL and credentials.
5. Add a **Build Step** of type **Execute shell** and use the following text:

```
WEB_IMAGE_NAME="myrepo/go-web:kube${BUILD_NUMBER}"
docker build -t $WEB_IMAGE_NAME .
docker login -u <your-dockerhub-username> -p ${DOCKER_HUB}
docker push $WEB_IMAGE_NAME
```

6. Add another **Build Step** of type **Execute shell** and use the following text:

```
WEB_IMAGE_NAME="myrepo/go-web:kube${BUILD_NUMBER}"
kubectl set image deployment/go-web go-web=$WEB_IMAGE_NAME --kubeconfig /var/lib/jenkins/config
```

Build

⋮

Execute shell

X

?

Command

```
WEB_IMAGE_NAME="myrepo/go-web:kube${BUILD_NUMBER}"
docker build -t $WEB_IMAGE_NAME .
docker login -u <your-dockerhub-username> -p ${DOCKER_HUB}
docker push $WEB_IMAGE_NAME
```

See [the list of available environment variables](#)

Advanced...

⋮

Execute shell

X

?

Command

```
WEB_IMAGE_NAME="myrepo/go-web:kube${BUILD_NUMBER}"
kubectl set image deployment/go-web go-web=$WEB_IMAGE_NAME --kubeconfig /var/lib/jenkins/config
```

See [the list of available environment variables](#)

1. Save the Jenkins item and test with **Build Now**.

Step 4: Connect GitHub webhook

1. In the Jenkins item you created, click **Configure**.
2. Under **Build Triggers**, select **GitHub hook trigger for GITScm polling** and **Save**. This automatically configures the GitHub webhook.
3. In your GitHub repo for go-web, click **Settings > Webhooks**.
4. Verify that the Jenkins webhook URL was added successfully. The URL should end in "github-webhook".

Build Triggers

☐ Trigger builds remotely (e.g., from scripts)
☐ Build after other projects are built
☐ Build periodically
☒ GitHub hook trigger for GITScm polling
☐ Poll SCM

Test the CI/CD process end to end

1. Update code for the repo and push/synch with the GitHub repository.
2. From the Jenkins console, check the **Build History** and validate that the job has run. View console output to see details.
3. From Kubernetes, view details of the upgraded deployment:

```
kubectl rollout history deployment/go-web
```

Next steps

- Deploy Azure Container Registry and store images in a secure repository. See [Azure Container Registry docs](#).
- Build a more complex workflow that includes side-by-side deployment and automated tests in Jenkins.
- For more information about CI/CD with Jenkins and Kubernetes, see the [Jenkins blog](#).

Use Draft with Azure Container Service and Azure Container Registry to build and deploy an application to Kubernetes

9/25/2017 • 7 min to read • [Edit Online](#)

[Draft](#) is a new open-source tool that makes it easy to develop container-based applications and deploy them to Kubernetes clusters without knowing much about Docker and Kubernetes -- or even installing them. Using tools like Draft let you and your teams focus on building the application with Kubernetes, not paying as much attention to infrastructure.

You can use Draft with any Docker image registry and any Kubernetes cluster, including locally. This tutorial shows how to use ACS with Kubernetes and ACR to create a live but secure developer pipeline in Kubernetes using Draft, and how to use Azure DNS to expose that developer pipeline for others to see at a domain.

Create an Azure Container Registry

You can easily [create a new Azure Container Registry](#), but the steps are as follows:

1. Create a Azure resource group to manage your ACR registry and the Kubernetes cluster in ACS.

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

2. Create an ACR image registry using [az acr create](#) and ensure that the `--admin-enabled` option is set to `true`.

```
az acr create --resource-group draft --name draftacs --sku Basic --admin-enabled true
```

Create an Azure Container Service with Kubernetes

Now you're ready to use [az acs create](#) to create an ACS cluster using Kubernetes as the `--orchestrator-type` value.

```
az acs create --resource-group draft --name draft-kube-acsc --dns-prefix draft-cluster --orchestrator-type  
kubernetes --generate-ssh-keys
```

NOTE

Because Kubernetes is not the default orchestrator type, be sure you use the `--orchestrator-type kubernetes` switch.

The output when successful looks similar to the following.

```

waiting for AAD role to propagate.done
{
  "id":
"/subscriptions/<guid>/resourceGroups/draft/providers/Microsoft.Resources/deployments/azurecli14904.93snip09",
  "name": "azurecli1496227204.9323909",
  "properties": {
    "correlationId": "<guid>",
    "debugSetting": null,
    "dependencies": [],
    "mode": "Incremental",
    "outputs": null,
    "parameters": {
      "clientSecret": {
        "type": "SecureString"
      }
    },
    "parametersLink": null,
    "providers": [
      {
        "id": null,
        "namespace": "Microsoft.ContainerService",
        "registrationState": null,
        "resourceTypes": [
          {
            "aliases": null,
            "apiVersions": null,
            "locations": [
              "westus"
            ],
            "properties": null,
            "resourceType": "containerServices"
          }
        ]
      }
    ],
    "provisioningState": "Succeeded",
    "template": null,
    "templateLink": null,
    "timestamp": "2017-05-31T10:46:29.434095+00:00"
  },
  "resourceGroup": "draft"
}

```

Now that you have a cluster, you can import the credentials by using the [az acs kubernetes get-credentials](#) command. Now you have a local configuration file for your cluster, which is what Helm and Draft need to get their work done.

Install and configure draft

1. Download draft for your environment at <https://github.com/Azure/draft/releases> and install into your PATH so that the command can be used.
2. Download helm for your environment at <https://github.com/kubernetes/helm/releases> and [install it into your PATH so that the command can be used](#).
3. Configure Draft to use your registry and create subdomains for each Helm chart it creates. To configure Draft, you need:

- your Azure Container Registry name (in this example, `draftacsdemo`)
- your registry key, or password, from

```
az acr credential show -n <registry name> --output tsv --query "passwords[0].value"
```

Call `draft init` and the configuration process prompts you for the values above; note that the URL format for the registry URL is the registry name (in this example, `draftacsdemo`) plus `.azurecr.io`. Your username

is the registry name on its own. The process looks something like the following the first time you run it.

```
$ draft init
Creating /home/ralph/.draft
Creating /home/ralph/.draft/plugins
Creating /home/ralph/.draft/packs
Creating pack go...
Creating pack python...
Creating pack ruby...
Creating pack javascript...
Creating pack gradle...
Creating pack java...
Creating pack php...
Creating pack csharp...
$DRAFT_HOME has been configured at /home/ralph/.draft.

In order to configure Draft, we need a bit more information...

1. Enter your Docker registry URL (e.g. docker.io/myuser, quay.io/myuser, myregistry.azurecr.io):
draftacsdemo.azurecr.io
2. Enter your username: draftacsdemo
3. Enter your password:
Draft has been installed into your Kubernetes Cluster.
Happy Sailing!
```

Now you're ready to deploy an application.

Build and deploy an application

In the Draft repo are [six simple example applications](#). Clone the repo and let's use the [Java example](#). Change into the examples/java directory, and type `draft create` to build the application. It should look like the following example.

```
$ draft create
--> Draft detected the primary language as Java with 91.228814% certainty.
--> Ready to sail
```

The output includes a Dockerfile and a Helm chart. To build and deploy, you just type `draft up`. The output is extensive, but should be like the following example.

```
$ draft up
Draft Up Started: 'handy-labradoodle'
handy-labradoodle: Building Docker Image: SUCCESS ☑ (35.0232s)
handy-labradoodle: Pushing Docker Image: SUCCESS ☑ (17.0062s)
handy-labradoodle: Releasing Application: SUCCESS ☑ (3.8903s)
handy-labradoodle: Build ID: 01BT0ZJ87NWCD7BBPK4Y3BTTPB
```

Securely view your application

Your container is now running in ACS. To view it, use the `draft connect` command, which creates a secured connection to the cluster's IP with a specific port for your application so that you can view it locally. If successful, look for the URL to connect to your app on the first line after the **SUCCESS** indicator.

NOTE

If you receive a message saying that no pods were ready, wait for a moment and retry, or you can watch the pods become ready with `kubectl get pods -w` and then retry when they do.

```
draft connect
Connecting to your app...SUCCESS...Connect to your app on localhost:46143
Starting log streaming...
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
== Spark has ignited ...
>> Listening on 0.0.0.0:4567
```

In the preceding example, you could type `curl -s http://localhost:46143` to receive the reply, `Hello World, I'm Java!`. When you CTRL+ or CMD+C (depending on your OS environment), the secure tunnel is torn down and you can continue iterating.

Sharing your application by configuring a deployment domain with Azure DNS

You have already performed the developer iteration loop that Draft creates in the preceding steps. However, you can share your application across the internet by:

1. Installing an ingress in your ACS cluster (to provide a public IP address at which to display the app)
2. Delegating your custom domain to Azure DNS and mapping your domain to the IP address ACS assigns to your ingress controller

Use helm to install the ingress controller.

Use **helm** to search for and install `stable/traefik`, an ingress controller, to enable inbound requests for your builds.

```
$ helm search traefik
NAME          VERSION DESCRIPTION
stable/traefik 1.3.0    A Traefik based Kubernetes ingress controller w...

$ helm install stable/traefik --name ingress
```

Now set a watch on the `ingress` controller to capture the external IP value when it is deployed. This IP address will be the one [mapped to your deployment domain](#) in the next section.

```
kubectl get svc -w
```

NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
ingress-traefik	10.0.248.104	13.64.108.240	80:31046/TCP,443:32556/TCP	1h
kubernetes	10.0.0.1	<none>	443/TCP	7h

In this case, the external IP for the deployment domain is `13.64.108.240`. Now you can map your domain to that IP.

Map the ingress IP to a custom subdomain

Draft creates a release for each Helm chart it creates -- each application you are working on. Each one gets a generated name that is used by **draft** as a *subdomain* on top of the root *deployment domain* that you control. (In this example, we use `squillace.io` as the deployment domain.) To enable this subdomain behavior, you must create an A record for `*.draft` in your DNS entries for your deployment domain, so that each generated subdomain is routed to the Kubernetes cluster's ingress controller.

Your own domain provider has their own way to assign DNS servers; to [delegate your domain nameservers to Azure DNS](#), you take the following steps:

1. Create a resource group for your zone.

```
az group create --name squillace.io --location eastus
{
  "id": "/subscriptions/<guid>/resourceGroups/squillace.io",
  "location": "eastus",
  "managedBy": null,
  "name": "zones",
  "properties": {
    "provisioningState": "Succeeded"
  },
  "tags": null
}
```

2. Create a DNS zone for your domain. Use the [az network dns zone create](#) command to obtain the nameservers to delegate DNS control to Azure DNS for your domain.

```
az network dns zone create --resource-group squillace.io --name squillace.io
{
  "etag": "<guid>",
  "id": "/subscriptions/<guid>/resourceGroups/zones/providers/Microsoft.Network/dnszones/squillace.io",
  "location": "global",
  "maxNumberOfRecordSets": 5000,
  "name": "squillace.io",
  "nameServers": [
    "ns1-09.azure-dns.com.",
    "ns2-09.azure-dns.net.",
    "ns3-09.azure-dns.org.",
    "ns4-09.azure-dns.info."
  ],
  "numberOfRecordSets": 2,
  "resourceGroup": "squillace.io",
  "tags": {},
  "type": "Microsoft.Network/dnszones"
}
```

3. Add the DNS servers you are given to the domain provider for your deployment domain, which enables you to use Azure DNS to repoint your domain as you want. The way you do this varies by domain provide; [delegate your domain nameservers to Azure DNS](#) contains some of the details that you should know.
4. Once your domain has been delegated to Azure DNS, create an A record-set entry for your deployment domain mapping to the `ingress` IP from step 2 of the previous section.

```
azurecli az network dns record-set a add-record --ipv4-address 13.64.108.240 --record-set-name '*.draft' -g squillace.io -z squillace.io
```

The output looks something like:

```
json { "arecords": [ { "ipv4Address": "13.64.108.240" } ], "etag": "<guid>", "id":
"/subscriptions/<guid>/resourceGroups/squillace.io/providers/Microsoft.Network/dnszones/squillace.io/A/*",
"metadata": null, "name": "*.draft", "resourceGroup": "squillace.io", "ttl": 3600, "type":
"Microsoft.Network/dnszones/A" }
```

5. Reinstall **draft**

- a. Remove **draftd** from the cluster by typing `helm delete --purge draft`.
- b. Reinstall **draft** by using the same `draft-init` command, but with the `--ingress-enabled` option:

```
bash draft init --ingress-enabled
```

Respond to the prompts as you did the first time, above. However, you have one more question to respond to, using the complete domain path that you configured with the Azure DNS. ``bash

6. Enter your top-level domain for ingress (e.g. draft.example.com): draft.squillace.io ``

7. When you call `draft up` this time, you will be able to see your application (or `curl` it) at the URL of the form

```
<appname>.draft.<domain>.<top-level-domain>
```

In the case of this example,

```
http://handy-labradoodle.draft.squillace.io
```

```
bash curl -s http://handy-labradoodle.draft.squillace.io Hello World, I'm Java!
```

Next steps

Now that you have an ACS Kubernetes cluster, you can investigate using [Azure Container Registry](#) to create more and different deployments of this scenario. For example, you can create a draft *basedomain.toplevel* domain DNS record-set that controls things off of a deeper subdomain for specific ACS deployments.

Make a remote connection to a Kubernetes, DC/OS, or Docker Swarm cluster

8/11/2017 • 6 min to read • [Edit Online](#)

After creating an Azure Container Service cluster, you need to connect to the cluster to deploy and manage workloads. This article describes how to connect to the master VM of the cluster from a remote computer.

The Kubernetes, DC/OS, and Docker Swarm clusters provide HTTP endpoints locally. For Kubernetes, this endpoint is securely exposed on the internet, and you can access it by running the `kubect1` command-line tool from any internet-connected machine.

For DC/OS and Docker Swarm, we recommend that you create a secure shell (SSH) tunnel from your local computer to the cluster management system. After the tunnel is established, you can run commands which use the HTTP endpoints and view the orchestrator's web interface (if available) from your local system.

Prerequisites

- A Kubernetes, DC/OS, or Docker Swarm cluster [deployed in Azure Container Service](#).
- SSH RSA private key file, corresponding to the public key added to the cluster during deployment. These commands assume that the private SSH key is in `$HOME/.ssh/id_rsa` on your computer. See these instructions for [macOS and Linux](#) or [Windows](#) for more information. If the SSH connection isn't working, you may need to [reset your SSH keys](#).

Connect to a Kubernetes cluster

Follow these steps to install and configure `kubect1` on your computer.

NOTE

On Linux or macOS, you might need to run the commands in this section using `sudo`.

Install kubect1

One way to install this tool is to use the `az acs kubernetes install-cli` Azure CLI 2.0 command. To run this command, make sure that you [installed](#) the latest Azure CLI 2.0 and logged in to an Azure account (`az login`).

```
# Linux or macOS
az acs kubernetes install-cli [--install-location=/some/directory/kubect1]

# Windows
az acs kubernetes install-cli [--install-location=C:\some\directory\kubect1.exe]
```

Alternatively, you can download the latest `kubect1` client directly from the [Kubernetes releases page](#). For more information, see [Installing and Setting up kubect1](#).

Download cluster credentials

Once you have `kubect1` installed, you need to copy the cluster credentials to your machine. One way to do get the credentials is with the `az acs kubernetes get-credentials` command. Pass the name of the resource group and the name of the container service resource:


```
az acs kubernetes get-credentials --resource-group=<cluster-resource-group> --name=<cluster-name>
```

This command downloads the cluster credentials to `$HOME/.kube/config`, where `kubectl` expects it to be located.

Alternatively, you can use `scp` to securely copy the file from `$HOME/.kube/config` on the master VM to your local machine. For example:

```
mkdir $HOME/.kube
scp azureuser@<master-dns-name>:.kube/config $HOME/.kube/config
```

If you are on Windows, you can use Bash on Ubuntu on Windows, the PuTTY secure file copy client, or a similar tool.

Use kubectl

Once you have `kubectl` configured, test the connection by listing the nodes in your cluster:

```
kubectl get nodes
```

You can try other `kubectl` commands. For example, you can view the Kubernetes Dashboard. First, run a proxy to the Kubernetes API server:

```
kubectl proxy
```

The Kubernetes UI is now available at: `http://localhost:8001/ui`.

For more information, see the [Kubernetes quick start](#).

Connect to a DC/OS or Swarm cluster

To use the DC/OS and Docker Swarm clusters deployed by Azure Container Service, follow these instructions to create a SSH tunnel from your local Linux, macOS, or Windows system.

NOTE

These instructions focus on tunneling TCP traffic over SSH. You can also start an interactive SSH session with one of the internal cluster management systems, but we don't recommend this. Working directly on an internal system risks inadvertent configuration changes.

Create an SSH tunnel on Linux or macOS

The first thing that you do when you create an SSH tunnel on Linux or macOS is to locate the public DNS name of the load-balanced masters. Follow these steps:

1. In the [Azure portal](#), browse to the resource group containing your container service cluster. Expand the resource group so that each resource is displayed.
2. Click the **Container service** resource, and click **Overview**. The **Master FQDN** of the cluster appears under **Essentials**. Save this name for later use.

Delete	
Essentials ^	
Resource group (change) danlep0126	Master FQDN danlep0126mgmt.westus.cloudapp.azure.com
Location West US	
Subscription name (change) Microsoft Azure - CloudApp	
Subscription ID 00000000-0000-0000-0000-000000000000	

Alternatively, run the `az acs show` command on your container service. Look for the **Master Profile:fqdn** property in the command output.

- Now open a shell and run the `ssh` command by specifying the following values:

LOCAL_PORT is the TCP port on the service side of the tunnel to connect to. For Swarm, set this to 2375. For DC/OS, set this to 80. **REMOTE_PORT** is the port of the endpoint that you want to expose. For Swarm, use port 2375. For DC/OS, use port 80.

USERNAME is the user name that was provided when you deployed the cluster.

DNSPREFIX is the DNS prefix that you provided when you deployed the cluster.

REGION is the region in which your resource group is located.

PATH_TO_PRIVATE_KEY [OPTIONAL] is the path to the private key that corresponds to the public key you provided when you created the cluster. Use this option with the `-i` flag.

```
ssh -fNL LOCAL_PORT:localhost:REMOTE_PORT -p 2200 [USERNAME]@[DNSPREFIX]mgmt.[REGION].cloudapp.azure.com
```

NOTE

The SSH connection port is 2200 and not the standard port 22. In a cluster with more than one master VM, this is the connection port to the first master VM.

The command returns without output.

See the examples for DC/OS and Swarm in the following sections.

DC/OS tunnel

To open a tunnel for DC/OS endpoints, run a command like the following:

```
sudo ssh -fNL 80:localhost:80 -p 2200 azureuser@acsexamplemgmt.japaneast.cloudapp.azure.com
```

NOTE

Ensure that you do not have another local process that binds port 80. If necessary, you can specify a local port other than port 80, such as port 8080. However, some web UI links might not work when you use this port.

You can now access the DC/OS endpoints from your local system through the following URLs (assuming local port 80):

- DC/OS: `http://localhost:80/`
- Marathon: `http://localhost:80/marathon`
- Mesos: `http://localhost:80/mesos`

Similarly, you can reach the rest APIs for each application through this tunnel.

Swarm tunnel

To open a tunnel to the Swarm endpoint, run a command like the following:

```
ssh -fNL 2375:localhost:2375 -p 2200 azureuser@acsexamplemgmt.japaneast.cloudapp.azure.com
```

NOTE

Ensure that you do not have another local process that binds port 2375. For example, if you are running the Docker daemon locally, it's set by default to use port 2375. If necessary, you can specify a local port other than port 2375.

Now you can access the Docker Swarm cluster using the Docker command-line interface (Docker CLI) on your local system. For installation instructions, see [Install Docker](#).

Set your DOCKER_HOST environment variable to the local port you configured for the tunnel.

```
export DOCKER_HOST=:2375
```

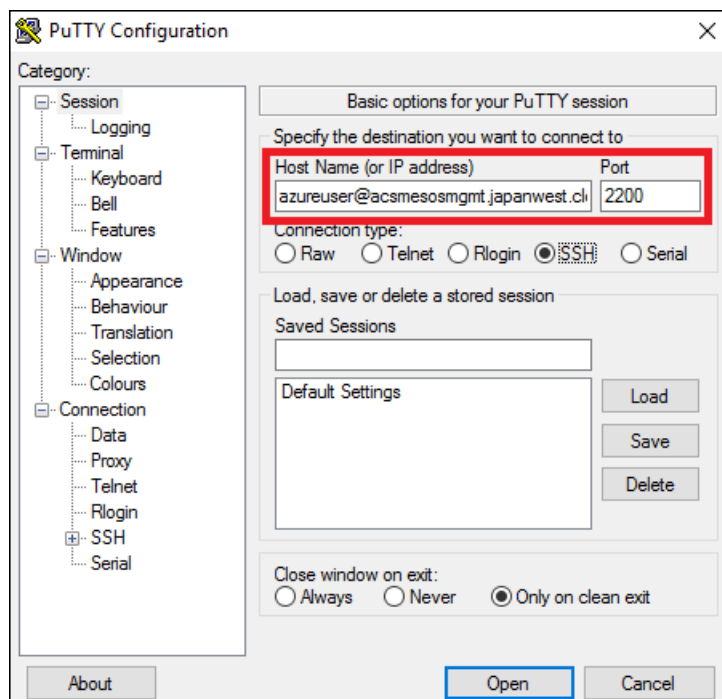
Run Docker commands that tunnel to the Docker Swarm cluster. For example:

```
docker info
```

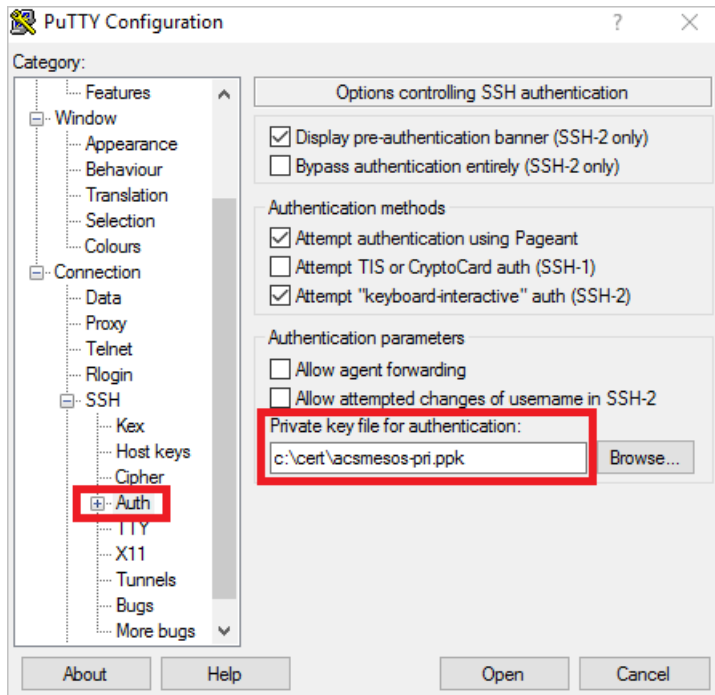
Create an SSH tunnel on Windows

There are multiple options for creating SSH tunnels on Windows. If you are running Bash on Ubuntu on Windows or a similar tool, you can follow the SSH tunneling instructions shown earlier in this article for macOS and Linux. As an alternative on Windows, this section describes how to use PuTTY to create the tunnel.

1. [Download PuTTY](#) to your Windows system.
2. Run the application.
3. Enter a host name that is comprised of the cluster admin user name and the public DNS name of the first master in the cluster. The **Host Name** looks similar to `azureuser@PublicDNSName`. Enter 2200 for the **Port**.



4. Select **SSH > Auth**. Add a path to your private key file (.ppk format) for authentication. You can use a tool such as [PuTTYgen](#) to generate this file from the SSH key used to create the cluster.



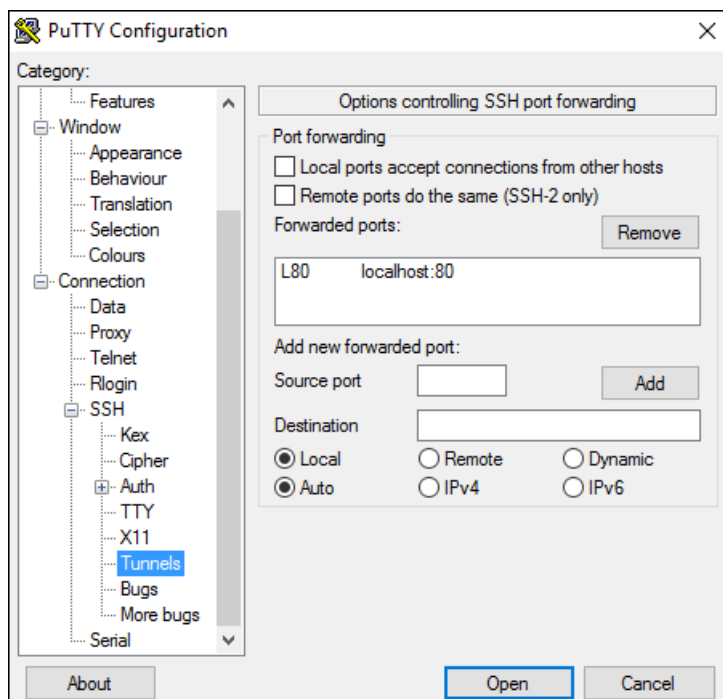
5. Select **SSH > Tunnels** and configure the following forwarded ports:

- **Source Port:** Use 80 for DC/OS or 2375 for Swarm.
- **Destination:** Use localhost:80 for DC/OS or localhost:2375 for Swarm.

The following example is configured for DC/OS, but will look similar for Docker Swarm.

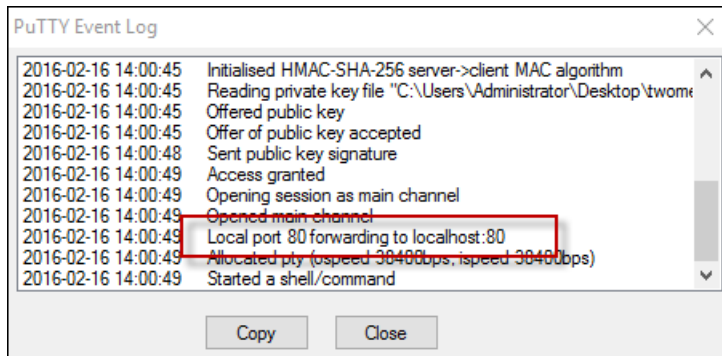
NOTE

Port 80 must not be in use when you create this tunnel.



6. When you're finished, click **Session > Save** to save the connection configuration.
7. To connect to the PuTTY session, click **Open**. When you connect, you can see the port configuration in the

PuTTY event log.



After you've configured the tunnel for DC/OS, you can access the related endpoints at:

- DC/OS: `http://localhost/`
- Marathon: `http://localhost/marathon`
- Mesos: `http://localhost/mesos`

After you've configured the tunnel for Docker Swarm, open your Windows settings to configure a system environment variable named `DOCKER_HOST` with a value of `:2375`. Then, you can access the Swarm cluster through the Docker CLI.

Next steps

Deploy and manage containers in your cluster:

- [Work with Azure Container Service and Kubernetes](#)
- [Work with Azure Container Service and DC/OS](#)
- [Work with the Azure Container Service and Docker Swarm](#)

Scale agent nodes in a Container Service cluster

8/11/2017 • 2 min to read • [Edit Online](#)

After [deploying an Azure Container Service cluster](#), you might need to change the number of agent nodes. For example, you might need more agents so you can run more container applications or instances.

You can change the number of agent nodes in a DC/OS, Docker Swarm, or Kubernetes cluster by using the Azure portal or the Azure CLI 2.0.

Scale with the Azure portal

1. In the [Azure portal](#), browse for **Container services**, and then click the container service that you want to modify.
2. In the **Container service** blade, click **Agents**.
3. In **VM Count**, enter the desired number of agents nodes.

containerservice-danlep0110dc - Agents

Agent FQDNs are used to connect to your deployed applications. The Master FQDN is used to connect to the cluster itself.

Application endpoints
Connecting to the cluster

Master FQDN: danlep0110dcmgmt.westus.cloudapp.azure.com

NAME	VM SIZE	VM COUNT	FQDN
agentpools	Standard_DS2	3	danlep0110dcagents.westus.cloudapp.azure.com

4. To save the configuration, click **Save**.

Scale with the Azure CLI 2.0

Make sure that you [installed](#) the latest Azure CLI 2.0 and logged in to an azure account (`az login`).

See the current agent count

To see the number of agents currently in the cluster, run the `az acs show` command. This shows the cluster configuration. For example, the following command shows the configuration of the container service named `containerservice-myACSName` in the resource group `myResourceGroup` :

```
az acs show -g myResourceGroup -n containerservice-myACSName
```

The command returns the number of agents in the `Count` value under `AgentPoolProfiles` .

Use the `az acs scale` command

To change the number of agent nodes, run the `az acs scale` command and supply the **resource group**, **container service name**, and the desired **new agent count**. By using a smaller or higher number, you can scale down or up, respectively.

For example, to change the number of agents in the previous cluster to 10, type the following command:

```
az acs scale -g myResourceGroup -n containerservice-myACSName --new-agent-count 10
```

The Azure CLI 2.0 returns a JSON string representing the new configuration of the container service, including the new agent count.

For more command options, run `az acs scale --help`.

Scaling considerations

- The number of agent nodes must be between 1 and 100, inclusive.
- Your cores quota can limit the number of agent nodes in a cluster.
- Agent node scaling operations are applied to an Azure virtual machine scale set that contains the agent pool. In a DC/OS cluster, only agent nodes in the private pool are scaled by the operations shown in this article.
- Depending on the orchestrator you deploy in your cluster, you can separately scale the number of instances of a container running on the cluster. For example, in a DC/OS cluster, use the [Marathon UI](#) to change the number of instances of a container application.
- Currently, autoscaling of agent nodes in a container service cluster is not supported.

Next steps

- See [more examples](#) of using Azure CLI 2.0 commands with Azure Container Service.
- Learn more about [DC/OS agent pools](#) in Azure Container Service.

Load balance containers in a Kubernetes cluster in Azure Container Service

8/11/2017 • 4 min to read • [Edit Online](#)

This article introduces load balancing in a Kubernetes cluster in Azure Container Service. Load balancing provides an externally accessible IP address for the service and distributes network traffic among the pods running in agent VMs.

You can set up a Kubernetes service to use [Azure Load Balancer](#) to manage external network (TCP) traffic. With additional configuration, load balancing and routing of HTTP or HTTPS traffic or more advanced scenarios are possible.

Prerequisites

- [Deploy a Kubernetes cluster](#) in Azure Container Service
- [Connect your client](#) to your cluster

Azure load balancer

By default, a Kubernetes cluster deployed in Azure Container Service includes an Internet-facing Azure load balancer for the agent VMs. (A separate load balancer resource is configured for the master VMs.) Azure load balancer is a Layer 4 load balancer. Currently, the load balancer only supports TCP traffic in Kubernetes.

When creating a Kubernetes service, you can automatically configure the Azure load balancer to allow access to the service. To configure the load balancer, set the service `type` to `LoadBalancer`. The load balancer creates a rule to map a public IP address and port number of incoming service traffic to the private IP addresses and port numbers of the pods in agent VMs (and vice versa for response traffic).

Following are two examples showing how to set the Kubernetes service `type` to `LoadBalancer`. (After trying the examples, delete the deployments if you no longer need them.)

Example: Use the `kubectl expose` command

The [Kubernetes walkthrough](#) includes an example of how to expose a service with the `kubectl expose` command and its `--type=LoadBalancer` flag. Here are the steps :

1. Start a new container deployment. For example, the following command starts a new deployment called `mynginx`. The deployment consists of three containers based on the Docker image for the Nginx web server.

```
kubectl run mynginx --replicas=3 --image nginx
```

2. Verify that the containers are running. For example, if you query for the containers with `kubectl get pods`, you see output similar to the following:

NAME	READY	STATUS	RESTARTS	AGE
mynginx-2380521294-ckdji	1/1	Running	0	27m
mynginx-2380521294-h4tmq	1/1	Running	0	27m
mynginx-2380521294-xjyy6	1/1	Running	0	27m

3. To configure the load balancer to accept external traffic to the deployment, run `kubectl expose` with `--type=LoadBalancer`. The following command exposes the Nginx server on port 80:


```
kubectl expose deployments mynginx --port=80 --type=LoadBalancer
```

4. Type `kubectl get svc` to see the state of the services in the cluster. While the load balancer configures the rule, the `EXTERNAL-IP` of the service appears as `<pending>`. After a few minutes, the external IP address is configured:

NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	10.0.0.1	<none>	443/TCP	18h
mynginx	10.0.161.47	13.82.93.130	80/TCP	17h

5. Verify that you can access the service at the external IP address. For example, open a web browser to the IP address shown. The browser shows the Nginx web server running in one of the containers. Or, run the `curl` or `wget` command. For example:

```
curl 13.82.93.130
```

You should see output similar to:

```
curl 13.82.93.130
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           Dload  Upload   Total   Spent    Left     Speed
100  612  100  612    0     0   3923      0  --:--:-- --:--:-- --:--:-- 3923<!
DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

6. To see the configuration of the Azure load balancer, go to the [Azure portal](#).
7. Browse for the resource group for your container service cluster, and select the load balancer for the agent VMs. Its name should be the same as the container service. (Don't choose the load balancer for the master nodes, the one whose name includes **master-lb**.)

+

 Add

≡

 Columns

🗑️

 Delete

🔄

 Refresh

➡️

 Move

Essentials

⌵

Subscription name (change)

Deployments















2 Succeeded

Subscription ID

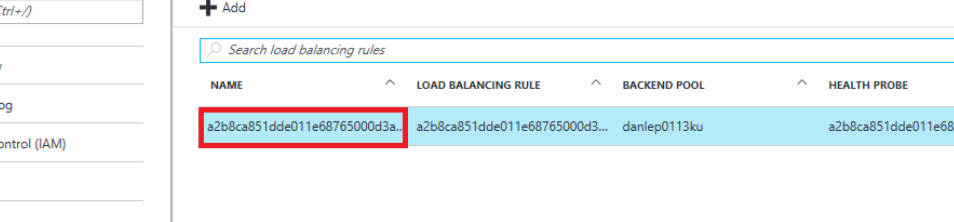
Location

East US

28 items

NAME	TYPE	LOCATION
 agent-availabilitySet-D814F076	Availability set	East US
 master-availabilityset	Availability set	East US
 danlep0113ku	Container service	East US
 danlep0113ku	Load balancer	East US
 k8s-master-lb-D814F076	Load balancer	East US
 k8s-agent-D814F076-nic-0	Network interface	East US
 k8s-agent-D814F076-nic-1	Network interface	East US
 k8s-agent-D814F076-nic-2	Network interface	East US
 k8s-master-D814F076-nic-0	Network interface	East US
 k8s-master-D814F076-nsg	Network security group	East US
 danlep0113ku-a2b8ca851dde011e68765000d3a1530e	Public IP address	East US
 k8s-master-ip-danlep0113ku-D814F076	Public IP address	East US
 k8s-master-D814F076-routetable	Route table	East US
 00b6m2qchwn2zoagnt0	Storage account	East US

- To see the details of the load balancer configuration, click **Load balancing rules** and the name of the rule that was configured.



The screenshot shows the AWS Management Console interface for a load balancer named 'danlep0113ku'. The left sidebar contains a navigation menu with options like Overview, Activity log, Access control (IAM), Tags, SETTINGS, Frontend IP pool, Backend pools, Health probes, Load balancing rules (highlighted with a red box), Inbound NAT rules, Properties, Locks, and Automation script. The main content area displays the 'Load balancing rules' page, which includes a search bar and a table of rules. The table has columns for NAME, LOAD BALANCING RULE, BACKEND POOL, and HEALTH PROBE. The first rule, 'a2b8ca851dde011e68765000d3a...', is highlighted with a red box.

NAME	LOAD BALANCING RULE	BACKEND POOL	HEALTH PROBE
a2b8ca851dde011e68765000d3a...	a2b8ca851dde011e68765000d3...	danlep0113ku	a2b8ca851dde011e68765000d3... ..

Example: Specify `type: LoadBalancer` **in the service configuration file**

If you deploy a Kubernetes container app from a YAML or JSON [service configuration file](#), specify an external load balancer by adding the following line to the service specification:

```
"type": "LoadBalancer"
```

The following steps use the Kubernetes [Guestbook example](#). This example is a multi-tier web app based on Redis and PHP Docker images. You can specify in the service configuration file that the frontend PHP server uses the Azure load balancer.

1. Download the file `guestbook-all-in-one.yaml` from [GitHub](#).
2. Browse for the `spec` for the `frontend` service.
3. Uncomment the line `type: LoadBalancer`.

```
apiVersion: v1
kind: Service
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  # if your cluster supports it, uncomment the following
  # to automatically create
  # an external load-balanced IP for the frontend service.
  type: LoadBalancer
  ports:
    # the port that this service should serve on
    - port: 80
```

4. Save the file, and deploy the app by running the following command:

```
kubectl create -f guestbook-all-in-one.yaml
```

5. Type `kubectl get svc` to see the state of the services in the cluster. While the load balancer configures the rule, the `EXTERNAL-IP` of the `frontend` service appears as `<pending>`. After a few minutes, the external IP address is configured:

NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
frontend	10.0.89.248	40.71.92.30	80/TCP	6m
kubernetes	10.0.0.1	<none>	443/TCP	5d
redis-master	10.0.56.64	<none>	6379/TCP	6m
redis-slave	10.0.159.7	<none>	6379/TCP	6m

6. Verify that you can access the service at the external IP address. For example, you can open a web browser to the external IP address of the service.



You can add guestbook entries.

7. To see the configuration of the Azure load balancer, browse for the load balancer resource for the cluster in the [Azure portal](#). See the steps in the previous example.

Considerations

- Creation of the load balancer rule happens asynchronously, and information about the provisioned balancer is published in the service's `status.loadBalancer` field.
- Every service is automatically assigned its own virtual IP address in the load balancer.
- If you want to reach the load balancer by a DNS name, work with your domain service provider to create a DNS name for the rule's IP address.

HTTP or HTTPS traffic

To load balance HTTP or HTTPS traffic to container web apps and manage certificates for transport layer security (TLS), you can use the Kubernetes [Ingress](#) resource. An Ingress is a collection of rules that allow inbound connections to reach the cluster services. For an Ingress resource to work, the Kubernetes cluster must have an [Ingress controller](#) running.

Azure Container Service does not implement a Kubernetes Ingress controller automatically. Several controller implementations are available. Currently, the [Nginx Ingress controller](#) is recommended to configure Ingress rules and load balance HTTP and HTTPS traffic.

For more information, see the [Nginx Ingress controller documentation](#).

IMPORTANT

When using the Nginx Ingress controller in Azure Container Service, you must expose the controller deployment as a service with `type: LoadBalancer`. This configures the Azure load balancer to route traffic to the controller. For more information, see the previous section.

Next steps

- See the [Kubernetes LoadBalancer documentation](#)
- Learn more about [Kubernetes Ingress and Ingress controllers](#)
- See [Kubernetes examples](#)

Monitor an Azure Container Service cluster with Microsoft Operations Management Suite (OMS)

8/11/2017 • 3 min to read • [Edit Online](#)

Prerequisites

This walkthrough assumes that you have [created a Kubernetes cluster using Azure Container Service](#).

It also assumes that you have the `az` Azure cli and `kubect1` tools installed.

You can test if you have the `az` tool installed by running:

```
$ az --version
```

If you don't have the `az` tool installed, there are instructions [here](#).

Alternatively, you can use [Azure Cloud Shell](#), which has the `az` Azure cli and `kubect1` tools already installed for you.

You can test if you have the `kubect1` tool installed by running:

```
$ kubect1 version
```

If you don't have `kubect1` installed, you can run:

```
$ az acs kubernetes install-cli
```

To test if you have kubernetes keys installed in your kubect1 tool you can run:

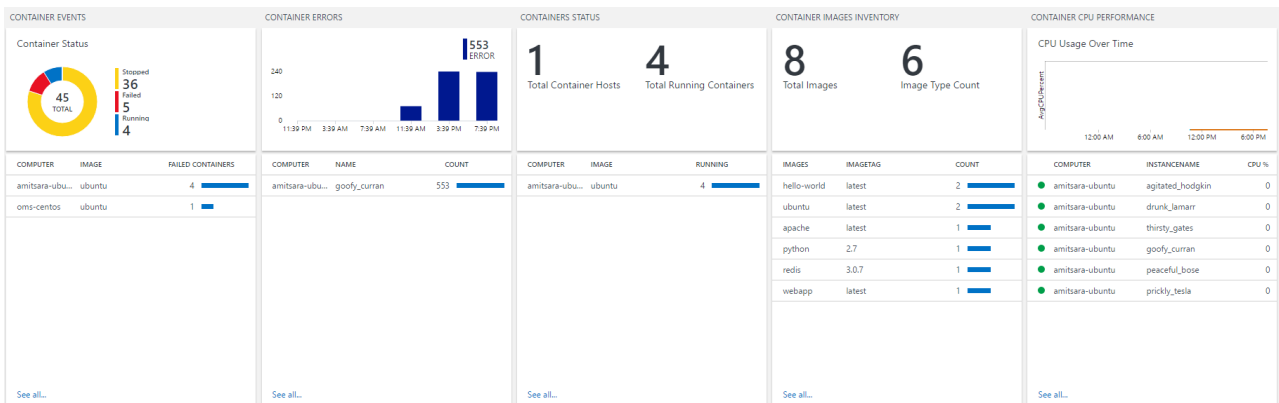
```
$ kubect1 get nodes
```

If the above command errors out, you need to install kubernetes cluster keys into your kubect1 tool. You can do that with the following command:

```
RESOURCE_GROUP=my-resource-group  
CLUSTER_NAME=my-acs-name  
az acs kubernetes get-credentials --resource-group=$RESOURCE_GROUP --name=$CLUSTER_NAME
```

Monitoring Containers with Operations Management Suite (OMS)

Microsoft Operations Management (OMS) is Microsoft's cloud-based IT management solution that helps you manage and protect your on-premises and cloud infrastructure. Container Solution is a solution in OMS Log Analytics, which helps you view the container inventory, performance, and logs in a single location. You can audit, troubleshoot containers by viewing the logs in centralized location, and find noisy consuming excess container on a host.

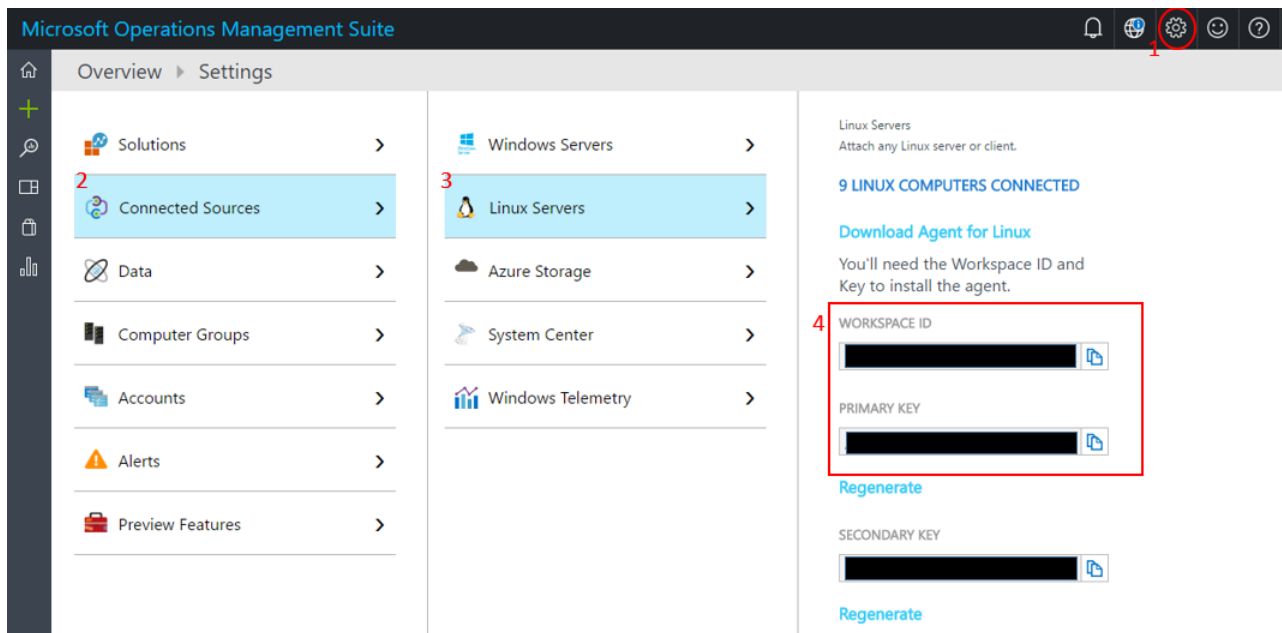


For more information about Container Solution, please refer to the [Container Solution Log Analytics](#).

Installing OMS on Kubernetes

Obtain your workspace ID and key

For the OMS agent to talk to the service it needs to be configured with a workspace id and a workspace key. To get the workspace id and key you need to create an OMS account at <https://mms.microsoft.com>. Please follow the steps to create an account. Once you are done creating the account, you need to obtain your id and key by clicking **Settings**, then **Connected Sources**, and then **Linux Servers**, as shown below.



Install the OMS agent using a DaemonSet

DaemonSets are used by Kubernetes to run a single instance of a container on each host in the cluster. They're perfect for running monitoring agents.

Here is the [DaemonSet YAML file](#). Save it to a file named `oms-daemonset.yaml` and replace the place-holder values for `WSID` and `KEY` with your workspace id and key in the file.

Once you have added your workspace ID and key to the DaemonSet configuration, you can install the OMS agent on your cluster with the `kubectl` command line tool:

```
$ kubectl create -f oms-daemonset.yaml
```

Installing the OMS agent using a Kubernetes Secret

To protect your OMS workspace ID and key you can use Kubernetes Secret as a part of DaemonSet YAML file.

- Copy the script, secret template file and the DaemonSet YAML file (from [repository](#)) and make sure they are on the same directory.
 - secret generating script - secret-gen.sh
 - secret template - secret-template.yaml
 - DaemonSet YAML file - omsagent-ds-secrets.yaml
- Run the script. The script will ask for the OMS Workspace ID and Primary Key. Please insert that and the script will create a secret yaml file so you can run it.

```
#> sudo bash ./secret-gen.sh
```

- Create the secrets pod by running the following: `kubectl create -f omsagentsecret.yaml`
- To check, run the following:

```
root@ubuntu16-13db:~# kubectl get secrets
NAME                                TYPE                                DATA  AGE
default-token-gv191                kubernetes.io/service-account-token  3      50d
omsagent-secret                    Opaque                              2      1d
root@ubuntu16-13db:~# kubectl describe secrets omsagent-secret
Name:          omsagent-secret
Namespace:     default
Labels:        <none>
Annotations:   <none>

Type:  Opaque

Data
====
WSID:   36 bytes
KEY:    88 bytes
```

- Create your omsagent daemon-set by running `kubectl create -f omsagent-ds-secrets.yaml`

Conclusion

That's it! After a few minutes, you should be able to see data flowing to your OMS dashboard.

Monitor an Azure Container Service cluster with DataDog

8/11/2017 • 1 min to read • [Edit Online](#)

Prerequisites

This walkthrough assumes that you have [created a Kubernetes cluster using Azure Container Service](#).

It also assumes that you have the `az` Azure cli and `kubect1` tools installed.

You can test if you have the `az` tool installed by running:

```
$ az --version
```

If you don't have the `az` tool installed, there are instructions [here](#).

You can test if you have the `kubect1` tool installed by running:

```
$ kubect1 version
```

If you don't have `kubect1` installed, you can run:

```
$ az acs kubernetes install-cli
```

DataDog

Datadog is a monitoring service that gathers monitoring data from your containers within your Azure Container Service cluster. Datadog has a Docker Integration Dashboard where you can see specific metrics within your containers. Metrics gathered from your containers are organized by CPU, Memory, Network and I/O. Datadog splits metrics into containers and images.

You first need to [create an account](#)

Installing the Datadog Agent with a DaemonSet

DaemonSets are used by Kubernetes to run a single instance of a container on each host in the cluster. They're perfect for running monitoring agents.

Once you have logged into Datadog, you can follow the [Datadog instructions](#) to install Datadog agents on your cluster using a DaemonSet.

Conclusion

That's it! Once the agents are up and running you should see data in the console in a few minutes. You can visit the integrated [kubernetes dashboard](#) to see a summary of your cluster.

Monitor an Azure Container Service Kubernetes cluster using Sysdig

8/11/2017 • 1 min to read • [Edit Online](#)

Prerequisites

This walkthrough assumes that you have [created a Kubernetes cluster using Azure Container Service](#).

It also assumes that you have the azure cli and kubectl tools installed.

You can test if you have the `az` tool installed by running:

```
$ az --version
```

If you don't have the `az` tool installed, there are instructions [here](#).

You can test if you have the `kubectl` tool installed by running:

```
$ kubectl version
```

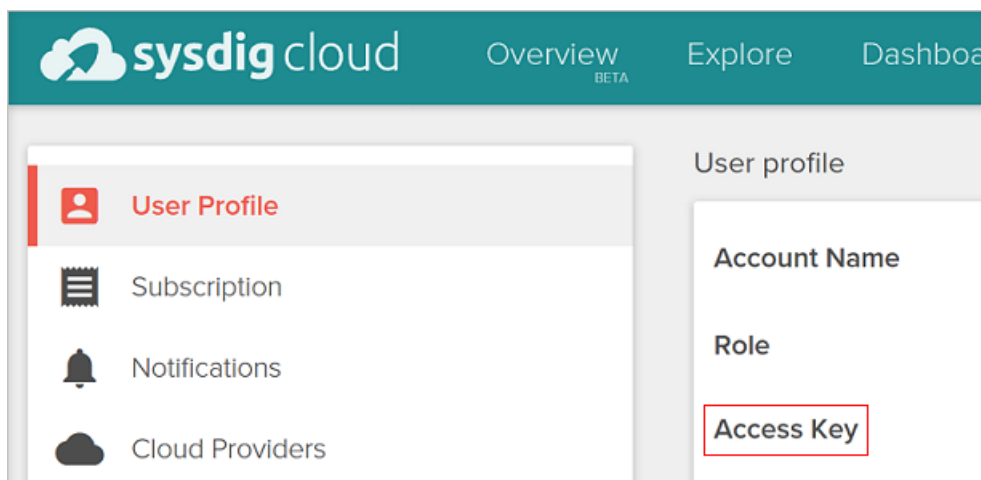
If you don't have `kubectl` installed, you can run:

```
$ az acs kubernetes install-cli
```

Sysdig

Sysdig is an external monitoring as a service company which can monitor containers in your Kubernetes cluster running in Azure. Using Sysdig requires an active Sysdig account. You can sign up for an account on their [site](#).

Once you're logged in to the Sysdig cloud website, click on your user name, and on the page you should see your "Access Key."



Installing the Sysdig agents to Kubernetes

To monitor your containers, Sysdig runs a process on each machine using a Kubernetes `DaemonSet`. DaemonSets

are Kubernetes API objects that run a single instance of a container per machine. They're perfect for installing tools like the Sysdig's monitoring agent.

To install the Sysdig daemonset, you should first download [the template](#) from sysdig. Save that file as

```
sysdig-daemonset.yaml
```

On Linux and OS X you can run:

```
$ curl -O https://raw.githubusercontent.com/draios/sysdig-cloud-scripts/master/agent_deploy/kubernetes/sysdig-daemonset.yaml
```

In PowerShell:

```
$ Invoke-WebRequest -Uri https://raw.githubusercontent.com/draios/sysdig-cloud-scripts/master/agent_deploy/kubernetes/sysdig-daemonset.yaml | Select-Object -ExpandProperty Content > sysdig-daemonset.yaml
```

Next edit that file to insert your Access Key, that you obtained from your Sysdig account.

Finally, create the DaemonSet:

```
$ kubectl create -f sysdig-daemonset.yaml
```

View your monitoring

Once installed and running, the agents should pump data back to Sysdig. Go back to the [sysdig dashboard](#) and you should see information about your containers.

You can also install Kubernetes-specific dashboards via the [new dashboard wizard](#).

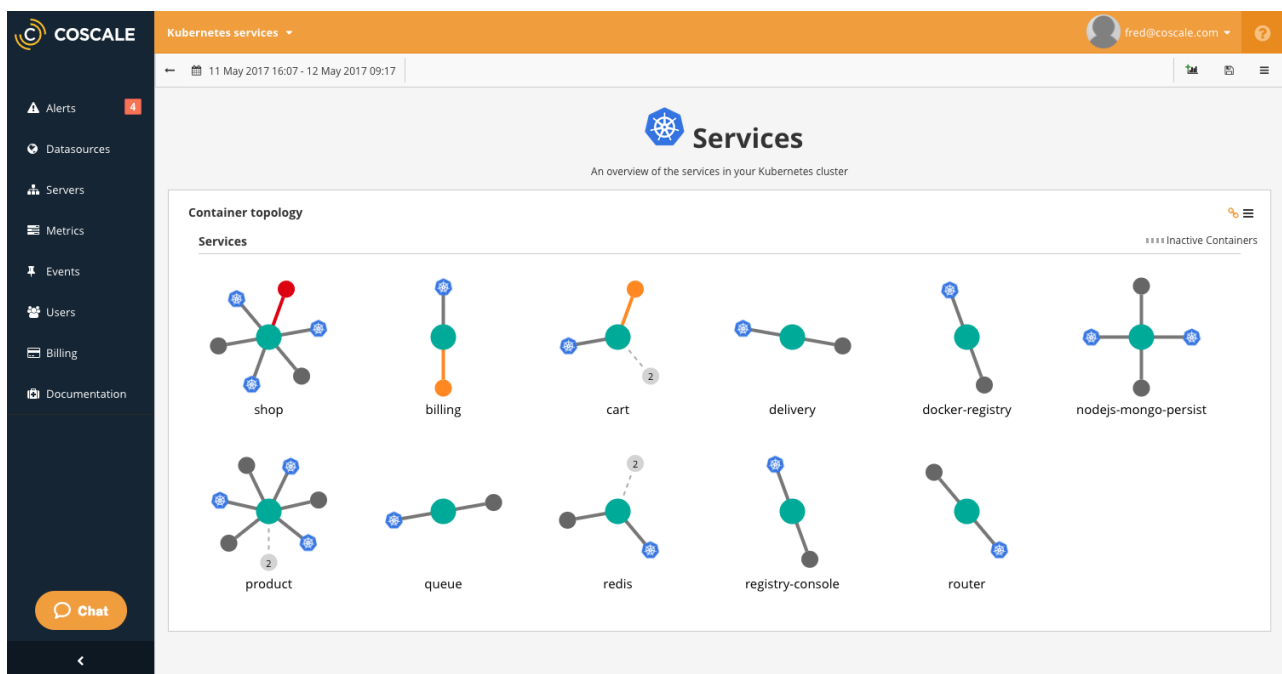
Monitor an Azure Container Service Kubernetes cluster with CoScale

8/11/2017 • 2 min to read • [Edit Online](#)

In this article, we show you how to deploy the [CoScale](#) agent to monitor all nodes and containers in your Kubernetes cluster in Azure Container Service. You need an account with CoScale for this configuration.

About CoScale

CoScale is a monitoring platform that gathers metrics and events from all containers in several orchestration platforms. CoScale offers full-stack monitoring for Kubernetes environments. It provides visualizations and analytics for all layers in the stack: the OS, Kubernetes, Docker, and applications running inside your containers. CoScale offers several built-in monitoring dashboards, and it has built-in anomaly detection to allow operators and developers to find infrastructure and application issues fast.



As shown in this article, you can install agents on a Kubernetes cluster to run CoScale as a SaaS solution. If you want to keep your data on-site, CoScale is also available for on-premises installation.

Prerequisites

You first need to [create a CoScale account](#).

This walkthrough assumes that you have [created a Kubernetes cluster using Azure Container Service](#).

It also assumes that you have the `az` Azure CLI and `kubect1` tools installed.

You can test if you have the `az` tool installed by running:

```
az --version
```

If you don't have the `az` tool installed, there are instructions [here](#).

You can test if you have the `kubect1` tool installed by running:

```
kubect1 version
```

If you don't have `kubect1` installed, you can run:

```
az acs kubernetes install-cli
```

Installing the CoScale agent with a DaemonSet

[DaemonSets](#) are used by Kubernetes to run a single instance of a container on each host in the cluster. They're perfect for running monitoring agents such as the CoScale agent.

After you log in to CoScale, go to the [agent page](#) to install CoScale agents on your cluster using a DaemonSet. The CoScale UI provides guided configuration steps to create an agent and start monitoring your complete Kubernetes cluster.

Configure a new agent

1. Deployment type

2. Operating System / Orchestrator


3. Plugins


4. Summary


5. Download & Install


Select your deployment/orchestration system:


Choose one:


Docker


Docker Swarm


Docker Datacenter


Kubernetes


OpenShift

< Previous Step

To start the agent on the cluster, run the supplied command:

Configure a new agent

1. Deployment type

2. Operating System / Orchestrator

3. Plugins

4. Summary

5. Download & Install

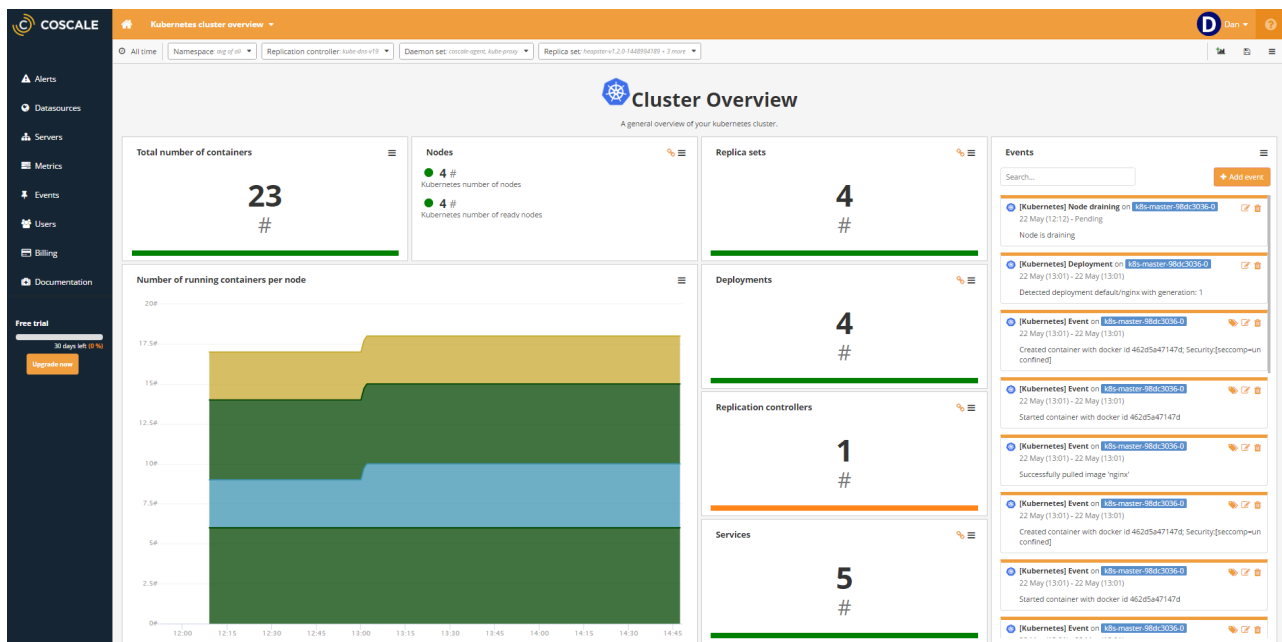
Your agent has been created!

Install instructions

Our agent will run in your Kubernetes cluster as a DaemonSet. Use the following command to start it:

```
cat <<EOF | kubectl apply -f -
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  labels:
    name: coscale-agent
  name: coscale-agent
spec:
  template:
    metadata:
      labels:
        name: coscale-agent
    spec:
      hostNetwork: true
      containers:
        - image: coscale/coscale-agent
          imagePullPolicy: Always
          name: coscale-agent
          env:
            - name: APP_ID
              value: "0179f652-1111-1111-1111-3b874513e0cd"
            - name: ACCESS_TOKEN
              value: "f6d08f23-1111-1111-1111-613fe37d0932"
            - name: TEMPLATE_ID
              value: "7727"
          volumeMounts:
            - name: dockersocket
              mountPath: /var/run/docker.sock
            - name: hostroot
              mountPath: /host
              readOnly: true
          volumes:
            - hostPath:
                path: /var/run/docker.sock
                name: dockersocket
            - hostPath:
                path: /
                name: hostroot
EOF
```

That's it! Once the agents are up and running, you should see data in the console in a few minutes. Visit the [agent page](#) to see a summary of your cluster, perform additional configuration steps, and see dashboards such as the **Kubernetes cluster overview**.



The CoScale agent is automatically deployed on new machines in the cluster. The agent updates automatically when a new version is released.

Next steps

See the [CoScale documentation](#) and [blog](#) for more more information about CoScale monitoring solutions.

Deploy a Spring Boot application on Linux in the Azure Container Service

8/11/2017 • 7 min to read • [Edit Online](#)

The **Spring Framework** is an open-source solution that helps Java developers create enterprise-level applications. One of the more-popular projects that is built on top of that platform is **Spring Boot**, which provides a simplified approach for creating stand-alone Java applications.

Docker is open-source solutions that helps developers automate the deployment, scaling, and management of their applications that are running in containers.

This tutorial walks you through using Docker to develop and deploy a Spring Boot application to a Linux host in the [Azure Container Service \(ACS\)](#).

Prerequisites

In order to complete the steps in this tutorial, you need to have the following prerequisites:

- An Azure subscription; if you don't already have an Azure subscription, you can activate your [MSDN subscriber benefits](#) or sign up for a [free Azure account](#).
- The [Azure Command-Line Interface \(CLI\)](#).
- An up-to-date [Java Developer Kit \(JDK\)](#).
- Apache's [Maven](#) build tool (Version 3).
- A [Git](#) client.
- A [Docker](#) client.

NOTE

Due to the virtualization requirements of this tutorial, you cannot follow the steps in this article on a virtual machine; you must use a physical computer with virtualization features enabled.

Create the Spring Boot on Docker Getting Started web app

The following steps walk you through the steps that are required to create a simple Spring Boot web application and test it locally.

1. Open a command-prompt and create a local directory to hold your application, and change to that directory; for example:

```
md C:\SpringBoot
cd C:\SpringBoot
```

-- or --

```
md /users/robert/SpringBoot
cd /users/robert/SpringBoot
```

2. Clone the [Spring Boot on Docker Getting Started](#) sample project into the directory you created; for example:

```
git clone https://github.com/spring-guides/gs-spring-boot-docker.git
```

3. Change directory to the completed project; for example:

```
cd gs-spring-boot-docker/complete
```

4. Build the JAR file using Maven; for example:

```
mvn package
```

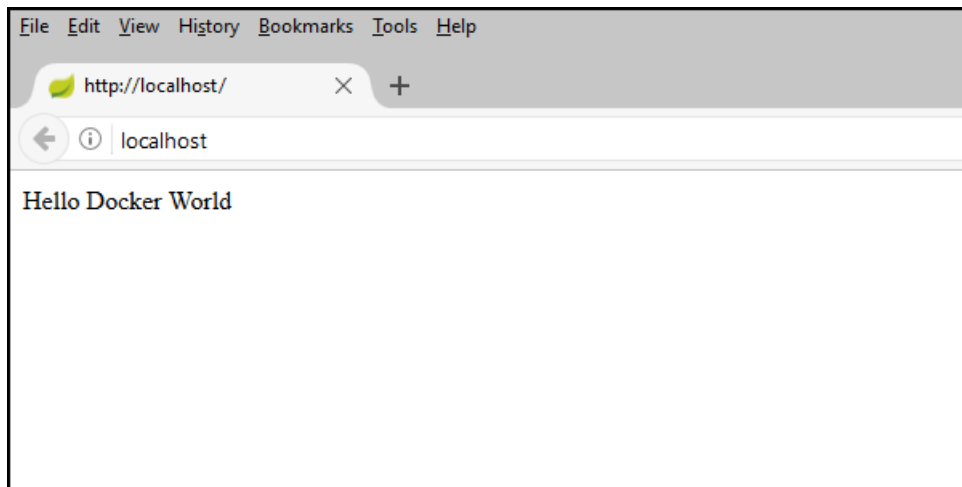
5. Once the web app has been created, change directory to the `target` directory where the JAR file is located and start the web app; for example:

```
cd target
java -jar gs-spring-boot-docker-0.1.0.jar
```

6. Test the web app by browsing to it locally using a web browser. For example, if you have curl available and you configured the Tomcat server to run on port 80:

```
curl http://localhost
```

7. You should see the following message displayed: **Hello Docker World!**



Create an Azure Container Registry to use as a Private Docker Registry

The following steps walk you through using the Azure portal to create an Azure Container Registry.

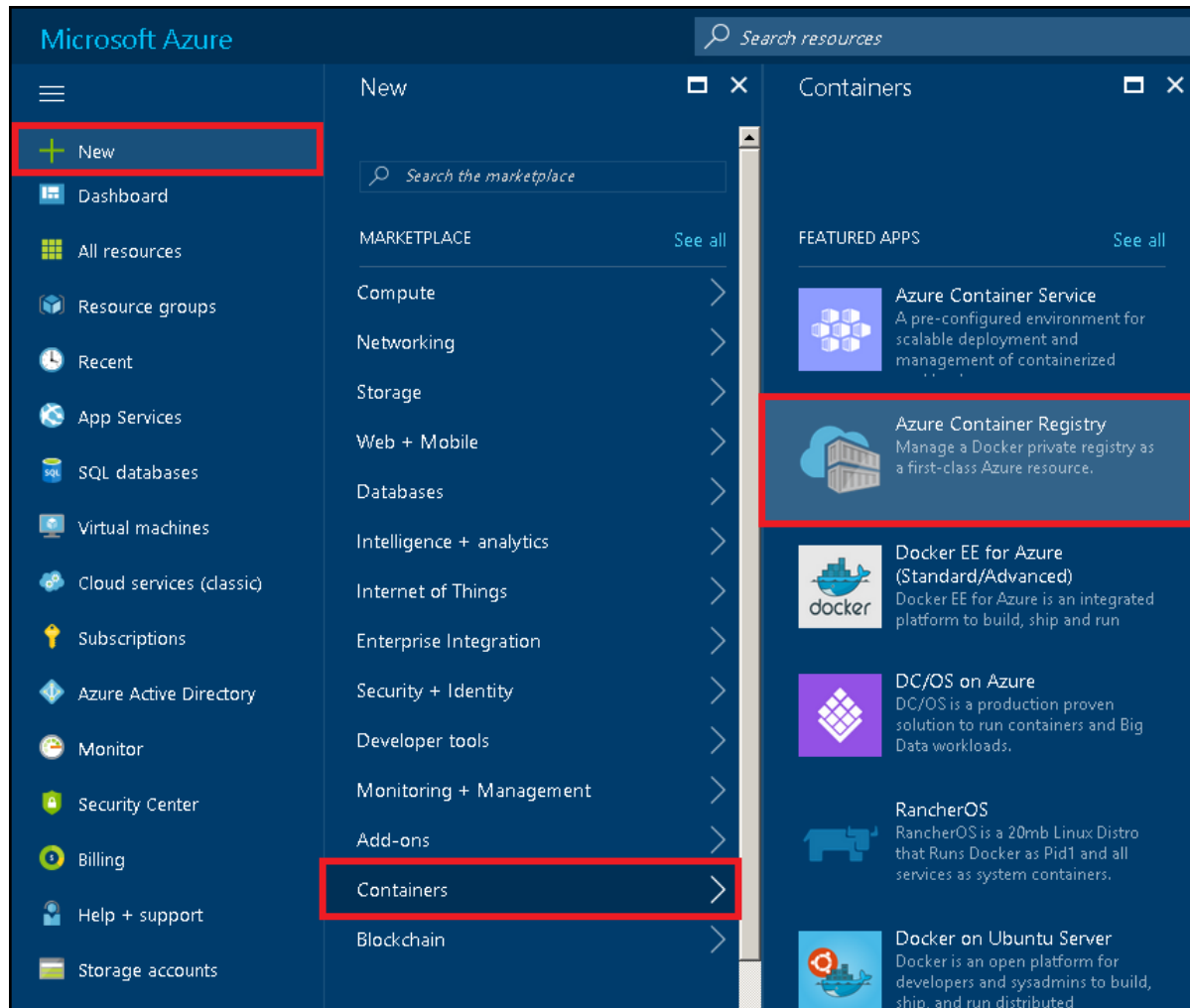
NOTE

If you want to use the Azure CLI instead of the Azure portal, follow the steps in [Create a private Docker container registry using the Azure CLI 2.0](#).

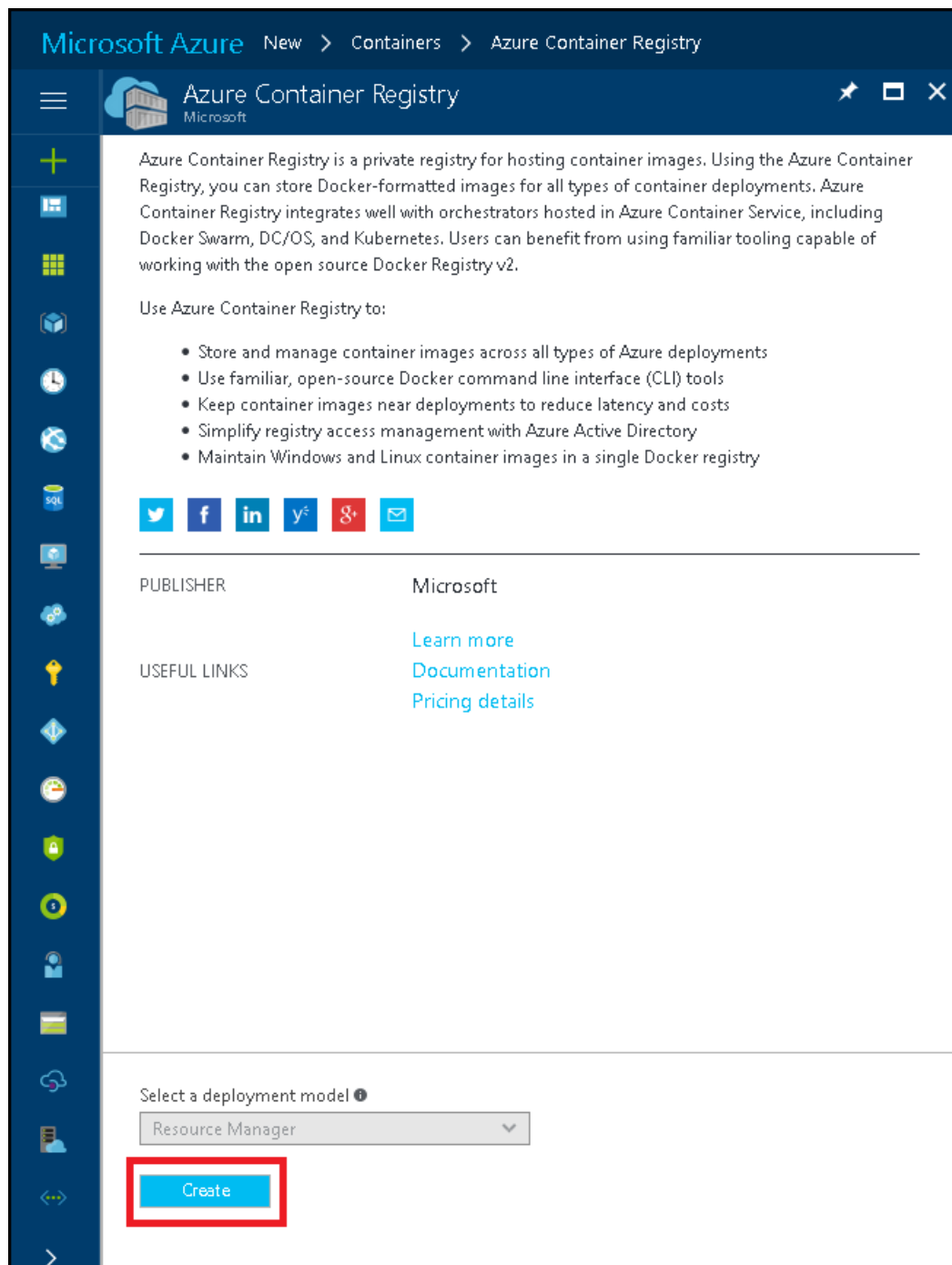
1. Browse to the [Azure portal](#) and sign in.

Once you have signed in to your account on the Azure portal, you can follow the steps in the [Create a private Docker container registry using the Azure portal](#) article, which are paraphrased in the following steps for the sake of expediency.

2. Click the menu icon for **+ New**, then click **Containers**, and then click **Azure Container Registry**.



3. When the information page for the Azure Container Registry template is displayed, click **Create**.



4. When the **Create container registry** page is displayed, enter your **Registry name** and **Resource group**, choose **Enable** for the **Admin user**, and then click **Create**.

Microsoft Azure New > Containers > Azure Container Registry > Create container registry

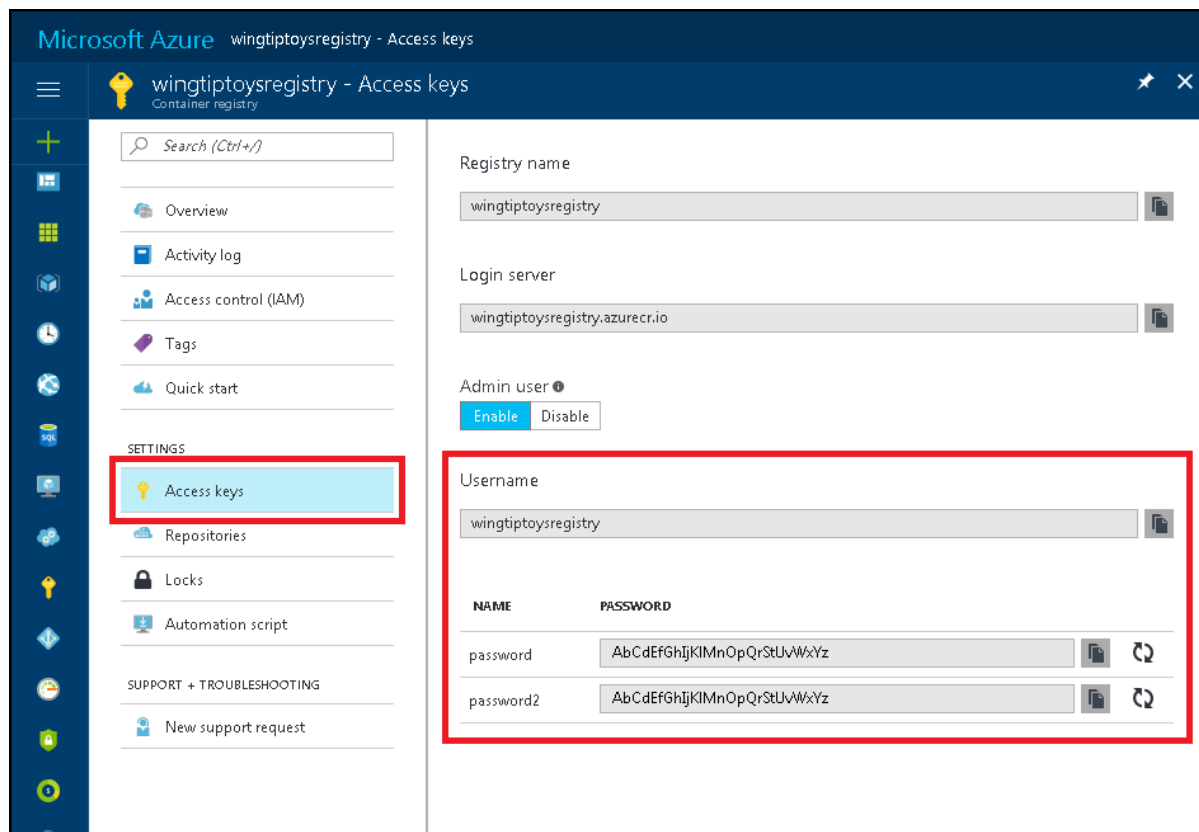
Create container registry

- * Registry name
wingtiptoyregistry ✓
azurecr.io
- * Subscription
Converted Windows Azure MSDN - Visual St
- * Resource group ⓘ
☒ Create new ☐ Use existing
wingtiptoyresources ✓
- * Location
South Central US
- * Admin user ⓘ
- * Storage account
(new) wingtiptoyregistr231643 >

☐ Pin to dashboard

[Automation options](#)

5. Once your container registry has been created, navigate to your container registry in the Azure portal, and then click **Access Keys**. Take note of the username and password for the next steps.



Configure Maven to use your Azure Container Registry access keys

1. Navigate to the configuration directory for your Maven installation and open the *settings.xml* file with a text editor.
2. Add your Azure Container Registry access settings from the previous section of this tutorial to the `<servers>` collection in the *settings.xml* file; for example:

```
<servers>
  <server>
    <id>wingtiptoyregistry</id>
    <username>wingtiptoyregistry</username>
    <password>AbCdEfGhIjKlMnOpQrStUvWxYz</password>
  </server>
</servers>
```

3. Navigate to the completed project directory for your Spring Boot application, (for example: "C:\SpringBoot\gs-spring-boot-docker\complete" or "/users/robert/SpringBoot/gs-spring-boot-docker/complete"), and open the *pom.xml* file with a text editor.
4. Update the `<properties>` collection in the *pom.xml* file with the login server value for your Azure Container Registry from the previous section of this tutorial; for example:

```
<properties>
  <docker.image.prefix>wingtiptoyregistry.azurecr.io</docker.image.prefix>
  <java.version>1.8</java.version>
</properties>
```

5. Update the `<plugins>` collection in the *pom.xml* file so that the `<plugin>` contains the login server address and registry name for your Azure Container Registry from the previous section of this tutorial. For example:

```
<plugin>
  <groupId>com.spotify</groupId>
  <artifactId>docker-maven-plugin</artifactId>
  <version>0.4.11</version>
  <configuration>
    <imageName>${docker.image.prefix}/${project.artifactId}</imageName>
    <dockerDirectory>src/main/docker</dockerDirectory>
    <resources>
      <resource>
        <targetPath></targetPath>
        <directory>${project.build.directory}</directory>
        <include>${project.build.finalName}.jar</include>
      </resource>
    </resources>
    <serverId>wingtiptoyregistry</serverId>
    <registryUrl>https://wingtiptoyregistry.azurecr.io</registryUrl>
  </configuration>
</plugin>
```

6. Navigate to the completed project directory for your Spring Boot application and run the following command to rebuild the application and push the container to your Azure Container Registry:

```
mvn package docker:build -DpushImage
```

NOTE

When you are pushing your Docker container to Azure, you may receive an error message that is similar to one of the following even though your Docker container was created successfully:

- [ERROR] Failed to execute goal com.spotify:docker-maven-plugin:0.4.11:build (default-cli) on project gs-spring-boot-docker: Exception caught: no basic auth credentials
- [ERROR] Failed to execute goal com.spotify:docker-maven-plugin:0.4.11:build (default-cli) on project gs-spring-boot-docker: Exception caught: Incomplete Docker registry authorization credentials. Please provide all of username, password, and email or none.

If this happens, you may need to sign in to your Azure account from the Docker command line; for example:

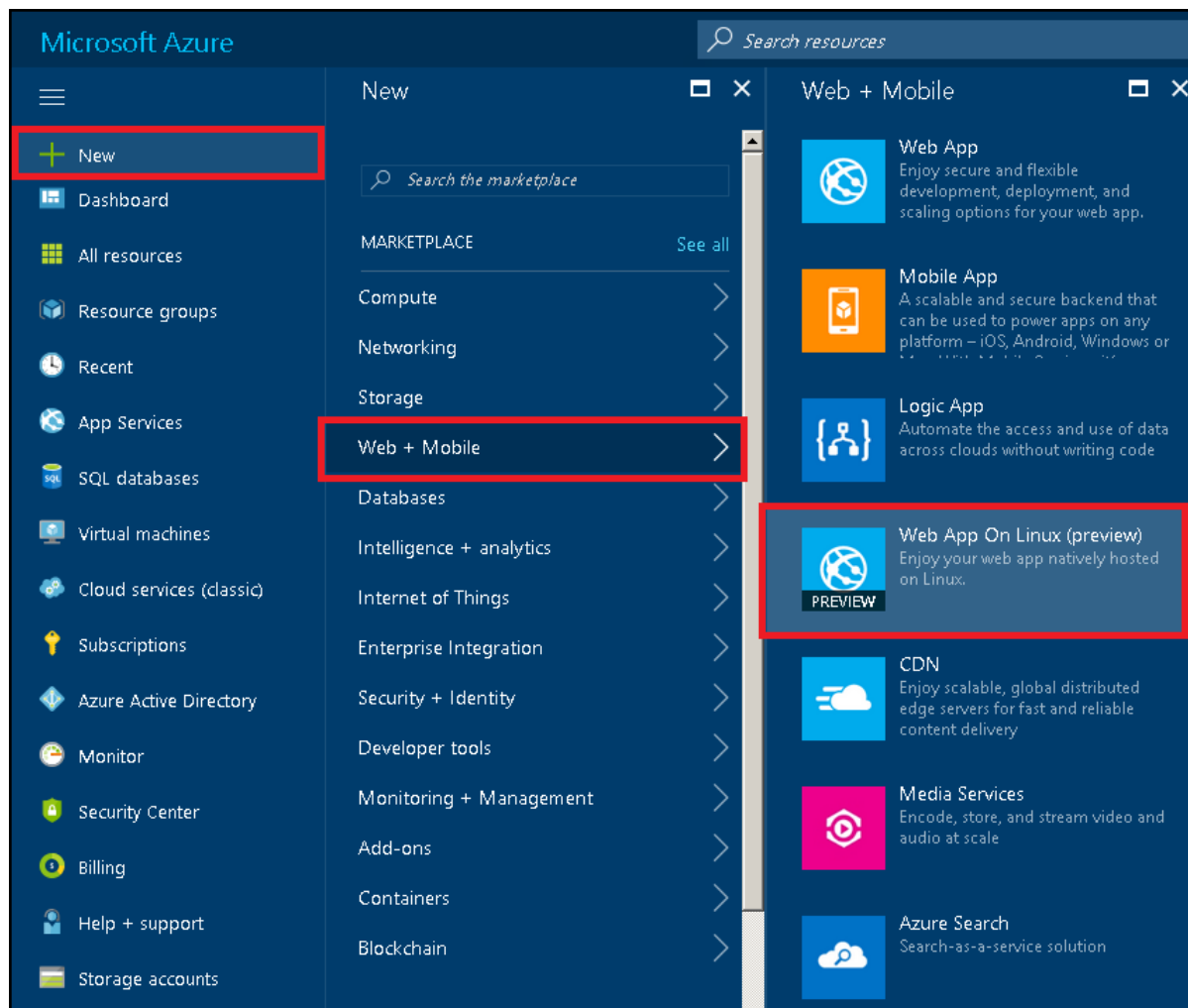
```
docker login -u wingtiptoyregistry -p "AbCdEfGhIjKlMnOpQrStUvWxYz" wingtiptoyregistry.azurecr.io
```

You can then push your container from the command line; for example:

```
docker push wingtiptoyregistry.azurecr.io/gs-spring-boot-docker
```

Create a web app on Linux on Azure App Service using your container image

1. Browse to the [Azure portal](#) and sign in.
2. Click the menu icon for **+ New**, then click **Web + Mobile**, and then click **Web App on Linux**.



3. When the **Web App on Linux** page is displayed, enter the following information:

- a. Enter a unique name for the **App name**; for example: "wingtiptoylinux."
- b. Choose your **Subscription** from the drop-down list.
- c. Choose an existing **Resource Group**, or specify a name to create a new resource group.
- d. Click **Configure container** and enter the following information:
 - Choose **Private registry**.
 - **Image and optional tag**: Specify your container name from earlier; for example:
"wingtiptoyregistry.azurecr.io/gs-spring-boot-docker:latest"
 - **Server URL**: Specify your registry URL from earlier; for example:
"<https://wingtiptoyregistry.azurecr.io>"
 - **Login username** and **Password**: Specify your login credentials from your **Access Keys** that you used in previous steps.
- e. Once you have entered all of the above information, click **OK**.

Microsoft Azure

New > Web + Mobile > Web App On Linux (preview) > Docker Container

Web App On Linux (preview)

Create

Docker Container

+

Home

Apps

Subscriptions

Resource Groups

Cloud Shell

Monitor

Alerts

Log Analytics

Automation

Key Vault

Network

Storage

Virtual Machines

Web Apps

Containers

APIs

DevOps

Security

Compliance

Support

Feedback

* App name

wingtiptoylinux

.azurewebsites.net

* Subscription

Converted Windows Azure MSDN - Visual St

* Resource Group

Create new Use existing

wingtiptoyresources

* App Service plan/Location

ServicePlan400580e4-b3b2(West...


* Configure container

node 4.5.0

☐ Pin to dashboard

Create

Automation options

 Docker Container

Web Apps on Linux leverage the power of Docker containers to let you use custom containers from Azure Container Registry, Docker Hub, a private container registry, or use one of our default containers provided by App Service.

Image source

Built-in Docker Hub Private registry

* Image and optional tag (eg 'image:tag')

wingtiptoyregistry.azurecr.io/gs-spring-boot-docker

* Server URL

https://wingtiptoyregistry.azurecr.io

* Login username

wingtiptoyregistry

* Password

.....

Startup File

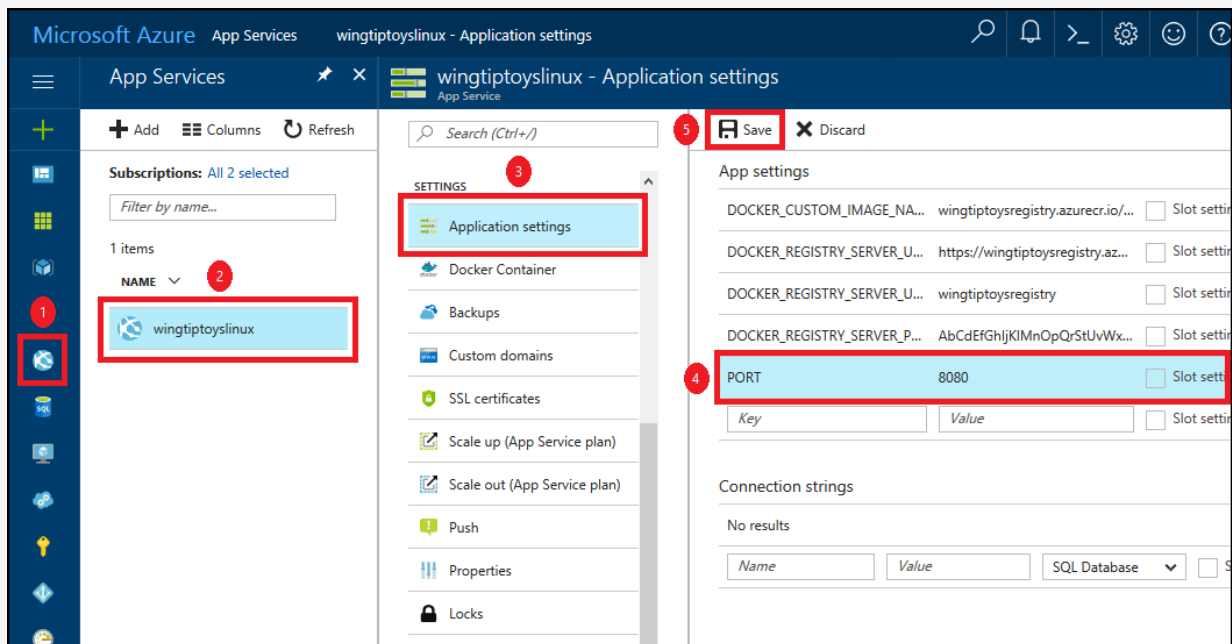
OK

4. Click **Create**.

NOTE

Azure will automatically map Internet requests to embedded Tomcat server that is running on the standard ports of 80 or 8080. However, if you configured your embedded Tomcat server to run on a custom port, you need to add an environment variable to your web app that defines the port for your embedded Tomcat server. To do so, use the following steps:

1. Browse to the [Azure portal](#) and sign in.
2. Click the icon for **App Services**. (See item #1 in the image below.)
3. Select your web app from the list. (Item #2 in the image below.)
4. Click **Application Settings**. (Item #3 in the image below.)
5. In the **App settings** section, add a new environment variable named **PORT** and enter your custom port number for the value. (Item #4 in the image below.)
6. Click **Save**. (Item #5 in the image below.)



Next steps

For more information about using Spring Boot applications on Azure, see the following articles:

- [Deploy a Spring Boot Application to the Azure App Service](#)
- [Deploy a Spring Boot Application on a Kubernetes Cluster in the Azure Container Service](#)

For more information about using Azure with Java, see the [Azure Java Developer Center](#) and the [Java Tools for Visual Studio Team Services](#).

For further details about the Spring Boot on Docker sample project, see [Spring Boot on Docker Getting Started](#).

For help with getting started with your own Spring Boot applications, see the **Spring Initializr** at <https://start.spring.io/>.

For more information about getting started with creating a simple Spring Boot application, see the Spring Initializr at <https://start.spring.io/>.

For additional examples for how to use custom Docker images with Azure, see [Using a custom Docker image for Azure Web App on Linux](#).

Deploy a Spring Boot Application on a Kubernetes Cluster in the Azure Container Service

8/11/2017 • 7 min to read • [Edit Online](#)

The **Spring Framework** is a popular open-source framework that helps Java developers create web, mobile, and API applications. This tutorial uses a sample app created using **Spring Boot**, a convention-driven approach for using Spring to get started quickly.

Kubernetes and **Docker** are open-source solutions that help developers automate the deployment, scaling, and management of their applications running in containers.

This tutorial walks you through combining these two popular, open-source technologies to develop and deploy a Spring Boot application to Microsoft Azure. More specifically, you use **Spring Boot** for application development, **Kubernetes** for container deployment, and the **Azure Container Service (ACS)** to host your application.

Prerequisites

- An Azure subscription; if you don't already have an Azure subscription, you can activate your [MSDN subscriber benefits](#) or sign up for a [free Azure account](#).
- The [Azure Command-Line Interface \(CLI\)](#).
- An up-to-date [Java Developer Kit \(JDK\)](#).
- Apache's [Maven](#) build tool (Version 3).
- A [Git](#) client.
- A [Docker](#) client.

NOTE

Due to the virtualization requirements of this tutorial, you cannot follow the steps in this article on a virtual machine; you must use a physical computer with virtualization features enabled.

Create the Spring Boot on Docker Getting Started web app

The following steps walk you through building a Spring Boot web application and testing it locally.

1. Open a command-prompt and create a local directory to hold your application, and change to that directory; for example:

```
md C:\SpringBoot
cd C:\SpringBoot
```

-- or --

```
md /users/robert/SpringBoot
cd /users/robert/SpringBoot
```

2. Clone the [Spring Boot on Docker Getting Started](#) sample project into the directory.

```
git clone https://github.com/spring-guides/gs-spring-boot-docker.git
```


3. Change directory to the completed project.

```
cd gs-spring-boot-docker
cd complete
```

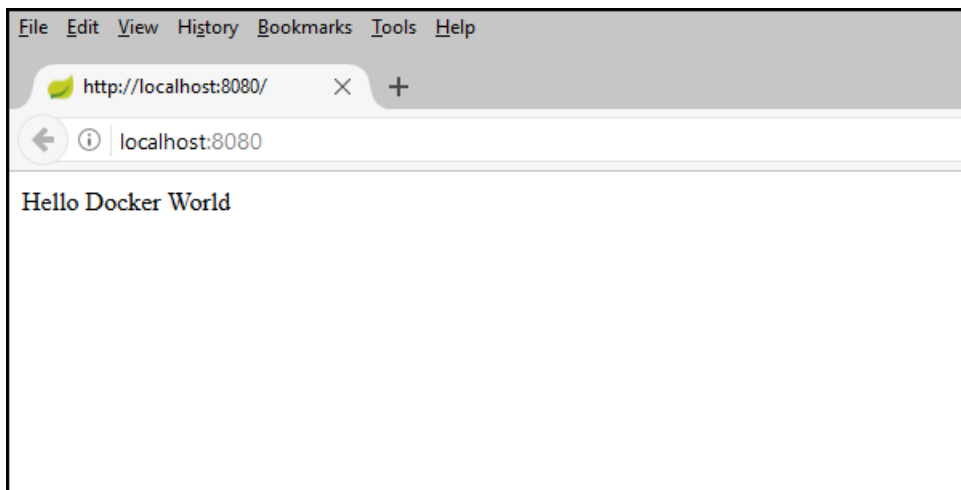
4. Use Maven to build and run the sample app.

```
mvn package spring-boot:run
```

5. Test the web app by browsing to <http://localhost:8080>, or with the following `curl` command:

```
curl http://localhost:8080
```

6. You should see the following message displayed: **Hello Docker World**



Create an Azure Container Registry using the Azure CLI

1. Open a command prompt.
2. Log in to your Azure account:

```
az login
```

3. Create a resource group for the Azure resources used in this tutorial.

```
az group create --name=wingtiptoy-kubernetes --location=eastus
```

4. Create a private Azure container registry in the resource group. The tutorial pushes the sample app as a Docker image to this registry in later steps. Replace `wingtiptoyregistry` with a unique name for your registry.

```
az acr create --admin-enabled --resource-group wingtiptoy-kubernetes --location eastus \
--name wingtiptoyregistry --sku Basic
```

Push your app to the container registry

1. Navigate to the configuration directory for your Maven installation (default `~/m2/` or

C:\Users\username.m2) and open the *settings.xml* file with a text editor.

2. Retrieve the password for your container registry from the Azure CLI.

```
az acr credential show --name wingtiptoyregistry --query passwords[0]
```

```
{
  "name": "password",
  "value": "AbCdEfGhIjKlMnOpQrStUvWxYz"
}
```

3. Add your Azure Container Registry id and password to a new `<server>` collection in the *settings.xml* file. The `id` and `username` are the name of the registry. Use the `password` value from the previous command (without quotes).

```
<servers>
  <server>
    <id>wingtiptoyregistry</id>
    <username>wingtiptoyregistry</username>
    <password>AbCdEfGhIjKlMnOpQrStUvWxYz</password>
  </server>
</servers>
```

4. Navigate to the completed project directory for your Spring Boot application (for example, "C:\SpringBoot\gs-spring-boot-docker\complete" or "/users/robert/SpringBoot/gs-spring-boot-docker/complete"), and open the *pom.xml* file with a text editor.
5. Update the `<properties>` collection in the *pom.xml* file with the login server value for your Azure Container Registry.

```
<properties>
  <docker.image.prefix>wingtiptoyregistry.azurecr.io</docker.image.prefix>
  <java.version>1.8</java.version>
</properties>
```

6. Update the `<plugins>` collection in the *pom.xml* file so that the `<plugin>` contains the login server address and registry name for your Azure Container Registry.

```
<plugin>
  <groupId>com.spotify</groupId>
  <artifactId>docker-maven-plugin</artifactId>
  <version>0.4.11</version>
  <configuration>
    <imageName>${docker.image.prefix}/${project.artifactId}</imageName>
    <dockerDirectory>src/main/docker</dockerDirectory>
    <resources>
      <resource>
        <targetPath></targetPath>
        <directory>${project.build.directory}</directory>
        <include>${project.build.finalName}.jar</include>
      </resource>
    </resources>
    <serverId>wingtiptoyregistry</serverId>
    <registryUrl>https://wingtiptoyregistry.azurecr.io</registryUrl>
  </configuration>
</plugin>
```

7. Navigate to the completed project directory for your Spring Boot application and run the following command to build the Docker container and push the image to the registry:

```
mvn package docker:build -DpushImage
```

NOTE

You may receive an error message that is similar to one of the following when Maven pushes the image to Azure:

- ```
[ERROR] Failed to execute goal com.spotify:docker-maven-plugin:0.4.11:build (default-cli) on project gs-spring-boot-docker: Exception caught: no basic auth credentials
```
- ```
[ERROR] Failed to execute goal com.spotify:docker-maven-plugin:0.4.11:build (default-cli) on project gs-spring-boot-docker: Exception caught: Incomplete Docker registry authorization credentials. Please provide all of username, password, and email or none.
```

If you get this error, log in to Azure from the Docker command line.

```
docker login -u wingtiptoyregistry -p "AbCdEfGhIjKlMnOpQrStUvWxYz" wingtiptoyregistry.azurecr.io
```

Then push your container:

```
docker push wingtiptoyregistry.azurecr.io/gs-spring-boot-docker
```

Create a Kubernetes Cluster on ACS using the Azure CLI

1. Create a Kubernetes cluster in Azure Container Service. The following command creates a *kubernetes* cluster in the *wingtiptoy-kubernetes* resource group, with *wingtiptoy-containerservice* as the cluster name, and *wingtiptoy-kubernetes* as the DNS prefix:

```
az acs create --orchestrator-type=kubernetes --resource-group=wingtiptoy-kubernetes \
--name=wingtiptoy-containerservice --dns-prefix=wingtiptoy-kubernetes
```

This command may take a while to complete.

2. Install `kubect1` using the Azure CLI. Linux users may have to prefix this command with `sudo` since it deploys the Kubernetes CLI to `/usr/local/bin`.

```
az acs kubernetes install-cli
```

3. Download the cluster configuration information so you can manage your cluster from the Kubernetes web interface and `kubect1`.

```
az acs kubernetes get-credentials --resource-group=wingtiptoy-kubernetes \
--name=wingtiptoy-containerservice
```

Deploy the image to your Kubernetes cluster

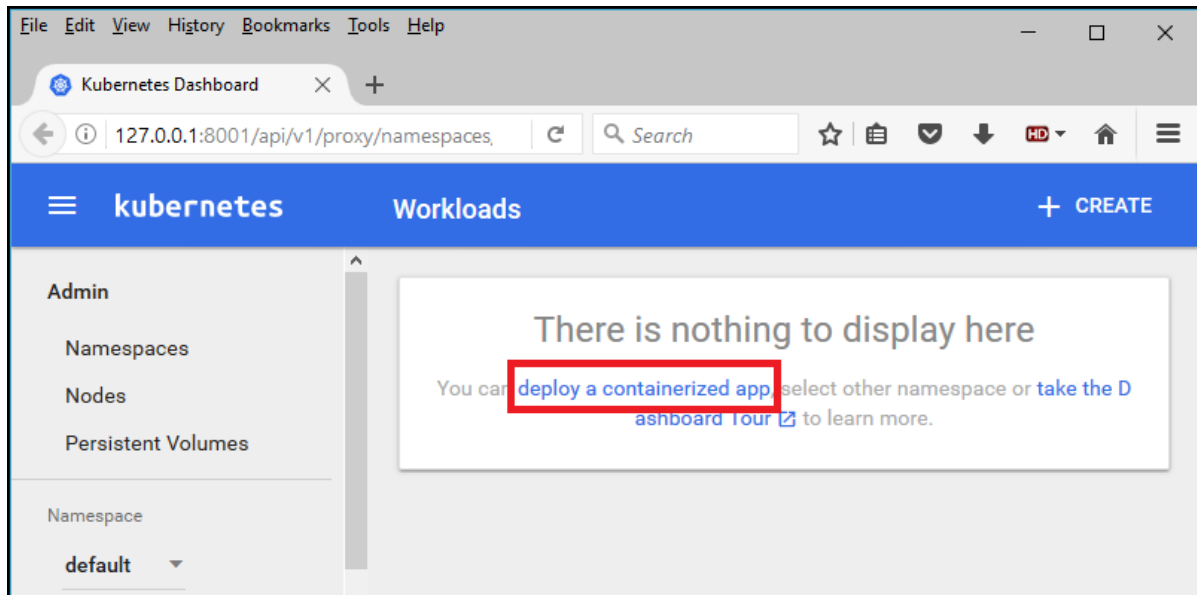
This tutorial deploys the app using `kubect1`, then allow you to explore the deployment through the Kubernetes web interface.

Deploy with the Kubernetes web interface

1. Open a command prompt.
2. Open the configuration website for your Kubernetes cluster in your default browser:

```
az aks kubernetes browse --resource-group=wingtiptoys-kubernetes --name=wingtiptoys-containerservice
```

3. When the Kubernetes configuration website opens in your browser, click the link to **deploy a containerized app**:



4. When the **Deploy a containerized app** page is displayed, specify the following options:
- Select **Specify app details below**.
 - Enter your Spring Boot application name for the **App name**; for example: `"gs-spring-boot-docker"`.
 - Enter your login server and container image from earlier for the **Container image**; for example: `"wingtiptoysregistry.azurecr.io/gs-spring-boot-docker:latest"`.
 - Choose **External** for the **Service**.
 - Specify your external and internal ports in the **Port** and **Target port** text boxes.

File Edit View History Bookmarks Tools Help

Kubernetes Dashboard

127.0.0.1:8001/api/v1/proxy/namespaces/kube-system/services

Search

kubernetes Create an app + CREATE

Admin

- Namespaces
- Nodes
- Persistent Volumes

Namespace

default

Workloads

- Deployments
- Replica Sets
- Replication Controllers
- Daemon Sets
- Stateful Sets
- Jobs
- Pods

Services and discovery

Deploy a Containerized App

☒ Specify app details below [To learn more, take the Dashboard Tour](#)

☐ Upload a YAML or JSON file

App name *

gs-spring-boot-docker 21 / 24

Container image *

wingtiptoyregistry.azurecr.io/gs-spring-boot-doc

Number of pods *

1

Service *

External

Port * Target port * Protocol *

80 8080 TCP

An 'app' label with this value will be added to the Deployment and Service that get deployed. [Learn more](#)

Enter the URL of a public image on any registry, or a private image hosted on Docker Hub or Google Container Registry. [Learn more](#)

A Deployment will be created to maintain the desired number of pods across your cluster. [Learn more](#)

Optionally, an internal or external Service can be defined to map an incoming Port to a target Port seen by the container. The internal DNS name for this Service will be: **gs-spring-boot-docker**. [Learn more](#)

5. Click **Deploy** to deploy the container.

File Edit View History Bookmarks Tools Help

Kubernetes Dashboard

127.0.0.1:8001/api/v1/proxy/namespaces/kube-system/services

Search

kubernetes Create an app + CREATE

Admin

- Namespaces
- Nodes
- Persistent Volumes

Services and discovery

- Services
- Ingresses
- Storage

Deploy a Containerized App

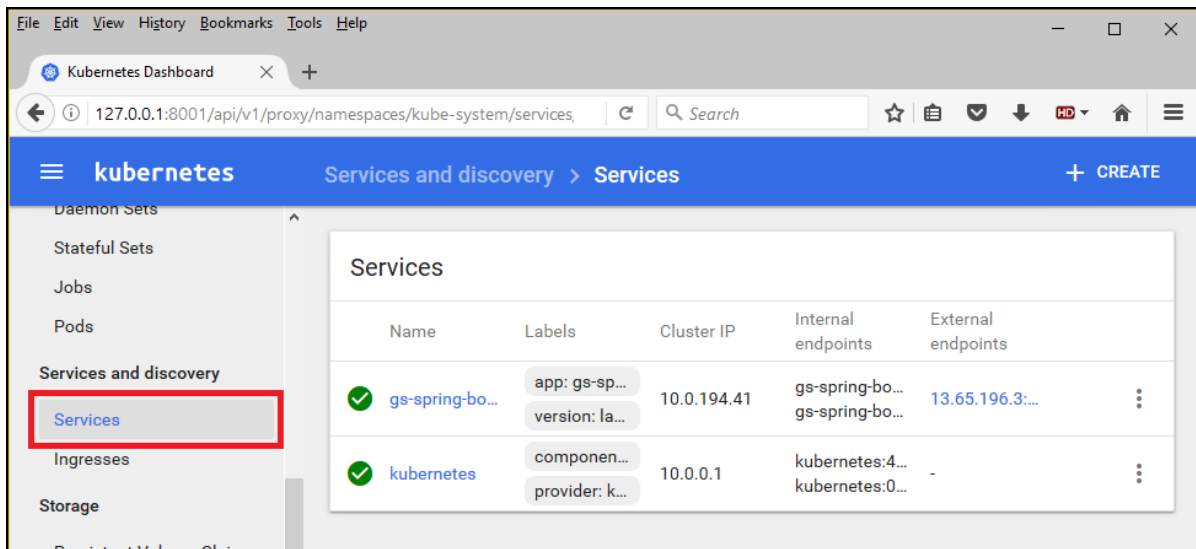
☒ Specify app details below [To learn more, take the Dashboard Tour](#)

☐ Upload a YAML or JSON file

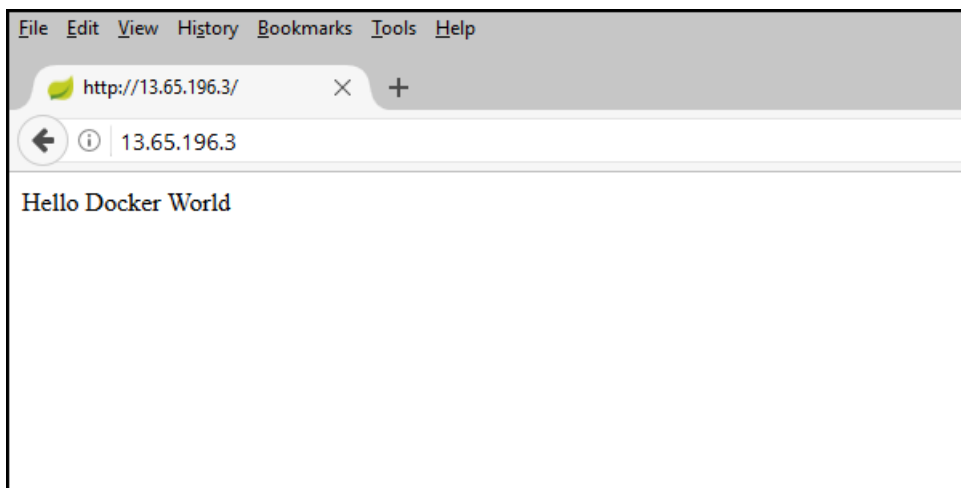
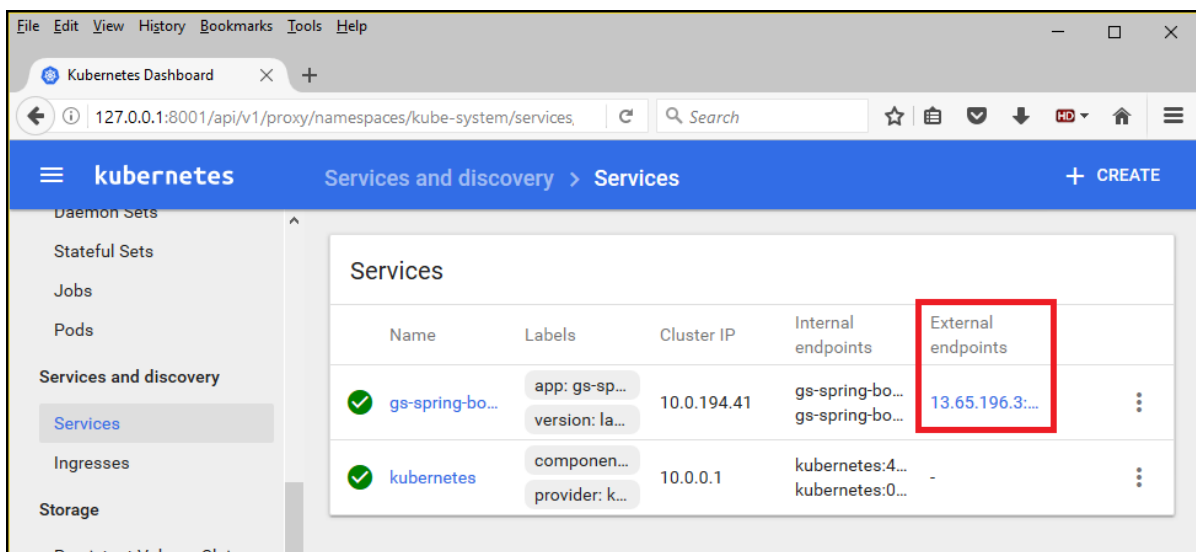
[SHOW ADVANCED OPTIONS](#)

DEPLOY CANCEL

6. Once your application has been deployed, you will see your Spring Boot application listed under **Services**.



7. If you click the link for **External endpoints**, you can see your Spring Boot application running on Azure.



Deploy with kubectl

1. Open a command prompt.
2. Run your container in the Kubernetes cluster by using the `kubectl run` command. Give a service name for your app in Kubernetes and the full image name. For example:

```
kubectl run gs-spring-boot-docker --image=wingtiptoyregistry.azurecr.io/gs-spring-boot-docker:latest
```

In this command:

- The container name `gs-spring-boot-docker` is specified immediately after the `run` command
- The `--image` parameter specifies the combined login server and image name as `wingtiptoyregistry.azurecr.io/gs-spring-boot-docker:latest`

3. Expose your Kubernetes cluster externally by using the `kubectl expose` command. Specify your service name, the public-facing TCP port used to access the app, and the internal target port your app listens on. For example:

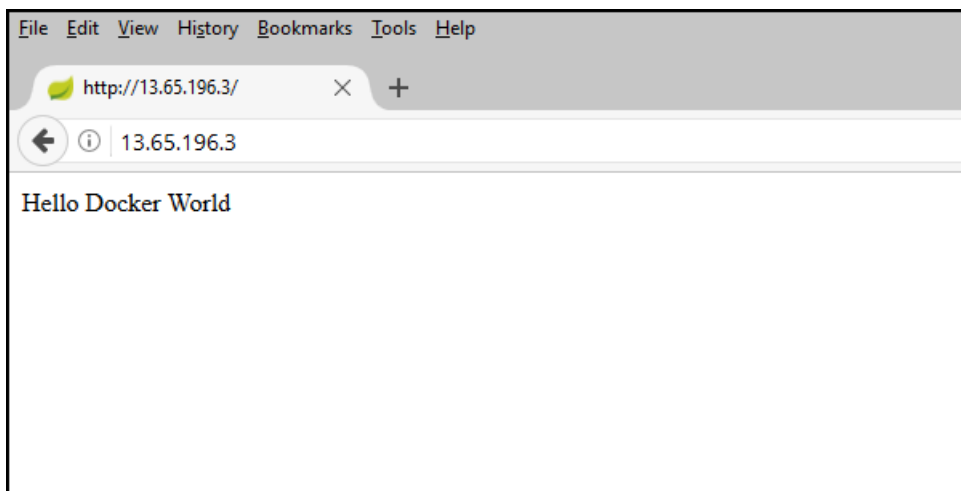
```
kubectl expose deployment gs-spring-boot-docker --type=LoadBalancer --port=80 --target-port=8080
```

In this command:

- The container name `gs-spring-boot-docker` is specified immediately after the `expose deployment` command
- The `--type` parameter specifies that the cluster uses load balancer
- The `--port` parameter specifies the public-facing TCP port of 80. You access the app on this port.
- The `--target-port` parameter specifies the internal TCP port of 8080. The load balancer forwards requests to your app on this port.

4. Once the app is deployed to the cluster, query the external IP address and open it in your web browser:

```
kubectl get services -o jsonpath={.items[*].status.loadBalancer.ingress[0].ip} --namespace=${namespace}
```



Next steps

For more information about using Spring Boot on Azure, see the following articles:

- [Deploy a Spring Boot Application to the Azure App Service](#)
- [Deploy a Spring Boot application on Linux in the Azure Container Service](#)

For more information about using Azure with Java, see the [Azure Java Developer Center](#) and the [Java Tools for Visual Studio Team Services](#).

For more information about the Spring Boot on Docker sample project, see [Spring Boot on Docker Getting Started](#).

The following links provide additional information about creating Spring Boot applications:

- For more information about creating a simple Spring Boot application, see the Spring Initializr at <https://start.spring.io/>.

The following links provide additional information about using Kubernetes with Azure:

- [Get started with a Kubernetes cluster in Container Service](#)
- [Using the Kubernetes web UI with Azure Container Service](#)

More information about using Kubernetes command-line interface is available in the **kubectl** user guide at <https://kubernetes.io/docs/user-guide/kubectl/>.

The Kubernetes website has several articles that discuss using images in private registries:

- [Configuring Service Accounts for Pods](#)
- [Namespaces](#)
- [Pulling an Image from a Private Registry](#)

For additional examples for how to use custom Docker images with Azure, see [Using a custom Docker image for Azure Web App on Linux](#).

Deploy a Spring Boot app using the Fabric8 Maven Plugin

9/25/2017 • 8 min to read • [Edit Online](#)

Fabric8 is an open-source solution that is built on **Kubernetes**, which helps developers create applications in Linux containers.

This tutorial walks you through using the Fabric8 plugin for Maven to develop to deploy an application to a Linux host in the [Azure Container Service \(ACS\)](#).

Prerequisites

In order to complete the steps in this tutorial, you need to have the following prerequisites:

- An Azure subscription; if you don't already have an Azure subscription, you can activate your [MSDN subscriber benefits](#) or sign up for a [free Azure account](#).
- The [Azure Command-Line Interface \(CLI\)](#).
- An up-to-date [Java Developer Kit \(JDK\)](#).
- Apache's [Maven](#) build tool (Version 3).
- A [Git](#) client.
- A [Docker](#) client.

NOTE

Due to the virtualization requirements of this tutorial, you cannot follow the steps in this article on a virtual machine; you must use a physical computer with virtualization features enabled.

Create the Spring Boot on Docker Getting Started web app

The following steps walk you through building a Spring Boot web application and testing it locally.

1. Open a command-prompt and create a local directory to hold your application, and change to that directory; for example:

```
md /home/GenaSoto/SpringBoot
cd /home/GenaSoto/SpringBoot
```

-- or --

```
md C:\SpringBoot
cd C:\SpringBoot
```

2. Clone the [Spring Boot on Docker Getting Started](#) sample project into the directory.

```
git clone https://github.com/spring-guides/gs-spring-boot-docker.git
```

3. Change directory to the completed project; for example:

```
cd gs-spring-boot-docker/complete
```

-- or --

```
cd gs-spring-boot-docker\complete
```

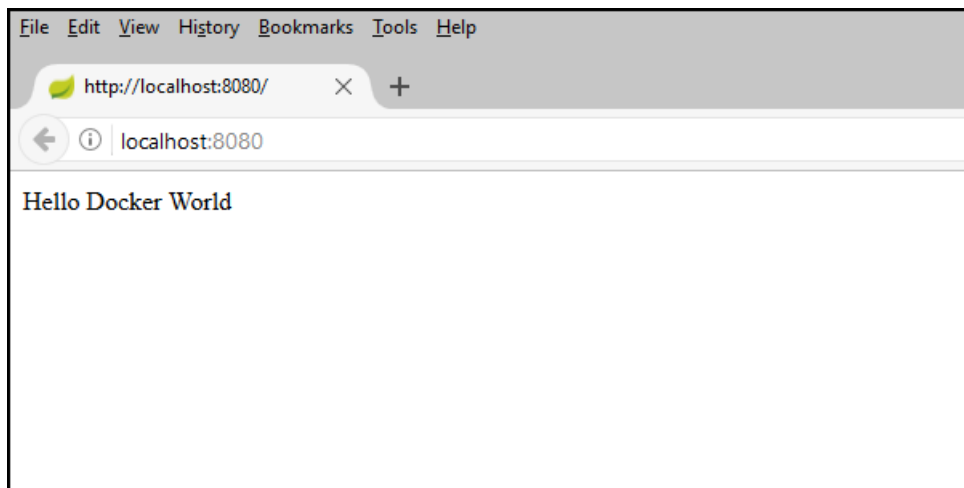
4. Use Maven to build and run the sample app.

```
mvn clean package spring-boot:run
```

5. Test the web app by browsing to <http://localhost:8080>, or with the following `curl` command:

```
curl http://localhost:8080
```

You should see a **Hello Docker World** message displayed.



Install the Kubernetes command-line interface and create an Azure resource group using the Azure CLI

1. Open a command prompt.
2. Type the following command to log in to your Azure account:

```
az login
```

Follow the instructions to complete the login process

The Azure CLI will display a list of your accounts; for example:

```
[
  {
    "cloudName": "AzureCloud",
    "id": "00000000-0000-0000-0000-000000000000",
    "isDefault": false,
    "name": "Windows Azure MSDN - Visual Studio Ultimate",
    "state": "Enabled",
    "tenantId": "00000000-0000-0000-0000-000000000000",
    "user": {
      "name": "Gena.Soto@wingtiptoy.com",
      "type": "user"
    }
  }
]
```

3. If you do not already have the Kubernetes command-line interface (`kubectl`) installed, you can install using the Azure CLI; for example:

```
az acs kubernetes install-cli
```

NOTE

Linux users may have to prefix this command with `sudo` since it deploys the Kubernetes CLI to `/usr/local/bin`.

If you already have `kubectl` installed and your version of `kubectl` is too old, you may see an error message similar to the following example when you attempt to complete the steps listed later in this article:

```
error: group map[autoscaling:0x00000000 batch:0x00000000 certificates.k8s.io:0x00000000 extensions:0x00000000 storage.k8s.io:0x00000000 apps:0x00000000 authentication.k8s.io:0x00000000 policy:0x00000000 rbac.authorization.k8s.io:0x00000000 federation:0x00000000 authorization.k8s.io:0x00000000 componentconfig:0x00000000] is already registered
```

If this happens, you will need to reinstall `kubectl` to update your version.

4. Create a resource group for the Azure resources that you will use in this tutorial; for example:

```
az group create --name=wingtiptoy-kubernetes --location=westeurope
```

Where:

- *wingtiptoy-kubernetes* is a unique name for your resource group
- *westeurope* is an appropriate geographic location for your application

The Azure CLI will display the results of your resource group creation; for example:

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

Create a Kubernetes cluster using the Azure CLI

1. Create a Kubernetes cluster in your new resource group; for example:

```
az acs create --orchestrator-type kubernetes --resource-group wingtiptoy-kubernetes --name wingtiptoy-cluster --generate-ssh-keys --dns-prefix=wingtiptoy
```

Where:

- *wingtiptoy-kubernetes* is the name of your resource group from earlier in this article
- *wingtiptoy-cluster* is a unique name for your Kubernetes cluster
- *wingtiptoy* is a unique name DNS name for your application

The Azure CLI will display the results of your cluster creation; for example:

```
{
  "id": "/subscriptions/00000000-0000-0000-0000-000000000000/resourceGroups/wingtiptoy-kubernetes/providers/Microsoft.Resources/deployments/azurecli0000000000.0000000000",
  "name": "azurecli0000000000.0000000000",
  "properties": {
    "correlationId": "00000000-0000-0000-0000-000000000000",
    "debugSetting": null,
    "dependencies": [],
    "mode": "Incremental",
    "outputs": {
      "masterFQDN": {
        "type": "String",
        "value": "wingtiptoy-smgmt.westeurope.cloudapp.azure.com"
      },
      "sshMaster0": {
        "type": "String",
        "value": "ssh azureuser@wingtiptoy-smgmt.westeurope.cloudapp.azure.com -A -p 22"
      }
    },
    "parameters": {
      "clientSecret": {
        "type": "SecureString"
      }
    },
    "parametersLink": null,
    "providers": [
      {
        "id": null,
        "namespace": "Microsoft.ContainerService",
        "registrationState": null,
        "resourceTypes": [
          {
            "aliases": null,
            "apiVersions": null,
            "locations": [
              "westeurope"
            ],
            "properties": null,
            "resourceType": "containerServices"
          }
        ]
      }
    ],
    "provisioningState": "Succeeded",
    "template": null,
    "templateLink": null,
    "timestamp": "2017-09-15T01:00:00.000000+00:00"
  },
  "resourceGroup": "wingtiptoy-kubernetes"
}
```

2. Download your credentials for your new Kubernetes cluster; for example:

```
az acs kubernetes get-credentials --resource-group=wingtiptoy-kubernetes --name wingtiptoy-cluster
```

3. Verify your connection with the following command:

```
kubectl get nodes
```

You should see a list of nodes and statuses like the following example:

NAME	STATUS	AGE	VERSION
k8s-agent-00000000-0	Ready	5h	v1.6.6
k8s-agent-00000000-1	Ready	5h	v1.6.6
k8s-agent-00000000-2	Ready	5h	v1.6.6
k8s-master-00000000-0	Ready,SchedulingDisabled	5h	v1.6.6

Create a private Azure container registry using the Azure CLI

1. Create a private Azure container registry in your resource group to host your Docker image; for example:

```
az acr create --admin-enabled --resource-group wingtiptoy-kubernetes --location westeurope --name wingtiptoyregistry --sku Basic
```

Where:

- *wingtiptoy-kubernetes* is the name of your resource group from earlier in this article
- *wingtiptoyregistry* is a unique name for your private registry
- *westeurope* is an appropriate geographic location for your application

The Azure CLI will display the results of your registry creation; for example:

```
{
  "adminUserEnabled": true,
  "creationDate": "2017-09-15T01:00:00.000000+00:00",
  "id": "/subscriptions/00000000-0000-0000-0000-000000000000/resourceGroups/wingtiptoy-kubernetes/providers/Microsoft.ContainerRegistry/registries/wingtiptoyregistry",
  "location": "westeurope",
  "loginServer": "wingtiptoyregistry.azurecr.io",
  "name": "wingtiptoyregistry",
  "provisioningState": "Succeeded",
  "resourceGroup": "wingtiptoy-kubernetes",
  "sku": {
    "name": "Basic",
    "tier": "Basic"
  },
  "storageAccount": {
    "name": "wingtiptoyregistr000000"
  },
  "tags": {},
  "type": "Microsoft.ContainerRegistry/registries"
}
```

2. Retrieve the password for your container registry from the Azure CLI.

```
az acr credential show --name wingtiptoyregistry --query passwords[0]
```

The Azure CLI will display the password for your registry; for example:

```
{
  "name": "password",
  "value": "AbCdEfGhIjKlMnOpQrStUvWxYz"
}
```

3. Navigate to the configuration directory for your Maven installation (default `~/m2/` or `C:\Users\username.m2`) and open the *settings.xml* file with a text editor.
4. Add your Azure Container Registry URL, username and password to a new `<server>` collection in the

`settings.xml` file. The `id` and `username` are the name of the registry. Use the `password` value from the previous command (without quotes).

```
<servers>
  <server>
    <id>wingtiptoyregistry.azurecr.io</id>
    <username>wingtiptoyregistry</username>
    <password>AbCdEfGhIjKlMnOpQrStUvWxYz</password>
  </server>
</servers>
```

5. Navigate to the completed project directory for your Spring Boot application (for example, "`C:\SpringBoot\gs-spring-boot-docker\complete`" or "`/home/GenaSoto/SpringBoot/gs-spring-boot-docker/complete`"), and open the `pom.xml` file with a text editor.
6. Update the `<properties>` collection in the `pom.xml` file with the login server value for your Azure Container Registry.

```
<properties>
  <docker.image.prefix>wingtiptoyregistry.azurecr.io</docker.image.prefix>
  <java.version>1.8</java.version>
</properties>
```

7. Update the `<plugins>` collection in the `pom.xml` file so that the `<plugin>` contains the login server address and registry name for your Azure Container Registry.

```
<plugin>
  <groupId>com.spotify</groupId>
  <artifactId>dockerfile-maven-plugin</artifactId>
  <version>1.3.4</version>
  <configuration>
    <repository>${docker.image.prefix}/${project.artifactId}</repository>
    <serverId>${docker.image.prefix}</serverId>
    <registryUrl>https://${docker.image.prefix}</registryUrl>
  </configuration>
</plugin>
```

8. Navigate to the completed project directory for your Spring Boot application, and run the following Maven command to build the Docker container and push the image to your registry:

```
mvn package dockerfile:build -DpushImage
```

Maven will display the results of your build; for example:

```
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 38.113 s
[INFO] Finished at: 2017-09-15T02:00:00-07:00
[INFO] Final Memory: 47M/338M
[INFO] -----
```

Configure your Spring Boot app to use the Fabric8 Maven plugin

1. Navigate to the completed project directory for your Spring Boot application, (for example: "`C:\SpringBoot\gs-spring-boot-docker\complete`" or "`/home/GenaSoto/SpringBoot/gs-spring-boot-`

docker/complete"), and open the *pom.xml* file with a text editor.

2. Update the `<plugins>` collection in the *pom.xml* file to add the Fabric8 Maven plugin:

```
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-maven-plugin</artifactId>
  <version>3.5.30</version>
  <configuration>
    <ignoreServices>false</ignoreServices>
    <registry>${docker.image.prefix}</registry>
  </configuration>
</plugin>
```

3. Navigate to the main source directory for your Spring Boot application, (for example: "C:\SpringBoot\gs-spring-boot-docker\complete\src\main" or "/home/GenaSoto/SpringBoot/gs-spring-boot-docker/complete/src/main"), and create a new folder named "fabric8".
4. Create three YAML fragment files in the new *fabric8* folder:
 - a. Create a file named **deployment.yml** with the following contents:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: ${project.artifactId}
  labels:
    run: gs-spring-boot-docker
spec:
  replicas: 1
  selector:
    matchLabels:
      run: gs-spring-boot-docker
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate
  template:
    metadata:
      labels:
        run: gs-spring-boot-docker
    spec:
      containers:
        - image: ${docker.image.prefix}/${project.artifactId}:latest
          name: gs-spring-boot-docker
          imagePullPolicy: Always
          ports:
            - containerPort: 8080
      imagePullSecrets:
        - name: mysecrets
```

- b. Create a file named **secrets.yml** with the following contents:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecrets
  namespace: default
  annotations:
    maven.fabric8.io/dockerServerId: ${docker.image.prefix}
type: kubernetes.io/dockercfg
```


c. Create a file named **service.yml** with the following contents:

```
apiVersion: v1
kind: Service
metadata:
  name: ${project.artifactId}
  labels:
    expose: "true"
spec:
  ports:
    - port: 80
      targetPort: 8080
  type: LoadBalancer
```

5. Run the following Maven command to build the Kubernetes resource list file:

```
mvn fabric8:resource
```

This command merges all Kubernetes resource yaml files under the *src/main/fabric8* folder to a YAML file that contains a Kubernetes resource list, which can be applied to Kubernetes cluster directly or export to a helm chart.

Maven will display the results of your build; for example:

```
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 16.744 s
[INFO] Finished at: 2017-09-15T02:35:00-07:00
[INFO] Final Memory: 30M/290M
[INFO] -----
```

6. Run the following Maven command to apply the resource list file to your Kubernetes cluster:

```
mvn fabric8:apply
```

Maven will display the results of your build; for example:

```
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 14.814 s
[INFO] Finished at: 2017-09-15T02:41:00-07:00
[INFO] Final Memory: 35M/288M
[INFO] -----
```

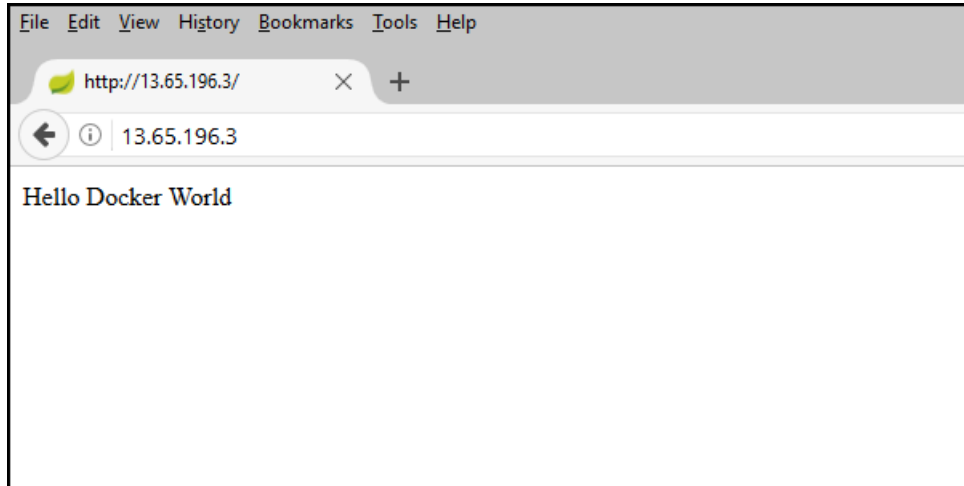
7. Once the app is deployed to the cluster, query the external IP address using the `kubectl` application; for example:

```
kubectl get svc -w
```

`kubectl` will display your internal and external IP addresses; for example:

NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	10.0.0.1	<none>	443/TCP	19h
gs-spring-boot-docker	10.0.242.8	13.65.196.3	80:31215/TCP	3m

You can use the external IP address to open your application in a web browser.



Delete your Kubernetes cluster

When your Kubernetes cluster is no longer needed, you can use the `az group delete` command to remove the resource group, which will remove all of its related resources; for example:

```
az group delete --name wingtiptoy-kubernetes --yes --no-wait
```

Next steps

For more information about using Spring Boot applications on Azure, see the following articles:

- [Deploy a Spring Boot Application to the Azure App Service](#)
- [Deploy a Spring Boot application on Linux in the Azure Container Service](#)
- [Deploy a Spring Boot Application on a Kubernetes Cluster in the Azure Container Service](#)

For more information about using Azure with Java, see the [Azure Java Developer Center](#) and the [Java Tools for Visual Studio Team Services](#).

For further details about the Spring Boot on Docker sample project, see [Spring Boot on Docker Getting Started](#).

For help with getting started with your own Spring Boot applications, see the **Spring Initializr** at <https://start.spring.io/>.

For more information about getting started with creating a simple Spring Boot application, see the Spring Initializr at <https://start.spring.io/>.

For additional examples for how to use custom Docker images with Azure, see [Using a custom Docker image for Azure Web App on Linux](#).

Container Service frequently asked questions

8/31/2017 • 4 min to read • [Edit Online](#)

Orchestrators

Which container orchestrators do you support on Azure Container Service?

There is support for open-source DC/OS, Docker Swarm, and Kubernetes. For more information, see the [Overview](#).

Do you support Docker Swarm mode?

Currently Swarm mode is not supported, but it is on the service roadmap.

Does Azure Container Service support Windows containers?

Currently Linux containers are supported with all orchestrators. Support for Windows containers with Kubernetes is in preview.

Do you recommend a specific orchestrator in Azure Container Service?

Generally we do not recommend a specific orchestrator. If you have experience with one of the supported orchestrators, you can apply that experience in Azure Container Service. Data trends suggest, however, that DC/OS is production proven for Big Data and IoT workloads, Kubernetes is suited for cloud-native workloads, and Docker Swarm is known for its integration with Docker tools and easy learning curve.

Depending on your scenario, you can also build and manage custom container solutions with other Azure services. These services include [Virtual Machines](#), [Service Fabric](#), [Web Apps](#), and [Batch](#).

What is the difference between Azure Container Service and ACS Engine?

Azure Container Service is an SLA-backed Azure service with features in the Azure portal, Azure command-line tools, and Azure APIs. The service enables you to quickly implement and manage clusters running standard container orchestration tools with a relatively small number of configuration choices.

[ACS Engine](#) is an open-source project that enables power users to customize the cluster configuration at every level. This ability to alter the configuration of both infrastructure and software means that we offer no SLA for ACS Engine. Support is handled through the open-source project on GitHub rather than through official Microsoft channels.

For additional details please refer to our [support policy for containers](#).

Cluster management

How do I create SSH keys for my cluster?

You can use standard tools on your operating system to create an SSH RSA public and private key pair for authentication against the Linux virtual machines for your cluster. For steps, see the [OS X and Linux](#) or [Windows](#) guidance.

If you use [Azure CLI 2.0 commands](#) to deploy a container service cluster, SSH keys can be automatically generated for your cluster.

How do I create a service principal for my Kubernetes cluster?

An Azure Active Directory service principal ID and password are also needed to create a Kubernetes cluster in Azure Container Service. For more information, see [About the service principal for a Kubernetes cluster](#).

If you use [Azure CLI 2.0 commands](#) to deploy a Kubernetes cluster, service principal credentials can be automatically generated for your cluster.

How large a cluster can I create?

You can create a cluster with 1, 3, or 5 master nodes. You can choose up to 100 agent nodes.

IMPORTANT

For larger clusters and depending on the VM size you choose for your nodes, you might need to increase the cores quota in your subscription. To request a quota increase, open an [online customer support request](#) at no charge. If you're using an [Azure free account](#), you can use only a limited number of Azure compute cores.

How do I increase the number of masters after a cluster is created?

Once the cluster is created, the number of masters is fixed and cannot be changed. During the creation of the cluster, you should ideally select multiple masters for high availability.

How do I increase the number of agents after a cluster is created?

You can scale the number of agents in the cluster by using the Azure portal or command-line tools. See [Scale an Azure Container Service cluster](#).

What are the URLs of my masters and agents?

The URLs of cluster resources in Azure Container Service are based on the DNS name prefix you supply and the name of the Azure region you chose for deployment. For example, the fully qualified domain name (FQDN) of the master node is of this form:

```
DNSNamePrefix.AzureRegion.cloudapp.azure.net
```

You can find commonly used URLs for your cluster in the Azure portal, the Azure Resource Explorer, or other Azure tools.

How do I tell which orchestrator version is running in my cluster?

- DC/OS: See the [Mesosphere documentation](#)
- Docker Swarm: Run `docker version`
- Kubernetes: Run `kubect1 version`

How do I upgrade the orchestrator after deployment?

Currently, Azure Container Service doesn't provide tools to upgrade the version of the orchestrator you deployed on your cluster. If Container Service supports a later version, you can deploy a new cluster. Another option is to use orchestrator-specific tools if they are available to upgrade a cluster in-place. For example, see [DC/OS Upgrading](#).

Where do I find the SSH connection string to my cluster?

You can find the connection string in the Azure portal, or by using Azure command-line tools.

1. In the portal, navigate to the resource group for the cluster deployment.
2. Click **Overview** and click the link for **Deployments** under **Essentials**.
3. In the **Deployment history** blade, click the deployment that has a name beginning with **microsoft-ac**s followed by a deployment date. Example: microsoft-ac-201701310000.
4. On the **Summary** page, under **Outputs**, several cluster links are provided. **SSHMaster0** provides an SSH connection string to the first master in your container service cluster.

As previously noted, you can also use Azure tools to find the FQDN of the master. Make an SSH connection to the master using the FQDN of the master and the user name you specified when creating the cluster. For example:

```
ssh userName@masterFQDN -A -p 22
```

For more information, see [Connect to an Azure Container Service cluster](#).

Next steps

- [Learn more](#) about Azure Container Service.
- Deploy a container service cluster using the [portal](#) or [Azure CLI 2.0](#).